ES14 (2023)

e.g ES6, ES7 → new version

defines core features of language
syntax, data types, operators, objects

Javascript → interactivity

ECMASCRIPT → rules of the language, scripting language specification

- it is a standard on which Javascript is based. It was created to ensure different documents in Javascript are actually talking about the same language

- ensures compatibility & consistency across different Javascript engines (like chrome's V8, Firefox's spidermonkey)

- Javascript is very liberal in what it allows

How to execute Javascript →

Javascript can be executed right inside one's browser
(~~cons~~ inspect ~~Javascr~~, webpage & go to console)

in a runtime like Node.js which can be installed and used to run Javascript code (run javascript outside browsers typically servers)

inside script tag in HTML document

- <body>
  <script>
  </script>
  </body>
- <script src="javascript.js"></script>

in body

Ryan Dahl - run Javascript outside the browser.

Node.js → allows developers to run Javascript on server side

→ built on google's V8 engine

inline Javascript

```
<button onclick="alert('Hello!')">
    clickme
</button>
```

Dynamically typed languages →

change type of variable on runtime

```
var a = 67;   → old
let a = 53;   → new
a = 'harry'
console.log(a)
```

(67: literal → meaning fixed value written as it is)

Rules for variable names

→ letters, digits, underscores & $ sign allowed

→ must begin with a $ _ or a letter

→ reserved words cannot be used as variable names

Before ES6 → var was used
↳ can introduce bugs therefore not commonly used

Scripting language → programming language designed for writing short programs or scripts that automate tasks or run inside other software

Hoisting → moving declarations to the top of their scope (only declaration not value)

let → block scoped variable

var vs let in javascript -

① var is globally scoped whereas let and const are block scoped

```
var b = "Max";
{
    var b = "this";
    console.log(b);
}
console.log(b);
```

output → this
this

② var can be updated & re-declared within its scope

```
{
    var a = 23
    var a = "Hello"
}
```

this allowed

③ let can be updated but not re declared

④ const can neither be updated nor be redeclared

b = 4; → without Keyword
- globally scoped
- throws an error in strict mode

⑤ var variables are initialized with undefined whereas let & const variables are not initialized

if you try to access before line of declaration

var	let
undefined	Reference Error (they live in temporal dead zone) (∴ not automatically assign undefined value)

Primitive data types
- B - Boolean
- B - Bigint
- U - undefined
- N - Null
- N → Number
- S → String
- S → symbol

Object → key-value pairs

allows to safely store & operate on large integers beyond limit of Number type

Number → 64 bit floating point
+ (2^53 - 1) to (2^53 - 1)

symbol

let s = Symbol("This is Symbol")

BigInt →

- add n to the end of an integer
- use bigint constructor

const a = BigInt("123 - - - - - - 5");

cannot mix BigInt & Number directly

Typeof →

to get type of a variable

console.Log(typeof s);

→ Symbol

Some important -

when number is added to string it results in concatenation

we can add or change value in a const object because const var is reference to an object in memory so changing object does not mean changing the const variable but changing the variable to point to another object or assign new value leads to error

Expressions & Conditionals →

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

e.g 77 or "Harry"

1 - Arithmetic operator →

+ add

-

*

* *

/ → 20/8 ⇒ 2.5

%

+ +

- -

2. assignment operators

=

+ =

- =

* =

/ =

% =

* * =

3. comparison operators

= = → checks only value

! =

= = = → checks data type also

! = =

>

<

< =

> =

2.

4. logical operators

&& logical and
|| logical or
! logical not

→ comments // & /* */
multiline comment

Conditional expressions →

- prompt → ask question
- alert → tells you

let a = prompt(" what is your age")

- CONSOLE →
→ what is your age
→ 30

- alert(" Hey")
Hey displayed on webpage

Typecasting
String to Number →
let a = prompt(" Hey whats you age?")
a = Number.parseInt(a)

- if else
if ( )
{ }
else if ( )
{ }
else if ( )
{ }
else
{ }

- switch

```
const expr = 'Papayas';
switch (expr) {
    case 'oranges':
        console.log('...');
        break;

    case 'mangoes':
    case 'papayas':
        console.log('...');
        break;

    default:
        console.log('sorry we are out of ${expr}.');
}
```

if break is not included, all the following cases of matched case will run. including default

- ternary

console.log("You can", a<18? "not drink" : "drink")
;

Object.values (obj) → array of values of obj

For loops →

```
for (let i=0; i<5; i++) {
    console.log(i)
}
```

In console.log '+' or ',' can be used to separate comments and variables

For in loop →

for in loop for keys of object and used in arrays

```
let obj = {
    a:67,
    b:88,
    c:93,
    d:82
}

for (let x in obj)
{
    console.log ("marks of ", x, "are", obj[x]
}
```

For of loop →

for of loop used for values in objects and with arrays and string

obj needs to be iterable

Javascript is interpreted language but in modern environments it's interpreted & compiled (JIT compiled) → compiled to machine code at runtime.

```
for (let val of object.values(obj))
{
    console.log(val)
}

for (let x of "Mayeet")
{
    console.log(x);
}
```

Interpreted part of code is compiled

while loop →

```
let i=0;
while (i<3) {
    console.log (i);
    i++;
}

let i=10;
do {
    console.log(i);
    i++;
} while (i<3)
```

Function → ~~onePlusAvg~~

```
function onePlusAvg (x, y){
    return 1 + (x+y)/2
}

const sum = (p,q) => {
    return p+q;
}
```

✓ Math.round() → for roundoff

PAGE No.
DATE

✓ Git

→ git init → creates a hidden .git/ directory to track changes

→ git remote add origin https: —.

links your local repo to remote repo hosted on Github

origin → nickname for remote ~~stack~~
URL → remote repo address

→ git branch -M main
renames current branch from master to main

- M forces rename even if main already exist

effect → your branch name now matches Github's default main

→ git push -u origin main

pushes local code to remote github for first time

origin → remote server we linked earlier

main → branch we are pushing

-u → sets the branch we are as default tracking branch

PAGE No.
DATE

- branch is separate timeline of code
- main branch is official published version

✓ object.keys(marks) → array of keys of marks
✓ object.values(marks) → array of values of marks

✓ Strings

~~let name~~ =
can be created using both single & double quotes

let name = "Max"
'Max'

name.length → prints length of string

→ name[0] {M}
→ name[1] {A}
→ name[2] {X}

✓ Template literals →

~~let boy1 = " Isamod~~

Template literals use backticks instead of quotes to define a string

let name = `Harry`

with template literals it is possible to use both single as well as double quotes inside a string

length → property

toUpperCase → Method

we can insert variables directly in template literal → string interpolation

let a = ` This is ${name}`

Escape sequence character

~~let name~~ =

let str = ' Your's '

we can use single quote escape sequence to solve the problem

let str = ' Your \'s '

length of this will be 6 only

\' → single character

" Your \"s "

Others are

\n → newline

\t → tab

\r → carriage return

String Methods → name = "Harry";

1. toUpperCase

console.log (name.toUpperCase())

2. toLowerCase

console.log(name. toLowerCase())

3. slice

console. Log (name. slice (2,4))

→ if only starting index is given then it goes upto end of string

not included → end index is

4. replace

console. Log (name. replace(" Har", "Per"))

if " har" then ~~only ar will b~~ nothing will change because it does not exist in name

5. concat → add, can also use +

friend = " Naman

console. Log (name. concat (" is a friend of ", friend, " OK"))

6. trim →

let friend2 = " Meena "

console. Log (friend 2. trim())

trim starting and ending space

// strings are immutable therefore every method returns new string and

→ you cannot change like str[3] = 'f'; it will not reflect on str;

PAGE No.
DATE

⑦ includes

let str = 'fox'
let sentence = 'fox in the box'
console.log (sentence.includes (str));

→ true

⑧ starts With / endsWith

let str = "Hello world"

→ true console.log (str.startsWith ("Hello"));
→ true console.log (str.endsWith ("world"));

- startsWith (substring, position)
- endsWith ("substring, length)

→ it assumes string of this length only

console.log (str.endsWith ("Hello", 5))

true

it assumes first 5 characters which are Hello

✓ Chapter → 5 Arrays

let arr = [42, 36, 11, 9, 83]

let x = ["Not present", false, Null, 4, 8]

PAGE No.
DATE

console.log (x[6])

→ undefined

→ ✓ length property →
console.log (x.length)

→ we can access and assign indexes which are not yet defined

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

arr[4] = 5

arr[6] = 7

console.log (arr) =>

→ [1, 2, 3, 4, 5, <1 empty item>, 7]

→ arrays are mutable
→ arrays can be changed
→ ✓ type of array is object

→ arrays can hold many values under a single name

✓ Methods of array →

~~let num = E~~

① toString () → converts array to a comma separated string and return.
→ original array does not change

let x = [7, 5, 9, 11, 1]
let s = x.toString ();
console.log (typeof x) object

→ console.log (s[1]) → 2

→ console.log (s) → 7, 5, 9, 1

→ console.log (type of s) → string

② join() → join the elements with a joiner in between and return a string
→ default joiner ","

③ pop() → updates the original array
→ returns the popped value
→ if pop an empty array the return popped value is undefined

④ push
→ modifies the original array
→ add a new element at the end of the array
→ return the new array length

⑤ shift → - modifies the original array
- Remove the first element and returns it

⑥ unshift → - add the element to beginning of ~~array~~ original array
- returns the length of new array

method, not an operator

⑦ delete → elements can be deleted using delete operator

let d = [7, 8, 9, 10]

delete d[1]

→ ~~up~~ deletes the element from original array
→ length remains same

⑧ concat → returns an array with adding elements from other arrays

let x_1 = a_1.concat(a_2, a_3)

⑨ sort() → sort an array alphabetically (considering string)
→ modifies the original array
→ takes an optional compare function
→

7 5 6 1 2

let compare = (a, b) => {
return b - a
}

let num = [55 1, 22, 3, 5, 6, 7]
num.sort (compare)

(10) reverse() → reverse original array

(11) splice() → used to add new items to an array
→ & delete also
→ modifies the array

const numbers = [1, 2, 3, 4, 5]
numbers.splice(2, 1, 23, 24)

returns deleted items
modifies the array

position to add — Number of elements to remove — elements to be added

(12) slice() → slice the array and returns it
~~num.slice(3)~~ → num.slice(3, 5)

included index — not included index

using loops with array →

(1) for Each loop

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

```
const a = [1, 2, 3]
a.forEach ((value, index, array) => {
  }
```

document.getElementsByClassName("sidebar-DO664")
HTML collection

(2) for of — for (let i of num) { }

(3) for in — for (let i in num) { i will give key }

(4) Array.from → used to create an array from another ~~object~~ array like objects

Array.from("Harry")

Map, filter & Reduce — // does not modify original array

(1) Map() → creates a new array by performing some operations on each array element

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

(2) filter an array with values that passes a test. Creates a new array

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

- map ( ) → can use a function which re-
- filter ( ) → can use a function
- reduce ( ) → e.g greater than 4

③ reduce method → reduces an array to a single value

```
const n = [1, 8, 7, 11]

let sum = ~~number~~ n.reduce((n1,n2) =>
        {
            return n1+n2
        })
```

filter & map callback func (val, index, array)
{ }

~~reduce fca~~ →

array.reduce (callback fun, initial value)

→ callback function (accumulator, currentvalue, currentindex, array)

e.g const sum = numbers.reduce (acc, curr) => acc+curr, 0);

Math.random() → return for random number between 0 & 1

e.g 0.63587 ...
// Math.floor
// Math.ceil

returns + one to keep the element & initial value

Javascript in the browser

- Browser has an embedded engine called the Javascript engine
- Javascript's ability in the browser is very limited to protect the user's safety

Developer tools →

Elements → All HTML elements

Console → all errors + logs → console.log

Network → all network requests

If Javascript

- script tag in HTML file
- script tag with source and javascript code in js that source
  - → caching → browser save these files to save time
  - → separation of concerns multiple js files for multiple functionalities

console.log(console) → gives console object & methods

Console object Methods →

→ ~~cons~~ assert()
clear()
log()
table()
warn()
error()
info()

① console.assert(5>53) → to test if a condition is true
→ Assertion failed → in red color ⊗
console.assert(555>53)
→ undefined → returning value, means no problem
occurs in console.log also

② console.clear() → clear the console

③ console.table() → gives table of object
e.g console.table(obj)

④ console.warn() → gives warning
console.warn("Hey please dont drink soda")
⚠ in yellow print

in javascript object is collection of key-value pairs. Method is a function inside

⑤ console.error() → gives error in red

⑥ console.info() → similar to log

warn → goes in warning tab
error → goes in error tab
info → goes in info tab

⑦ console.log() → print output to the browser console
→ used for debugging
→ returns undefined

⑧ console.time() →
console.timeEnd() → gives time between time line of code & timeEnd line of code

prompt() → takes input as a string
→ so convert to number
inp = prompt("Hi", "No") → optional default value

confirm, prompt, alert → pause execution of the script

confirm:- shows a message and wait for user to press ok or cancel. Returns true for ok and false for cancel

Window object, BOM & DOM –

Window
- DOM
- BOM
- Javascript core

document object model

पूरे HTML page को एक Javascript object बना दिया गया है और उसे document नाम दे दिया गया है।

window object represents browser window and provides methods to control it is a global object

```
window.console.log("Hi")
window.document
```

(1) Document Object Model →
Document represents page content as HTML

Document.body → page body as JS object

```
document.body.style.background = "green"
```

↳ changes page background to green

(2) Browser Object Model (BOM) →

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

alert/ confirm/ prompt are part of BOM

```
location.href = "https://codewithharry.com"
```

redirect to website

Walking the DOM →

→ document.getElementByClassName("cursor-pointer")[3].style.color = "red"

→ DOM tree refers to the HTML page where all the nodes are objects

There can be three types of nodes

in the DOM tree →
1) 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 node

Auto correction →

if an erroneous HTML is encountered by the browser. it tends to correct it. for example, if we put something after the body, it is automatically moved inside the body

Walking the DOM →

```
<html>
<head>
<title> hello </title>
</head>
<body>
</body>
</html>
```

document.body → page body tag

document → document HTML → page HTML tag

document.head

DOM NAVIGATION

document.title → returns string

✓ document.body will return Null if javascript is written before body tag ~~in say like~~

```
<script>
console.log document.body
</script>
<body>
</body>
```

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

First Child, Lastchild & child Nodes
element.firstchild → first child element

element.lastchild → last child element

element.childNodes → All child nodes

e.g
```
<body>
<div></div>
</body>
```

body.firstchild → null be blank space → text node
div will be second

== $0 → element you selected can directly use on console

→ elem.childNodes[0] === elem.firstChild

→ elem.chilNodes[elem.childNodes.length-1] === elem.lastChild

elem.hasChildNodes()

→ check whether there any child nodes

Note → childNodes too is a collection we can use Array.from(collection) to convert it into an Array

DOM collections → group of DOM nodes returned by various DOM methods and properties

→ They are read only

→ They are live elem.childNodes variable (reference) will automatically update if childNodes of elem is changed

→ They are iterable using for...of loop

all type of nodes → NodeList

HTML collection → only element nodes

document.querySelectorAll()
childNodes

- getElementsByClassName()
- getElementsByTagName()
- children

document.documentElement → html tag or root element

Siblings and the parent →

siblings are nodes that are children of same parents

→ <body> — nextsibling of → <head>

← — previous sibling of —

✓ alert(document.documentElement.parentNode); // document

✓ alert(document.documentElement.parentElement); // Element

Element only Navigation →

* Node — element (element node) ⇒ text node & comment Node & others

→ firstChild → gives the text node as first

→ firstElementChild → ignores the text node as it is not element and gives first element which is div

sometimes we don't want text or comment nodes, some links only take Element nodes into account, for eg

document.previousElementSibling
document.nextElementSibling
document.firstElementChild
document.lastElementChild

Table links →

certain DOM elements may provide additional properties specific to their types for convenience.

Table elements support the following properties

table.rows → collection of tr elements
table.caption → reference to <caption>
table.tHead → reference to <thead>
table.tFoot → reference to <tfoot>
table.tBodies → collection of <tbody> elements
tbody.rows → collection of <tr> inside

table.rows[0]

tr.cells → collection of td & th
tr.sectionRowIndex → index of tr inside enclosing element
tr.rowIndex → Row number starting from 0
tr.cellIndex → no. of cells inside enclosing <tr>
or
th

searching the DOM →

DOM navigation → when elements are close to each other

if elements are not close to each other →

1. document.getElementById()
   get element with a given "id" attribute

2. document.querySelectorAll()
   Returns all elements inside an element matching the given CSS selectors
   e.g
   let x = document.querySelectorAll(".card-title")

   class selector (we can use # for id, div, ul, li, etc)

   x[0].style.color = "blue"

   returns a node list

3. ~~document.querySelector(".this").style.CSS = "~~

PAGE No.
DATE

③ document.querySelector() →
Returns the first element for the given CSS selector

A efficient version of elem.querySelectorAll(CSS)[0]

e.g document.querySelector(".this")

④ document.getElementByTagName()
Returns element with the given tag name

e.g document.getElementBy Tag Name("a")
↳ anchor tag

⑤ document.getElementsBy ClassName()
Return elements that have the given CSS class

⑥ document.getElementsBy Name()
search elements by the name attribute
↳ present as an option in elements properties

PAGE No.
DATE

Matches, closest and contains methods →

1 → Three important methods to search the DOM

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

② elem.closest(CSS) → look for the nearest ancestor that matches the given CSS selector. The element's itself is checked also checked

③ elemA.contains(elemB)
returns true if elemB is inside elemA (~~a descendant of elemA~~) or when elemA == elemB

→ If only <tr> & <td> used in table Javascript auto corrects and put it in tbody

→ to change style for a Node list, we use forEach to select each element

→ right click + change all occurences in VS code

→ ctrl+f → find & replace
ctrl+h

Chapter - 8 → Events & other DOM properties

1. console.dir function

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

```
let ele = document.getElementsByTagName('span')[0];

console.log(ele)
→ <span> Hey I am span </span>
console.dir(ele)
→ ▷ span
```

2. tagName / nodeName

used to read tag name of an element
tagName → only exist for Element nodes
nodeName → defined for any node

```
console.log(document.firstChild.nodeName)
console.log(document.body.firstElementChild.nodeName)
```

→ #text
→ SPAN

3. innerHTML and outerHTML

→ `<p id="first"><b>something</b></p>`
→ first.innerHTML
`<b>something</b>`
→ first.innerHTML = "else"

innerHTML property allows to get the HTML inside the element as a string

outerHTML contains full HTML
innerHTML + the element itself

innerHTML is valid only for element nodes. for other we can use nodeValue or data

first.outerHTML
→ `<p id="first"><b>something</b></p>`

4. textContent →

provides access to the text inside the element → only text, minus all tags

5. the hidden property →

hidden attribute in HTML makes element hidden

DOM property specify whether the element is hidden or not

ele.hidden
→ false
ele.hidden = true
→ gets hidden

1800 × 400000 = 720,000,000

Attribute Methods →

① elem.hasAttribute(name) → method to check for existence of an attribute

→ ele.hasAttribute('id')
→ true

② elem.getAttribute(name) → Method used to get the value of an attribute

ele.getAttribute('id')
→ first

③ elem.setAttribute(name, value)
ele.setAttribute('hidden', 'true')

④ elem.removeAttribute(name)

⑤ elem.attributes → get collection of attributes

data-attributes →

we can always create custom attributes but the ones starting with "data-" are reserved for programmer use. They are available in a property named dataset.

if an element has an attribute named "data-one", it is available as element.dataset.one

<div id="first" data-game="mario data-player="harry">
first = document.getElementById("first")
first.dataset
→ game
player

first.dataset.game
→ mario

Insertion Methods

→ we can do it through innerHTML but for ~~using~~ inserting multiple elements and for loop, we use →

another way →

let div = document.createElement('div')

div.className = "alert"
div.innerHTML = "hello"
document.body.append(div)

some more insertion methods →

1 → node.append(e) → append at end of node
2 → node.prepend(e) → insert at beginning of a node

when making css, you need to use border to understand making

3) node.before(e) → insert before node
4) node.after(e) → insert after node

5) node.replaceWith(e) → replaces node with the given node

insertAdjacentHTML/Text/Element

e.g. First.insertAdjacentHTML('beforeend' beforeend);

1 → Method is used to insert HTML. The first parameter is a code word, specifying where to insert. Must be one of the following:-

1 - 'beforebegin' - insert HTML immediately before element
2 - 'afterbegin' - insert HTML into element at the begin
3 - 'before end' - insert HTML into element at the end
4 - 'after end' - insert HTML immediately after element

Node Removal →

removes a node

→ ~~let i~~ id1.remove()

ClassName & classlist →

- if we assign something to elem.elem.className, it replaces the whole string of classes

- if we want to add/remove/toggle a single class

① e.g elem.classlist.add/remove("class")

② e.g elem.classlist.toggle("class")
- adds the class if it doesn't exist, otherwise removes it

③ elem.classlist.contains("class") - checks for the given class, returns true/false.

Set Timeout and setInterval →

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

Syntax of setTimeout is as follows:

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

returns a timer id — in ms

e.g const sum = (a, b) ⇒ { console.log ("working) * (a+b) ~~return a+b~~ }

returns a+b

setTimeout (sum, 1000, 1, 2)

⇒ working3 → on console after 1 second

→ clearTimeout →
used to cancel the execution in case we change our mind

e.g let timerId = setTimeout (() ⇒ alert ("never"), 5000)

clearTimeout (timerId)
↳ cancel the execution

→ setInterval →
~~it runs~~ same as setTimeout but runs the function not only once, but regularly after the given interval of time

to stop calls, we can use clearInterval

Browser Events

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

some important DOM events are :-
Mouse events → click, right click (context menu), mouseup/mousedown, mouseover/mouseout, mousemove

Keyboard events → keydown and keyup

form element events → submit, focus, etc

Document events → DOM Contentloaded
when DOM is fullyloaded

Javascript के द्वारा HTML onclick वाला take precedence over the inline one i.e Adding a handler with Javascript overwrites the existing handler

e.g HTML file
<div class="container" onmouseenter="console.log ('container के अंदर Mouse आया')
onclick="console.log('container पर click हुआ')

Events can be handled through HTML attributes
<input value="Hey" onclick="alert('hey')"
type="button">

can be another JS function

* script tag at last of body

Events can also be handled through the onclick property

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

addEventListener and removeEventListener

→ addEventListener is used to assign multiple handles to an event

```
element.addEventListener(event, handler)

element.removeEventListener(event, handler)
```

handler must be the same function object for this to work

e.g
```
let x = () => {
        alert("Hello world!")
}

btn.addEventListener('click', x)

// btn.removeEventListener('click', x)
```
↳ removes the event and can be used after a set interval or an if condition

() ⇒ window.location("https://www.google.com")

The Event object

→ when an event happens
→ browser creates an event object
→ puts details into it and passes it as an argument to the handler

```
elem.onclick = function(event){
```
any variable name will work as well
e.g can use e

```
}
```

event.type → Event type
event.currentTarget → Element that handled the event
(e.g painter event)
event.clientX / event.clientY → coordinates of the cursor

To get Date →
```
let a = new Date()
let h = a.getHours()
let m = a.getMinutes()
let s = a.getSeconds()
let d = a.getDate()
console.log(h, m, s, d)
```

height: 100vh → makes the element's height 100% of viewport height
(ex.)

background:

PAGE No.
DATE

Chapter - 9 call backs, promises & async/await

Asynchronous actions → actions that we initiate now & they finish later e.g setTimeout

Synchronous actions → actions that initiate and finish one by one

Callback functions →

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

```
function LoadScript(src, callback){
    let script = document.createElement
                    ('script')
    script.src = src
    script.onload = () => callback(script)
    document.head.append(script)
}

loadscript('https://cdn.harey.com',
        (script) => {
            alert('script is loaded')
            alert(script.src) } )
```

event handler that runs automatically when a webpage or an element has completely loaded

PAGE No.
DATE

- This is called "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

usecase → to run code after an operation completes or to customize behaviour

→ some code takes time and you want to run something only after it finishes

Handling errors →

```
function loadscript(src, callback){
    - - -
    - -
    script.onload = () => callback(null,
                        script);
    script.onerror = () => callback(
            newError("Failed ~~now script failed~~
                there is some error"),
    - - -
}
```

and callback will be like

```
callback(error, src)
{
}
```

```
if (error){
    handles error
        }
    else {
        script loaded
    }
}
```

Pyramid of doom →
when we have callback inside callbacks, the code gets difficult to manage

```
loadscript ((...){
        loadscript ..
            loadscript ...
                loadscript ...
```

as call becomes more nested, the code becomes ~~more~~ deeper and increasingly more difficult to manage, especially if we have real code.
hard to red
hard to debug
hard to mantain

→ also when you pass callback to another function, you lose control

3000 × 1000 =

prom ise ⇒ eventual result of an asynchronous operation

promises ⇒ clearer more manageable code

Introduction to promises →
The solution to the callback hell is promises. A promise is a "promise of code execution". The code either executes or fails, in both the cases the subscriber will be notified

```
let promise = new Promise(function
                (resolve, reject)
{
    //
});
```

predefined in JS engine

resolve and reject are two callbacks provided by javascript itself. They are called like this:-

resolve (value) → if job is finished successfully
reject (error) → if the job fails

promise object returned by new promise constructor has these properties →

1 State → initially pending then changes to either "fulfilled" when resolve is called or "rejected" when reject is called

2 result → initially undefined, then changes to value if resolved resolve(value)

in real world multiple promises run parallely

or error when rejected
reject(error)

// used when processing parallely

consumers: then & catch →

consuming code can recieve the final result of a promise through then & catch

→ promise.then(function (result){ }
function (error){ // handles error
*/ });

can be used in catch also

if we are interested only in successful completions, we can provide only one function argument to then():→

```
let promise = new prose (resolve => {
        settimeout ( () => resolve("done"), 1000 );
    });

promise.then(alert);
```

then(success Handler, Error Handler)

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

∴ then (null, f)
or we can use catch

promise. catch (alert)

action inside promise	result
resolve()	then() success handler runs
reject()	then() error handler runs

Promise chaining →

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 . . .
```

means resolve(2) & return

either you can also create a promise and then return

idea is to pass the result through the chain of .then handler

Flow of execution

1 → the initial promise resolves in 1 seconds (Assumption)

2 → The next .then() handler is then called, which returns a new promise (resolved with 2 values)

3 → The next .then() gets the result of previous one and this keeps on going

every call to .then() returns a new promise whose value is passed to the next one & so on. We can even create custom promises inside .then()

~~Attatching multiple handlers~~

* if this promise is done then do this then do this callback hell is resolved using promise chaining

e.g p1. then (value) ⇒ {
(some promise lets say timeout)

```
console.log(value)
let p2 = new promise ((resolve, reject) →
{ set Timeout (() ⇒
resolve ("promise 2") }, 2000)
})
return p2
}). then ((value) ⇒ {
console.log ("we are done")
})
```

* promise chaining is when you use .then() repeatedly one after the other, to execute a series of asynchronous tasks in a specific order

* catch can be used at last to catch the error.

Attatching multiple handlers

p. then (handler 1)
p. then (handler 2)
p. then (handler 3)

e.g let p1 = new promise ((resolve, reject) ⇒
{ set Timeout (() ⇒ {
resolve(1); }, 2000)
})

static → belong to class itself
method → directly used using class Name
no need to create obje it

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

p.then( () => {
        console.log('congrats')

})
```

Promise API

There are 6 static methods of promise class →

```
let promise_all = Promise.all([p1, p2, p3])

promise_all.then ((value) => {
        console.log(value)
})
```

1. Promise.all (promises) → waits for all promises to resolve and returns the array of their results
If any one fails, it becomes error & all other results are ignored

2. Promise.allSettled (promises) → waits for all promises to settle and returns their results as an array of objects with status and value.

3. Promise.race (promise) → waits for the first promise to settle and its result / error becomes the outcome

4. Promise.any(promises)
waits for the first promise to fulfill (and not rejected), and it result becomes the outcome
Throws aggregate error if all promises are rejected

5. Promise.resolve(value) → makes a resolved value promise with the given value

6. Promise.reject(error) → makes a rejected promise with the given error

```
// let p = Promise.resolve(6)
p = Promise.reject
        (new Error("Hey"))

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

Async / Await

special syntax to work with promises in javascript

a function can be made async →

```
asyn function hello() {
        return 7;
}
```

- An async function always returns a promise
- other values are wrapped in a promise automatically

e.g harry().then(alert);

So, async ensures that function returns a promise and wrap non promises in it

AWAIT KEYWORD →

another keyword called await that works only inside async functions

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

await keyword makes javascript wait until the promise settle and returns its value

more elegant way of getting promise result than promise then + its easier to read & write

→ makes async code look like its running step by step
→ solution of promise chaining in easy way

callback, promise, async-await → try to solve asynchronous function after other
e.g async → sync → async

e.g

```
asyn function fetchUserData() {
  try {
    let response = await fetch("https://
        api-example.com/users");
    let data = await response.json();
    console.log(data);
  } catch (error) {
      console.error("Error:", error);
  }
}
```

using promise chaining

```
fetch("https://api.example.com/users")
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error("Error:",
    error));
```

Error Handling →

execution happen even if there is error by using try & catch

e.g rahul not defined

```
try {
    ~~console.log(error)~~
    console.log(rahul)
}
catch(error) {
    console.log(error)
}
```

→ any variable maybe x

- try catch works synchronously

- if an exception happens in scheduled code, like in setTimeout, then try catch wont catch it:

```
try {
    setTimeout (function () {
        //error code
    }
```

→ script dies and catch wont work

→ function itself is executed later, when engine has already left the try.... catch construct

→ need to use try-catch inside setTimeout to catch it

The error object →

For all built in errors, the error object has 2 ~~prop~~ main properties

```
try {
    hey;
}
catch (err)
{
    console.log (err.name)
    console.log (err.message)
    console.log (error.stack)
}
```

Throwing custom error →

```
if (age >180) {
    throw new Error ("Invalid Age")
```

we can also throw a particular error by using the built in constructor for standard errors.

```
let err = new SyntaxError (message)
    or
    new ReferenceError (message)
```

The Finally clause →

if it exist it runs in all cases.

If there is a return in try, finally is executed just before the control returns to the outer code.

whenever an error is encountered the control goes to catch (error)

// If there is error in catch then also finally runs

// close the file
// Exit the loop
// write to the log file

Fetch API → make asynchronous HTTP requests (GET, POST, PUT, DELETE)

```
let p = fetch("api")
p.then((val) => {
    return value.json
}).then((val) => {
    ~~return val.json~~
    console.log(val)
})
```

returns a promise → it gives a response object

→ Javascript can be used to send and retrieve information from the network when needed (AJAX)

asynchronous Javascript & XML

outdated using XML

now also used to refer fetch API

~~Fetch API~~ →
Fetch API →
it is used to get data over the network

```
let promise = Fetch(url, [options])
```

without options a get request is sent

JSON → Javascript object notation

Getting a response is a 2 stage process →

1 An object of response class containing "status" & "ok" properties

status → The http status code, e.g 200

ok → boolean, true if http status code is 200-299

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()

website

PAGE No.
DATE

Note → we can use only one body reading method, if we have already got the response with response.text() then response.json() wont work

- Response headers →
Response headers are available in response.headers

- Request headers →
To set a request header in fetch, we can use the headers option

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

POST REQUEST
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": "harry"}'
                    });
```

JSON placeholder

JSON.parse() → convert JS obj string to obj

JSON.stringify → convert Javascript object to string

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

Javascript cookies

cookies are small strings of data directly stored in the browser

In Javascript document.cookie provides access to cookies

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, the browser sends the cookie using the cookie HTTP- header. That way the server knows who sent the request.

We can also access cookies using document.cookie property:

```
alert(document.cookie)
```

contains key=value pairs separated by a ;

PAGE No.
DATE

writing to cookie →

document.cookie = "game = harry1123"

updates only cookie named game to harry1123

Encode URI component

let key = prompt(" ")
let value = prompt(" ")
document.cookie = `${key}=${value}`

now if key has a equal to symbol or a semicolon then value part will not be taken, so we use URI component

now, document.cookie =
`${encodeURIcomponent(key)} =
${encodeURIcomponent(value)}`

Now, decodeURIcomponent(" ")

put then encoded value not direct key

cookies is sent everytime there is a request

PAGE No.
DATE

cookie options →

cookies have several options which can be provided after key = value to a set call like this :

document.cookie = " user = bhz ;
path = /a ;
expires =
Tue, 29 March 2041 03:18:22 GMT"

path option makes cookie visible at /a, /a/b, etc,
expires set the cookie expiration time

Note →

1 → name = value pair, after URI encode URI component, should not exceed 4KB

2 → Total number of cookies per domain is limited to around 20+

LocalStorage →

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

inspect → application → localstorage

PAGE No.
DATE

localstorage.setitem("name", "harry")

Methods provided by 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- ~~key~~

6- length — number of stored items
property not function

→ we can get and set values like an object

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

→ Both key and value must be strings

→ we can use the two JSON methods to store objects in local storage

```
JSON.stringify(object)
JSON.parse(string)
```

PAGE No.
DATE

Session storage →

used less often than local storage. properties and methods are same as local storage but -

→ exist only within current browser tab.

→ data survives page refresh, but not closing/opening the tab

Storage event →

when the data gets updated in localstorage or session storage, storage event triggers with these properties:

1- key: The key
2- old value: → previous value
3- new value → new value
4- URL → page URL
5- storageArea → local or session storage

```
window.onstorage = (e) => {
    alert("changed")
    console.log(e);
}
```

triggers when data is updated

contains the above properties

make → programming counters app

height → 100% vh → increment height → local storage

height = 100vh

todo list
create/a
delete/a
access all todo's

label → label for an HTML form element usually an input
for attribute must match id of corresponding input

input → user can enter data

type :- text, email, password, checkbox, radio, submit

name :- used when sending form data to server

value :- initial value
placeholder

form :-

action :- url of server side script to handle form submission

method :- get or post

submit button submit form data to server

Radiobutton →
allows only one option from a group

```
<label>
    <input type="radio" name="gender" value="male">
</label>

<label>
<input type="radio" name="gender" value="female"> female
</label>
```

only one radio button in the same name="gender" group can be selected

CHECKBOX
allows multiple selections

```
<label>
    input type="checkbox"
        name="hobbies"
        value="reading"> Reading
</label>

<label>
<input type="checkbox"
        name="hobbies"
        value="traveling">
    Traveling
</label>
```

→ local storage value is string
JSON.parse() → to convert it into an array

- Toggle switch
- File Upload Button → opens file selector dialog
- color Picker Button → opens ~~clicker button~~ color picker
- Range Slider

selecting a number within a range using slider

label e.g

```
<label for="task">Enter Task
</label>
<input id="task" type="text"
    placeholder="Enter Task">
```

Object oriented Programming

[[Prototype]]

- Javascript objects have a special property called prototype that is either null or references another object

[prototype object]
↑ [[prototype]]
[object]

when we try to read a property from an ~~prototype~~ object and its missing, javascript automatically takes it from prototype. This is called "prototypal inheritance"

setting prototype —

we can set prototype by setting __proto__

if we have a method in object, it will be called from the object. if its missing in object and present in prototype, its called from the prototype

prototypes chain can also occur. if not find in 1st object then goes to 2nd object then 3rd
⋮

classes & Objects

In object oriented programming, a class is an extensible program code for template for creating objects, providing initial values for the state (member variables) and implementation of behaviour (member functions)

Basic syntax for writing a class is

```
constructor() { ... }
method1() { ... }
```

```
    method2 () { --- }
}
```

we can then use ~~any~~ new MyClass() to create a new object with all the listed methods

Example -

```
class RailwayForm {
    submit () {
        alert (this.name + " your form has been submitted for train number" + this.num)
    }
    cancel () {
        alert( this.name + "your form has been cancelled for train number" + this.num)
    }
    fill(name, num) {
        this.name = name
        this.num = num }
}

let myform = new Railwayform()
myform.fill('manjeet', 12789 3)
myform.submit()
myform.cancel()
```

constructor method -

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

```
class RailwayForm {
    constructor (name, num) {
        this.name = name
        this.num = num
    }
}
```

class Inheritance →

way for one class to extend another class

Extends Keyword →

```
class Monkey extends Animal {
    eat Banana() { console.log (this.name +
    }
}
```

```
class Monkey extends Animal {
    eat Banana () {
        console.log (this.name + " is eating banana") }}
```

PAGE No.
DATE

Method Overriding →

if we create our own implementation of sin, it will not be taken from the animal class, This is called method overriding

Super Keyword -

```
class Employee {
    requestleaves(leave) {
      console.log(' ${leave} granted');
    }
}

class programmer extends Employee
{
    requestleaves (leave) {
      super. ~~leaves~~ Requestleaves(leave)
      console. Loge (one extra
                    granted leave);
}}
```

- if we don't write constructor in extended class automatically the constructor of parent is called
- and if we create then using super constructor to set value in parent class is necessary
- use super before this in child constructor (no parameter.) else it will give error

PAGE No.
DATE

static Methods →

- implement functions that belongs to a class as a whole and not to any particular object
- not available for individual objects

```
class Employee {
    static sMethod () {
        alert ('Hey');
    }
}

Employee.sMethod ();
```

Getter & setter →

classes may include getter & setter to get & set the computed properties

example →

```
class person {
  constructor (name) {
    this._name = name }
```

PAGE No.
DATE

```
get name ( ){
    return this._name; }

set name (newName
    return this._name

set name (newName){

    this._name = newName}
```

first the name property is changed to _name to avoid the name collision with the getter & setter. Then the getter uses the get keyword as shown ~~alone~~ above.

```
let a = new person ("Jack")

a.name = "Bruno"
console.log (a.name);
```

Instance of operator →

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

Syntax is
<obj> instanceof <class>

onclick vs addeventlistener
only one handler — multiple handler
modern

PAGE No.

it returns true if obj belongs to class or any other class inheriting from it
i.e any child class

if we define get & set, we can directly change properties & access prop of an object

```
console.log
e.g (a.age)

a.age = 16;
```

CSS Knowledge →
start with biggest to smallest
* → body → divs

e.g
```
* {
    margin: 0;
    padding: 0;
    box-sizing: border box;
}
```
↳ height includes content + padding + border + margin

```
body {
    flex-direction: column;
    align-items: centre
    justify-content: flex start
    min-height: 100vh
```

transition

PAGE No.
DATE

```
    padding: 20px;
}

.drip
{

    align-items: center
    justify-content: center
    gap: 15px
    padding: 15px 25px; //
    border-radius: 12px; //
    border-shadow: 0 4px 10px rgba
}

label
{
    color: same as box of label


}
```

child

```
#task {
    padding: 10px 15px
    border-radius:
    width: 220px
    transition: border-color 0.3s
                transition
                of
                property
}
```

transition margin padding

```
#task:focus {
    border-color: blue
}

button {
    padding: 10px 18px
    font-size: 14px
    border-radius: 8px;
    background: blue
    color: white;
    cursor: pointer; // cursor
                        changes

    transition: background 0.3s,
                transform 0.2s;
}

button:hover {
    background: more blue
    transform: translate Y(-2px);
}

button:active {
    transform: translateY(1px)
}
```

margin padding display bg color gap padding
border radius box-shadow max-width
width

```
.tasklist {
    width: 100%
    max width: 500px;
}

ul li {
    transition: box shadow 0.3s

}

ul li: hover {
    box-shadow: 0 3px 8px []

}
```

// to remember

- go from outer to inside — checking hand it needs
- border radius
- flex → flex direction
  - → align items — center
  - → justify content — space around, space between, flex start
  - → gap
- padding
- transition
- max width
- margin-top
- interactivity → by transition
  - box shadow
  - background
  - transform
  - → cursor: pointer

Transition →

it smoothly interpolates one CSS value to another over time when that property changes

properties

property	duration	timing function	delay
which CSS property	how long animation runs	speed curve	wait before transition

slow start fast middle slow end — ease

linear — constant speed

ease in

cubic-bezier

custom curve

- properties that are good to transition →
- transform (translate / scale / rotate)

opacity

color

background-color

border color

box-shadow (expensive)

avoid transitioning layout properties that cause reflow — width, height, margin, padding

How to trigger a transition →

- triggered when → :hover : focus, adding/removing class with JS

Application - Local storage

```
elem.addEventListener('transitionend',
(e) => {
    if (e.propertyName == 'opacity') {

    }

});
```

@ Media in CSS
apply styles only when certain conditions are true (screen size, orientation, or device type)

```
@ media (max-width: 600px) {
    body {
        font-size: 14px;
    }
}
```

if screen width is 600px or less, text shrinks

common media features
- max-width
- min-width
- orientation: portrait/landscape
- (light/dark mode) prefers-color-scheme: dark
- (motion sensitive users) prefers-reduced-motion: reduce
- hover: none → touch devices without hover support

Password Generator

Advanced Javascript

IIFE → immediately invoked function expressions
it is a javascript function that runs as soon as it is defined

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

avoid polluting the global namespace, execute an async await, etc

- Before ES6 this was used to emulate block scope

Destructuring →
destructuring assignment is used to unpack values from an array, or properties from objects, into distinct variables

let [x, y] = [7, 20]

MDN - web docs
↳ original documentation

→ let [a, b] = [3, 5, 8]
a = 3, b = 5

→ let [a, b, c] = [3, 5] → throws an error

→ let [a, b, c, d, ... rest] = [3, 5, 8, 9, 12, 14]
a = 3
b = 5
c = 8
d = 9
rest = [12, 14]

→ let [a, , , ... rest] = [3, 5, 8, 9, 12, 14]
a = 3 rest = [9, 12, 14]

→ let {a, b} = {a: 1, b: 5}

---

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 add the key-value pairs to the object being created.

e.g

① let arr1 = [3, 5, 8]
let obj1 = { ... arr1 }
{0:3, 1:5, 2:8}
console.log(obj1) // ~~{0:3, 1:5, 2:8}~~

② function sum(v1, v2, v3) {
return v1 + v2 + v3
}
console.log(sum(... arr1))
→ 16

③ let obj2 = {
name: "Harry"
company: "Company xyz",
address: "XYZ"
}

console.log { ... obj2, name: "John"}

---

function ~~local~~, global & block scopes

Javascript has three types of scopes:

1. Block scope ⎫ local scope
2. function scope ⎭
3. Global scope

- variable declared within a javascript function, become local to the function

- A variable declared outside a function, becomes global

Hoisting →
refers to the process whereby the interpreter appears to move the declarations to the top of the code before execution

variables can thus be referenced before they are declared in Javascript

```
greet()
    function greet() {
        conside.log("Hello")
    }
```

Javascript only hoist declarations not initializations. The variable will be undefined until the line where its initialized is reached

Hoisting with let & var

with let & var hoisting is different

```
console.log(num)
let num = 6;
```

→ reference error if let or const

→ var undefined

function expression & class expression are not hoisted

```
fun() → throws error
const fun = () => {
    console.log("ran");
}
```

↓ same for let & var

```
const rectangle = class {

}
```

always declare all variables at the beginning of every scope

closures in javascript →

closure is created when a function 'remembers' that variable from the scope in which it was created, even after that scope has finished executing

closure gives you access to an outer functions variable from an inner function, even after the outer function is done running

```
function outer() {
    let count = 0;
    function inner() {
        count++;
        console.log(count);
    }
    return inner;
}

let counter = outer();
counter();   1
counter();   2
counter();   3
```

instead of destroying count, javascript keeps it alive

why closures are useful?

1. Data privacy/encapsulation
2. create specialized functions

```
function multiplier(factor) {
    return function(x) {
        return x * factor;
    };
}

let double = multiplier(2);
let triple = multiplier(3);
```

Arrow Functions

→ `const sayHello = name => console.log("hello" + name)`

→ when using two arguments then parenthesis are required

```
const x = {
    name: "Harry",
    role: "Js developer",
    show: function() {
        console.log("The name is ${this.name}")
    }
}
```

refers to this object

→ inside function this becomes global ~~window~~ object which is window ~~and event~~

→ in dom event handler, this refers to the element that received the event

→ arrow function in event handler → this = outer scope, not the element

---

REPL → read, evaluate, play loop
different from replit
→ on terminal when write node NODE REPL runs

Ryan Dahl → NODE

- Node filename → run on vs code
- npm init → initialize npm file without all details
- npm init -y → faster way without asking questions

→ e.g. module
- npm i pdf-node → install all dependencies in ~~package.~~ node_modules

package.json

```
"dependencies" : {
    "pdf-node" : "1.0.0"
}
```

→ you don't want to send node_modules to anybody because it is very heavy and can be ~~run~~ installed in another computer using
→ npm install
↳ install all dependencies

---

yarn same as npm
→ yarn add

another package manager

Modules in JS →

module1.js →

```
const hello = () => {
    console.log("Hello Harry")
}
```

lets take another function ahello

module.exports {hello, ahello}

type in package.json can be commanjs or module, default is commanJS

use in another file → 96.js

→ const hello = require("./module1")
~~→ console.log(hello())~~
→ hello.hello()
→ hello.ahello("Shivani")
→ hello.ahello("Garima")

alternatively
→ const { hello, ahello } = require("./module1")
→ hello()
→ ahello("Shivani")

ES6 modules; package.json → type:"module"

ex module2.js →

```
export const hello = () => {
        console.log("hello")
}

export const ahello = (name) => {
        console.log("Hello"+name)
}
```

96.js

```
import { hello, ahello } from "./module2.js";
```

again make todolist on your own after completing

if addition in module2.js

```
export default harry = () => {
        console.log("Hello"+Harry")
}
```

addition in 96.js

```
import harry, { hello, ahello } from "./module2.js";
```

---

export default function →

- default is used when you want to export one main value from a file
- value can be function, class, object or variable

```
export default function add(a,b){
        return a+b;
}

import sum from "./math.js";
```

export one main function or value from a module, which can be imported with any name

~~http://localhost~~
http://localhost:${port} → given → localhost

ctrl+C to exit server

configure tail and with express

Express.js →
- npm i express
- npm install express
- copied hello world from express website to vs code to run

```
app.get('/', (req, res) => {
    console.log(req.query)
    res.send('Hello World!')
})
```

(req → request object, res → response object)

NODEMON → global package

npm i -g nodemon → whenever save the file server restarts automatically

now you can use
nodemon file.js
instead of
node file.js

serving HTML file using express

```
const path = require('path')
app.get('/', (req, res) => {
    res.sendfile(path.join(__dirname, './index.html'))
```

(path → built in module in node.js; from website)

Regular expressions →

```
const regex = /very/g
const text = "Harry is a very very nice awesome nice very boy"
console.log(text.replace("very", "Very"))
```

change only one occurrence

replace with regex for all occurrences of very

character classes

1. . → any character except newline
   /. away/g
2. \w \d \S → word, digit, whitespace
   ~~/\d away~~ /\d away/g
3. \W \D \S → not word, digit, white space

^ → negated set

/i in place of /g
case insensitive

/gi

④ [a x g] → any of a, b or c

/[h l g c] array /g

⑤ [^abc] → not a, b or c

⑥ [a-g] → character between a and g

Anchors →

① ^abc$ → start/end of string

/^abc/g

↳ match abc hello

Escaped characters →

\0 \' \" \t \n \r →

escaped special characters

Groups

(abc) capture group → use operator on this group

---

quantifiers & alternation → repeat

- a* a+ a? → 0 or more, 1 or more, 0 or 1
- a{5} → exactly 5 times
- a|cd

Event loop →

what is javascript →

① it is a single threaded
↳ one main thread called call stack
↳ only one thing at a time in that thread

② non blocking - file I/O, API calls
- avoid waiting for slow task
- instead of freezing the thread it deligates task to environment
- keep running without waiting for slow tasks

③ asynchronous →
some operations will complete later

④ concurrency → manage multiple tasks at same time, even though it is

asynchronous → start now but finish later

synchronous → task run one after the other

single threaded.

- it does using event loop + callback queue + microtask queue

- it appears it is concurrent because it switches between tasks efficiently

- V8 → calls stack
  - heap

runtime of javascript

- Heap → memory allocation
- call stack → execution context

say hi()
greeting()

Asyncheronous callbacks →

- sometimes Javascript can take a lot of time and this can block page rendering
- Javascript has asynchronous callbacks for non blocking behaviour

phato & upload on Github today complete javascript

- Browser → runtime + web
  JS APIs

~~Node.js~~

Task queue → javascript can do only one thing at a time

→ not are queued to the task queue waiting to be executed

→ when we run setTimeout, webapis will run a timer and push the function provided to setTimeout to task queue once timer ends

→ task will be pushed to stack where they can be executed

Event Loop →

- runtime model based on eventloop
- responsible for ① executing code
  ② collecting & processing events
  ③ executing queued sub tasks
- pushes tasks from task queue to call stack

Heap →

shift + tab
vs-code outdent

Web Api + call stack + Task Queue
/runtime

Macrotask queue

set timeout callback

then

Macrotask queue, queue microtask

Single iteration looks like →

1 Event loop picks next macrotask from macrotask queue and pushes it to call stack

2 execute that macrostack until callstack is empty

3 drain the microstack queue completely - run all microtasks

4 optionally perform rendering (reflow/paint)

5 → repeat

Rendering → update screen after microtask drains before next macrotask

---

HTML AudioElement Interface

- if you don't want to mess with HTML → use it JS

→ var audio = new Audio (' ');
→ audio.play();
→ audio.pause
→ audio.currentTime = 0;

displays time & play sound at a specified time

use stack overflow

Project → Pdf Merger WebApp
→ Bulk Rename Desktop App

→ const path = require ('path')
↳ built in module

→ app.get . . . (
{
res.sendFile (path.join (__dirname, " templates/index.html"))
})

Bootstrap → 4 version
→ get boilerplate from website
navbar forms

< input class = " btn btn-primary my-4 "
type = "submit" >

base class
primary
margin vertical axis
spacing scale (0-5)

form of file

< form method = " post " action = "/merge"
enctype = " multipart/form-data " >

used when uploading files to server
encoding type
→ split form data in multiple

app-get → route handler for HTTP requests
(path, callback) → function executed when route is matched
URL endpoint

parts
Multer → middleware for handling multipart / form-data
→ npm i multer
- PDF MERGER JS
- EXPRESS static file serve

Bulk Rename Desktop in Node.JS →
- Reading files Node.JS → Node.JS website
  get name from this &
  then use rename to replace
  by traversing

.sh → shell script on macOS
.bat → series of commands that cmd can execute one after another

let a = new Date().getTime()
~~a.getTime()~~ ↳ unique no. is generated

PHOTOS upload mindmap create