**Javascript**

- JS is a client-side scripting language used to create dynamic web-applications and websites

- It runs in the browser(client-side) and also in our local environment(server-side) with NodeJs

- It adds the functionalities/actions to web-pages and makes them interactive Or it reacts to user-events like click event , request-response from server event , mouse-movement event etc .

- Since it was specificly developed for browser , it doesn’t have low-level access of memory or cpu and hence Javascript is safer though.

- JS gets executed through Browser

**difference b/w Dynamic-website(web-app) and Static-website**

|  |  |
| --- | --- |
| Static site | Dynamic site |
| No request response scenario | It generates data /pages in real-time , as per the request to server . |
| Pages/Data will remain same until someone changes its manually | Content of pages are different for different visitors |
| Less compelexity | High complexity |
| Information changes rarely | Information changes frequently |
| less time for loading | more time for loading |
| Data-base is not used | database is used |

**AJAX(Asynchronous Javascipt and XML)**

- Reads data from a web-server after a web page has loaded

- Update a web-page dynamically without reloading the page

- Send data to a web-server in the background

- It can send and receive informations in various formats like JSON , XML , HTML and text files

**DOM(Document Object Model)**

- When a web-page is loaded, the browser creates DOM of the page . Hence all the HTML elements like , div , nav , body etc comes under DOM

- With HTML DOM , Javascript can access and change all the elements of an HTML document

- The HTML DOM model is constructed as a tree of Objects

HTML DOM Tree of Objects



**Cookies**

- It is a piece of data consisting user’s information(that is stored by the browser on user’s system(l**ocal storage**)) by which the server will recognizes the user when the user sends a request to the server next-time .

- Cookies are saved in name-value pairs . Ex: username = John Koay

**Local-storage**

- It is a property that allows JS sites and apps to save key-value pairs in a web browser with no expiration date , it means that the data stored in the browser will persist even after the browser window is closed .

**Same-origin-policy** - It is a critical security mechanism that restricts how a document or script loaded by one origin can interact with a resource from another origin . like , one website owner can’t make an ajax-request to get it’s user’s facebook details

**What can’t IN-BROWSER JS do?**

- Read/write to and from the system’s hardisk

- Same-origin-policy applied

**What makes JS unique ?**

- HTML/CSS support

- Simple APIs for simple things

- Major modern-browsers support(enabled by-default)

**ECMAScript(European Computer Manufacturers Association Script) - It is a standard for scripting languages like , Javascript ,Jscript , ECMAscript(Google it)**

**Variables , Datatypes , Operators , if-else , switch-case , arrays , functions , loops**

**Variables**

**var a = 2**; (Integer) var b = “String” (String variable) var c =12.55 (float)

**-JS is a dynamically typed language , means the datatypes of variables are checked at runtime and we don’t need to pre-specify the datatypes .**

**let a = 6 ;**

- **var** is function-scoped and **let** is block-scoped

**const = “constant” ;**

- const variables are **block-scoped** and can’t **reinitializable(updated) , redeclarable**

|  |  |  |
| --- | --- | --- |
| **var** | **let** | **const** |
| functional scope | block scope | block scope |
| can be updated and redeclarable in the cope | can be updated but can’t be redeclarable into the scope | It can’t be updated and redeclarable intot the scope |
| It can be declared without initialization | It can be declared without initialization | It can’t be declared without initialization |
| It can be accessed without initialization as its default value is “undefined” | It can’t be accessed without initialization as it returns error | It can’t be accessed without initialization as it can’t be declared without initialization |

**Types of values in Javascript**

|  |  |
| --- | --- |
| **Primitive values** | **Reference values(Not Reference variable)** |
| These are not any objects and has no methods | These are objects that consists of multiple values and methods |
| STRING , NUMBER , BOOLEAN ,SYMBOL , UNDEFINED , NULL , | ARRAY , OBJECT , FUNCTION |
| Stored in stack | Stored on Heap |
| Fixed size | Not fixed size |
|  |  |

- When you declare a variable , the JS-engine allocates the memory for them on two memory locations either **stack or heap**

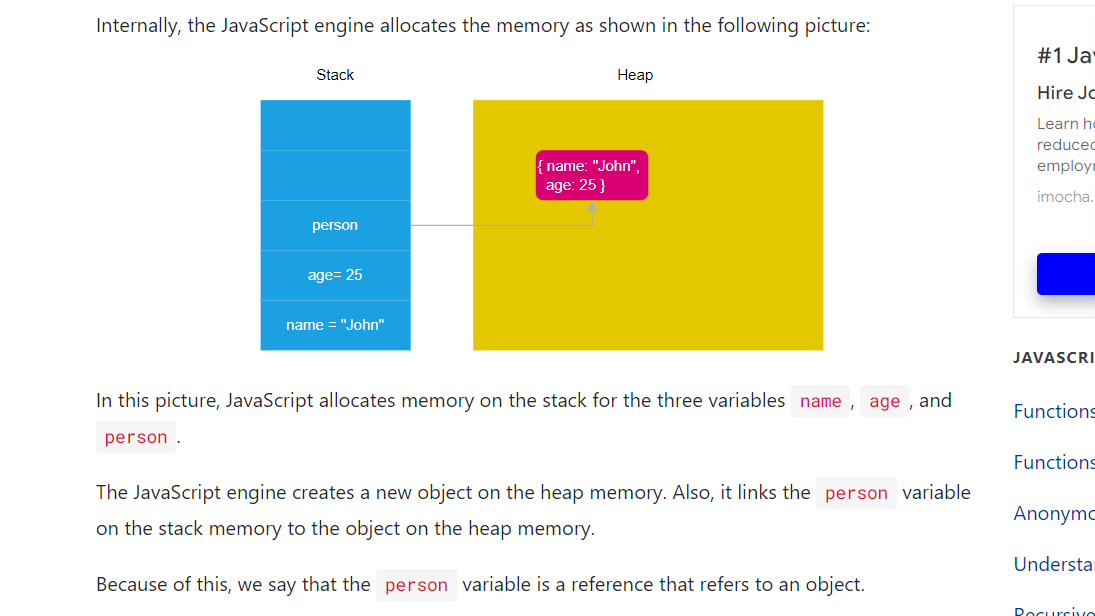
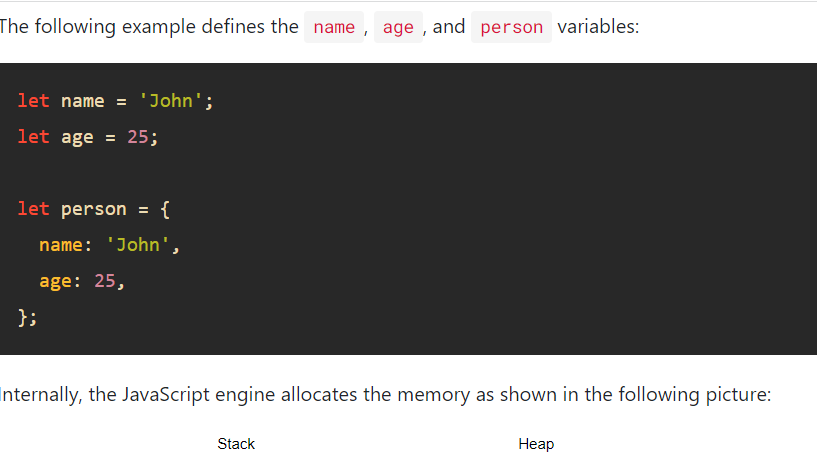
- **Static data** : It has fixed size at compile time . These are ,

1 . **Primitive values** and their variables. like name and age in below diagram and their values

2.**Reference Variable** which refers to object. like person in below diagram

- Since the size of static data doesn’t change , JS-engine store these data in Stack

- Since , the JS-engine doesn’t allocate a fixed amount of memory for object values instead it will allocate more space as needed , hence JS-engine stores them on Heap



**Operators**

Unary operator - Sinle operand . ex:- -- , ++ etc

Binary Operator - Two operands . ex:- Arithmetic{+,-,\*,/,%(MOD),\*\*(exponentiation)} , logical(&& , || , !) , Relational(> , < , == , !=) , assignment(= , += , \*= , -= , /=)

**Strings**

var str1 = “hel’lo”; var str2 = ‘Hel”lo’; var name = “Rahul”; var str3 = “person”;

name[2] = h

temp = **`${name}** is a good **${str3}` ;** <-This way you can include variables inside the string

String by string-constructor : **var y = new String(“person”);**

name.**length** -> Gives the length of name variable

var str = “This is a string”;

var position = **str.indexOf(‘is’)** = 2 -> It gives the index of first occurrence of substring

position = str.**lastIndexOf(‘is’)** = 5 ->index of last occurrence of substring

substr = str.**slice(1,7)** = his is -> slices the string and gives from 1 to 7-1 characters as substring

- Function substring(1,7) gives the same output as slice(1,7) , but slice() can take negative values unlike substring()

substr1 = str.**substr(1,2)** = hi -> It exatracts the substring from index 1 to 2

var replaced = **str.replace(‘string’ , ‘Rahul’)** -> It simply replace ‘string’ to ‘Rahul’ and the new string is assigned to the new variable without changing the previous string

**- toUpperCase()** and **toLowerCase()** methods make string UPPER and lower case respectively

- **concat(‘new string’)** concatenates the parameter to the string

- **trim()** trims the extra spaces of string from beginning and from end

- **charAt(n)** gives the n-index character of string

- **charCodeAt(n)** gives the character-code of n-index character

**if-else(conditional statements)**

*SAME AS C/CPP/JAVA*

**Switch-case**

*SAME AS C/CPP/JAVA*

**Objects & Arrays**

**Object**

- In JS almost everything is object except primitives(Primitives can also be objects if defined by **new** keyword) like , arrays , functions etc

- Object is a collection of key-value pair .

Ex:- **let employee = { name:”John” , age:20 , profile:”Data Analyst” , eyes:”Blue” , “name 2”:”Roshan”};**

Access the key-value pair :

**employee.name = employee[‘name’]** = **“**John”

**employee[‘name 2’]** = “Roshan”

**Arrays**

- In JS arrays is an ordered list of values . **It can hold values of different data-types .**

Ex:- let names = [1,2,”Roy” , undefined , 0 , 1]; OR let names = **new Array**(1,2,”Roy” , undefined , 0 , 1);

**names.length** : Gives the number of elements in array(length)

**names.sort()** : Sorts the array

**names.push(element)** : Pushes the element at last

let names = new **Array(23 , 45)** <- creates an array and putting 23 and 45 as elements

let names = new **Array(23)** <- Creates an array of 23 empty elements

array.**push(element) -** this method is used to insert any array element into the array

array.**indexof(element) –** it gives the index of the given element in an array

array.**splice(index , number of elements)** – it takes off the given index-element from an array

array.**length** – returns the array length .

**Functions**

**- *function*** keyword is used to define a function and one don’t need to give parameters’ data-types .

Important keywords/Functions:-

- One can define functions using **function-declaration** or **function-expression**

Declaration:-

*function demo(arg1 , arg2 = “default value of arg2”)*

*{ console.log(arg1 + arg2); }*

Expression:- A function expression can be stored in variables

let x = function(a , b) {return a\*b ; }

The variable ‘x’ can be used as function

**Loops**

**for , while and do-while**

*ARE SAME AS C/CPP*

**forEach()** loop

let friends = [“Rohan” , “Sanjiv “ , ”Rekha” , “Pooja”];

**friends.forEach**( function f(**element**) { console.log(“Hello ” + **element**)});

Output : Hello Rohan , Hello Sanjiv , Hello Rekha....

**for-of** loop

**for(element of friends)**{ console.log((“Hello ” + **element**)) };

Output : Hello Rohan , Hello Sanjiv , Hello Rekha....

- **forEach** and **for-of** loops are used to **iterate arrays** , and **for-in** loop is used to **iterate objects**

**for-in** loop

let employee = {name:"Ram" , salary:2 , age:34};

**for(key in employee)**{ console.log(`The **${key}** is **${employee[key]}**`) }

output : The name is Ram

The salary is 2

The age is 34

**modal dialog box** - Until this dialog box is closed you can’t access any part of the page

**alert(“message”)** - It’s a method Used to display a message to the user . It’s return values is undefined

let name = **prompt(“What is your name” , default string)**; - Displays a dialog-box and will take users input and return it to store in any variable

**confirm(“String”)** - displays a dialog box with OK and Cancel , it returns Boolean values , like true if OK pressed and false if cancel pressed . It is used to double-check the user’s action ,i.e if the user really want that section or he mistakenly did the action.

**document** – It is the keyword allows to get access to HTML page .

**getElementById()** – It allows us to grab any element of HTML by ID .

**console.log(token)** – print items

var x = **window.prompt(“Enter = ”) –** it will receive an user-input and will assign it to the variable x

**DOM Navigation**

- With object model , JS gets all the power it needs to create Dynamic HTML(more interactive html)

<div id="main" class="container">

<ul id="lists"class="nav">

<li>Home</li>

<li>About</li>

<li>help</li>

<li>About</li>

</ul>

</div>

<div class=”container”>Hello</div>

<script>

- let main = document.**getElementById('main')**;

console.log(main);

- let list = document.getElementById('lists');

console.log(list);

</script>

- lists.**innerHTML** - It will give the HTML code under lists id as a **string .**

**i.e**  “ <li>Home</li>

<li>About</li>

<li>help</li>

<li>About</li> ”

- let **containers** = **document.getElementsByClassName(‘container’)** - This will return the element with given class-name or the collection of elements which are having the same given class-name .

**containers[0]** <div class=”**container**”>Hello</div>

**containers[1]** <div class=”**container**”>Boyeii</div>

- let **sel** = document**.querySelector(‘.container’)**

**-** Clearly , this method is used to grab elements by any **CSS - element selectors** .

**-** This will return the **first element** having ‘container’ class and will assign the the variable(sel) , so that we can modify or select the returned elements.

- While **querySelectorAll()** returns all elements . After using this , *sel[0] = Home , sel[1] = About* and so on.

**More on OBJECTS**

**objects –** it is the collection of key-value pair

**var student = {firstName: ”Rahul” , lastName:”Kumar” , age : 12};**

**var student = {} ; <- empty object**

**var x = new Object();** - we can create an empty object by this only . Afterwards , we can add key-

value pair to that empty object

After creating object , we can add key-value to it by following way and same way will be applicable for an empty object.

***x.keyOne = value1;***

***x.keyTwo = value2;***

*- We can create javascript objects manually by functions(constructors) also .*

***function Student(first,last,age)***

***{***

***//down below , we are creating keys/properties explicitly and we are adding their values by function-parameters***

***this.firstName = first;***

***this.lastName = last ;***

***this.Age = age ;***

***this.greeting = function(){***

***return "Hello everyone , this is " + this.firstName +" " + this.lastName + " and I am " + this.Age + " years old . " ;***

***}***

***}***

***var studentOne = new Student(“Rahul”,”Kumar”,85); <- This is the object of Student()***

**-** ***In javascript , we create the objects of a function too like in classes***

variable**.value –** value function modifies or set the value of the variable(inputs)

variable**.innerText** – it works as same as **.value** but for variables holding non-input elements .

- target.**addEventListener(event-type , listener)** – this method allows you to set up a function to be called when a specified event happens , such as when user clicks a button .

- **parseFloat()** method turn a string into float

- **console.log()** and **alert()** methods turns anything into strings automatically

- Default behavior of form is that it **refreshes** itself on every submission and wipe all your data off . To prevent this we off this default feature , using **preventDefault()** method .

event.**preventDefault()** – it cancels the event if it is canceleable , or the default action that belongs to the event will not occur .

**Events , Event-Handlers & Event-Listeners**

**Events**

- HTML events are “things” that happen to HTML elements OR these can be something the browser does or something a user does

ex: - button clicked , webpage finished loading etc.

- When JS is used in HTML , JS can react on these events OR JS lets you execute code when event are detected

**Event Handlers**

- These are the attributes of the HTML elements which manages how the element should react when an event fires

- Event handlers holds the function or JS-code which is to be executed on an event

|  |  |  |
| --- | --- | --- |
| **Events** | **Event-Handler/Attribute** | **Discription** |
| click | onclick | when Mouse click on an element |
| mouseover | onmouseover | cursor of the mouse comes over the element |
| mouseout | onmouseout | cursor of mouse leaves the element |
| keydown & keyup | onkeydown & onkeyup | When user press & release the key |
| focus | onfocus | when the user focuses on an element |
| submit | onsubmit | user submits the form |
| load | onload | when browser finishes the loading of the page |

**Event Listener**

- **addEventListener()** method is one of the event-listeners

- It attaches an event handler(like , click , mousedown , mouseover) to the specified element without overwriting the existing event

Syntax : ***element.addEventListener(event , function , useCapture(optional));***

- **removeEventListener()** method is used to remove an event-listener on element

**setTimeout() , clearTimeout() , setInterval() , clearInterval()**

setTimeout()

- Executes the code/function after provided milliseconds

***let variable = setTimeout(function , time(in miliseconds) , function\_arguments);***

- setTimeout function returns an unique **timeout-id** which gets stored in the variable

- **clearTimeout(timeout-id)** method clears the timer set by setTimeout() method by accessing the unique timeout-id

setInterval()

- Executes the code/function repeatedly after every given milliseconds .

- Its format is as same as of setTimeout() and it also returns unique **interval-id**

- **clearInterval(interval-id)** method is used to clear the intervals set by setInterval().

- function can’t be called under the parenthesis of setInterval() or setTimeout() , we can’t use paranthesis after function-name like: we can write setTimeout(**function**) instead setTimeout(**function()**)

ex: **display time at a page after every sec:-**

<html> The time is <span id="time"></span></html>

<script>

function **displayTime(){**

**let time = new Date();** <- Date object , gives current date and time

document.getElementById('time').innerHTML = time;

}

**setInterval(displayTime , 1000);**

</script>

output :  The time is Tue Jan 25 2022 15:05:51 GMT+0530 (India Standard Time) -> changes every second

**Date & Time**

- Date objects are created with **new Date()** constructor

let dt = new Date(milliseconds) ; <- By default milliseconds are accepted

more specificly,

let dt = **new Date(year , month , date , hours , minutes , seconds , milliseconds) ;**

**dt.getDate() : returns date only**

**dt.getFullYear() : returns year only**

**dt.getMonth() : returns month only**

**dt.getHours() : returns hour only** and so on......

**dt.setDate(15)** : sets date to 15

**dt.setDate(32)** : gets autocorrected by JS as **setDate(1)**

**dt.setHours(x)**

**dt.setMinutes(x)**

- For dates , 1 January 1970 5:30pm is taken as reference . hence if , new dt = new Date(0(ms)) , it assigns 1 January 1970 to dt and arguments under **Date(x)** are considered as milliseconds after 1 January 1970 5:30 pm

- **Date.now()** returns the number of milliseconds(or Time-stamps in ms) since 1 Jan 1970

**Arrow Functions**

- It allows us to write shorter function syntax

Before:-

**let fun1 = function(a,b) {return a\*b;}**

With arrow-function:-

let fun1 **= (a,b) => {return a\*b}** or **let fun1 = (a,b) => a\*b** also **let fun1 = a => a\*2**

Using arrow-function in setTimeout() ,

setTimeout( **function fun(){return a}** , 3000}) == setTimeout( **()=>{ return a }** , 300) == setTimeout( **() => a** , 300)

**lexical this**

**lexical-scoping :** nested-function of a parent-function have the access to the variables and other resources of their parent scope

-In an object , nested **arrow-functions** have **lexical-scoping** means these arrow-function can access the keys and values of main parent , i.e. of object . But , the general nested-function have not lexical-scoping .

scope of ‘this’ is to whole obj1 , thatswhy , the greeting key is being accessed here . Hence , here ‘**this’** means **‘obj1’**

Ex:- let **obj1** = {

**greeting** : “Good Morning”,

normal parent-function has by-default lexical scoping

**names** : [“Harry” , “Pija” , “Ravi” , “Tom” ],

speak(){

**this.names**.forEach( (student) => {console.log**(this**.**greeting** + student)});

}

If this nested-function had been a normal function**[function s(student){}**] then its scope wouldn’t be as same as arrow-function and in that case **‘this’** will give an undefined value

}

Here also , **‘this’** means **‘obj1’**

**Math object**

- This object allows you to perform mathematical tasks on numbers

**console.log(Math)** : It will give list of all the methods wrapped under Math-object , i.e ,

E: 2.718281828459045 = Math.E

LN2: 0.6931471805599453 = Math.LN2

LN10: 2.302585092994046 = Math.LN10

Math.round(3.555) = 4 , round the number

Math.pow(2 , 3) = 2 raised to the power 3

Math.sqrt(64) = 8 , gets square-root

Math.ceil(5.66) = 6 , It gives the rounded-up value

Math.floor(5.66) = 5 , It gives the rounded down value

Math.abs(-5.55) = 5.55 , It gives the absolute value

Math.min(4,5,6) = 4

Math.max(4,5,6) = 6

Math.random() = generates random number b/w 0 and 1

**Random number b/w a and b** = **a + (b-a)\*Math.random() ,** and for integer value use **Math.round()**

**JSON(Javascript Object Notation)**

- It is light-wight data intrerchange and data-storing text-format

- It is easy for humans to read and write .

- It is easy for machines to parse(breakdown and explaining each elements) and generate

- It is used for data-transportation between systems

- It is language-independent , means it can be used/parsed in any languages like , python , c# , ruby etc

- It is purely a string with a specified data format ; It contains only properties and no methods .

Ex = ‘ {“name” : “John” , “age”:30 , “car”:null} ’

- For transfer of data to web-servers it should be in string format , hence we need to stringify the js-object

- To stringify/convert an object to get it turn into a JSON string-format we use **JSON.stringify(JS\_obj)** or JSON.stringify() method takes js-object and returns a string

ex:- const obj = {name: "John", age: 30, city: "New York"};

const jsonStr = JSON.stringify(obj) ;

output: jsonStr = ‘ {“name”: "John", “age”: 30, “city”: "New York”} ‘ 🡨 JSON string-format

- To replace a string-value in JSON-string we use a string-method called **replace(‘target\_str\_name’ , ‘new\_str\_name’)** , that gives a new JSON-string with replaced string-value .

ex: If we want to replace “John” to “Mohan” we simply go like ,

jsonStr = jsonStr.replace(‘John’ , ‘Mohan’) ;

- In order to turn the JSON-string into a JS-object we use **JSON.parse(JSON\_string)** method or this method takes JSON-string and returns js-object

ex:- let jsonStr = '{"name":"John", "age":30, "city":"New York"}'

let jsonObj = JSON.parse(jsonStr) ;

output : jsonObj = {name:"John", age:30, city:"New York"} 🡨 JS-object