**JavaScript**

**Variable Declaration:**

var a;

a=20;

var b;

b=”Hello World”

**document.write(“<br>”); -** used to print anything on screen (br tag is used to break the line using javascript)

**alert(); -** used to pop any window on the screen

var a = "Hello, This is a new \"Youtuber\" "

var b = 100

document.write(a);

document.write(b);

document.write(typeof(a));

== (It is used to compare the value only)

=== (It is used to compare the value and datatype also)

var a = 10

var b = “10”

!==

switch(expression){

case constant: code;

break;

case constant: code;

break;

default: code;

}

while(condition){

code;

}

do{

code;

}

while(condition);

for(i=0;i<=n;i++)

{

Code;

}

function fun\_name(){

group of statements;

}

To call a function: fun\_name();

var a = sum(10, 20);

document.write(a);

function sum(b, c)

var add = b + c;

return add;

var A = [1,2,3];

var B = [‘A’,’B’,’C’];

NEW Keyword: used to initialize the array

var C = new Array(‘A’,’B’,’C’);

var array = [‘A’,’B’,’C’];

array.length(); used to count number of elements

document.write(length);

sort() method is used to sort elements in ascending order by default

var array = [‘A’,’B’,’C’];

array.sort();

document.write(array);

push() and pop() is used to add a new element at end of an array and delete last element of an array respectively.

shift() is used to delete first element of an array.

var array = [‘A’,’B’,’C’];

array.push(‘D’);

document.write(array);

array.pop();

array.shift();

array.unshift(‘D’) is used to add an element at first position of an array.

reverse() is used to reverse the sequence of an element of an array.

array.reverse();

concat() is used to join two or more arrays.

var array = [‘A’,’B’,’C’];

var array1 = [‘D’,’E’,’F’];

document.write(array.concat(array1));

indexOf() searches the array for the specified item, and returns it position.

Document.write(array.indexOf(‘A’));

JavaScript Dialog Boxes:

* **Prompt** (if you want the user to input a value before entering a page)

var name = prompt(“ What is your name? ”);

if(name!=””){

document.write(“Hello”+name);

}

* **Alert**
* **Confirmation** (It is used to take user’s permission on any option. This box is used only when someone clicks on button.)

var user = confirm(“Do you want to visit website?”);

if(user == true){

window.open(“www.google.com”);

return true;

}else{

document.write(“Donot want to visit”);

return false;

}

var firstname = “John and John” typeof(firstname):string

var lastname = “Andrew”

var firstname = new String(“John”); typeof(firstname):object

**document.write(firstname.length);**

**charAt():** returns character at a specified index in a string

document.write(firstname.charAt(8));

**indexOf():** return the index of first occurrence of specified text in a string

document.write(firstname.indexOf(“John”));

**lastIndexOf():** return the index of last occurrence of specified text in a string

document.write(firstname.lastIndexOf(“John”));

**replace():** replace a specified value with another value in a string

document.write(firstname.replace(“and”, “&”));

**toUpperCase():** document.write(firstname.toUpperCase());

**toLowerCase():** document.write(firstname.toLowerCase());

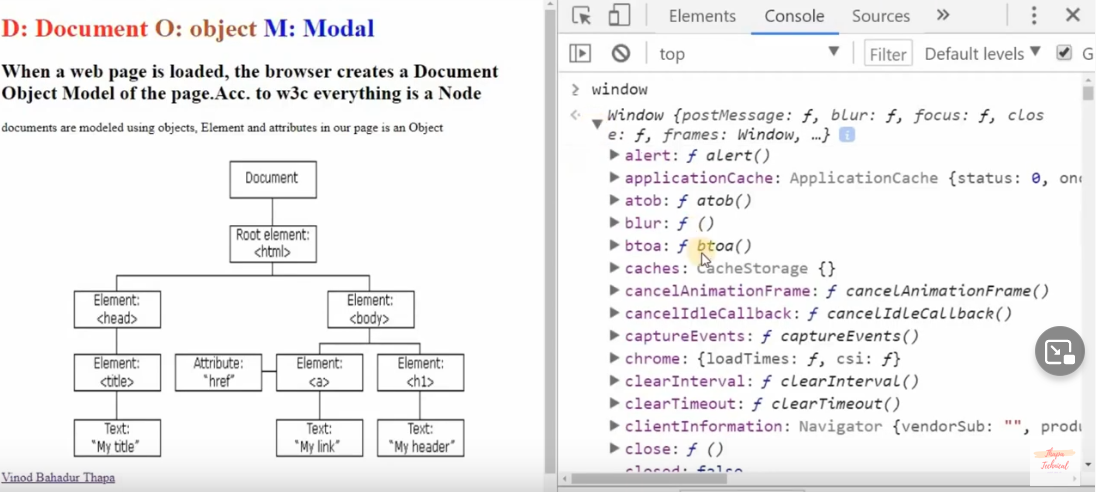
**concat():** document.write(firstname.concat(lastname));

when a page is loaded, the browser creates a document object model (DOM) of the page.

Go to console after right click on web page and click on inspect element. In console, write WINDOW.DOCUMENT

Complete web page is a document.

Under Document, there is **ALL** option, under which there are many objects. (attributes and elements of the page is termed as objects).



**Type ‘window’ on console: window object will open -> goto ‘document’ section -> goto ‘all’ section -> goto ‘html’ section -> goto ‘childNodes’ section**

Id and Class attribute: id value must be unique and class value can be same.

<p id="welcome">Welcome</p>

<p class="home">Home</p>

<p class="home">Home 1</p>

<script type = "text/javascript">

var a = document.getElementById("welcome").innerHTML; **//access elements by ID**

document.write(a);

var b = document.getElementsByClassName("home")[0].innerHTML; **//access elements by class**

document.write(b);

var c = document.getElementsByTagName("p”); **//access elements by tag p(paragraph tag)**

document.write(c);

**console.log(c);**

var d = document.querySelector("p”).innerHTML; **//access first element or first child(if you have multiple paragraph and you want to access first paragraph.**

document.write(d);

</script>

<p id="welcome">Welcome</p>

<p class="home">Home</p>

<p class="home">Home 1</p>

<ul><li>Like</li><li>Share</li><li>Subscribe</li></ul>

<script type = "text/javascript">

var child = document.body.children; **//access children of a body**

console.log(child);

var para = document.createElement(“p”); **//add a new child in a body (p represent for paragraph, you can take other element also like UL, A, IMG etc.)**

var text = document.createTextNode(“This is new paragraph”);

para.appendChild(text);

console.log(para);

document.body.appendChild(para);

var **list** = document.querySelector(“ul”);

console.log(list.firstChild); **//access first child of an element**

console.log(list.lastChild); **//access last child of an element**

var firstsibling = welcome.nextElementSibling; **//access next sibling of an element ID named as “welcome”**

var firstsibling = welcome.nextElementSibling.nextElementSibling; **//next to next sibling**

console.log(firstsibling);

console.log(**list**.parentNode); **//access parent node of an element name as “list” (UL)**

</script>

**SET CSS in JAVASCRIPT**

**One by one**

<h1 id = ”change”>Hello World</h1>

<h1 id = ”changecss”>Hello India</h1>

document.getElementById(id).style.property = new style **//SYNTAX**

document.getElementById(‘change’).style.color = “red”;

**All at once**

The setAttribute() method adds a new attribute

document.getElementById(‘changecss’).setAttribute(“style“, “color: red; background-color: black; text-align: center; height: 100px”);

**JAVASCRIPT WINDOW LOCATION**

The location object contain information about the current URL. The location object is part of the window (global object in console) object and is accessed through window.location property.

**Window Location Href**

window.location.href property returns the URL of the current page.

**Window Location Hostname**

window.location.hostname property returns the name of internet host.

**Window Location Pathname**

window.location.pathname property returns the pathname of the current page.

**Window Location Protocol**

window.location.protocol property returns the web protocol of the page.

**Window Location Assign**

window.location.assign() method loads a new document.

Example: **document.write(window.location.href);**

**window.location //it is used to redirect the page.**

window.location = “https://www.youtube.com/”; OR

**Example:**

**<button onClick = “myfun()”>Click</button>**

**<script type = “text/javascript”>**

**Function myfun(){**

**window.location.assign(“https://www.youtube.com/”);**

**}**

**</script>**

**PRINT PAGE (specific content) using JAVASCRIPT**

window.print() is used to print the current whole web page.





**EVENTS in JAVASCRIPT (Important)**

When the page loads, it is called an event. When the user clicks a button, that click too is an event. Other events include like pressing any key, closing a window, resizing a window etc.

**onclick**

<button onclick=”functionname()”>Click</button>

**onmouseover and onmouseout**

<button onmouseover=”functionname()”>Click</button>

<button onmouseout=”functionname()”>Click</button>

**onkeydown (when key is pressed), onkeypress (when key is presses and released), onkeyup (when key is released)**

<button onkeydown=”functionname()”>Click</button>

<button onkeypress=”functionname()”>Click</button>

<button onkeyup=”functionname()”>Click</button>

**DATE OBJECT in JAVASCRIPT (yyyy, mm, dd) - Jan 01, 1970**

var d = new Date();

document.write(d);

getFullYear() //get year as four digit number (yyyy)

getMonth() //get month as a number (0-11)

getDate() //get the day as a number (1-31)

var d = new Date(); **//how to get a date**

var month = d.getUTCMonth()+1;

var day = d.getUTCDate();

var year = d.getUTCFullYear();

UTC is Universal Coordinated Time which is set by World Time Standard.

var d = new Date(); **//how to set a date**

d.setMonth(11);

d.setDate(20);

d.setFullYear(2020, 11, 2);

document.write(d);

**TIME OBJECT in JAVASCRIPT**

var d = new Date();

var n = d.getTime(); **// return milliseconds since 1970/01/01**

var hour = d.getHours(); **//0-23**

var min = d.getMinutes(); **//0-59**

var sec = d.getSeconds();

document.write(hour+”:“+min+”:“+sec);

**DYNAMIC TIME in JAVASCRIPT**

setInterval() method will continue calling the function until clearInterval() is called or the window is closed. clearInterval() method stops the execution of the function specified in setInterval() method.

toLocaleTimeString() method returns the time portion of a Date object as a string, using locale conventions.

<button onclick = “**clearInterval(clock)**”>Click Me</button>

<p id = “**showclock**”></p>

var **clock** = setInterval(clocktime, 1000); **//1000 represent 1 sec, clocktime is callback function**

function clocktime(){

var d = new Date(); var t = d.toLocaleTimeString();

document.getElementById(‘**showclock**’).innerHTML = t;

}

**setTimeout(): It will popup some window after sometime. Popup will open according to the function and time will set after which the function performs.**

var **d** = setTimeout(myfun, 3000); **//3000 represent 3 sec**

function myfun(){

alert(“Hello”);

}

**EVAL() in JAVASCRIPT**

It evaluates or executes an argument. If the argument is an expression, eval() evaluate the expression. If the argument is one or more javascript statement, eval() executes the statements.

**MATH OBJECT in JAVASCRIPT**

Math.PI //returns the ratio of circle’s circumference to its diameter

Math.round(x) //returns the value of x rounded to its nearest value

Math.pow(x,y) //returns the value of x to the power y

Math.sqrt(x) //returns the square root of x

Math.abs(x) //returns the absolute positive value of x

Math.ceil(x) //returns the value of x rounded up to its nearest integer

Math.floor(x) // returns the value of x rounded down to its nearest integer

Math.min(x,y,z,….) and Math.max(x,y,z,….) //used to find lowest and highest value in list of arguments

Math.random() //returns random number between 0 (inclusive) and 1 (exclusive)

document.write(Math.pow(8,2));

**GENERATE RANDOM NUMBER (between 0 and 19)**document.write(Math.floor(Math.random() \* 20));

**ANIMATE DIV WHEN SCROLL TO ITS POSITION (SCROLL EFFECT)**

* **Bootstrap Container** (https://www.w3schools.com/bootstrap4/bootstrap\_get\_started.asp)
* **Two Basic Bootstrap 4 Pages** (<https://www.w3schools.com/bootstrap4/bootstrap_cards.asp>)
* **Animate CSS** (<https://cdnjs.cloudflare.com/ajax/libs/animate.css/4.1.1/animate.min.css>)
* **animate.css** (add animate with function\_name in a div class which you want e.g animated bounceInLeft)

$(document).ready(function(){

$(window).scroll(function(){

var positiontop = $(document).scrollTop();

if((positiontop > 10) && (positiontop < 213)){

$(‘#card-one’).addClass(‘animated bounceInLeft’);

$(‘#card-two’).addClass(‘animated zoomIn’);

}

});

});

Here #card-one is div id in which we have add class of animation

**SCOPE CHAIN in JAVASCRIPT**

Scope Chain is the locations where identifiers are declared that is searched to resolve the value of an identifier.

Execution Context refers to scope.

Lexical Scope means that in a nested group of functions, the inner function have access to the variables and other resources of their parent scope.

var a = 10;

console.log(a); //a will give 10

fi(){

var b = 20;

f2(){ //climb up the scope chain

console.log(a); //a will give 10

console.log(b); //b will give 20 because f2() is declared inside f1()

}

}

**JAVASCRIPT OBJECTS**

It is a collection of named values also called as properties.

Objects are variable containing variables.

Javascript variables can contain single values.

Objects are variable too but objects can contain many values.

The values are written as **name : value** pairs (name and value separated by colon)

var technical = {

f\_name : ”Aayush”,

l\_name : ”Agarwal”,

age : 20

}

console.log(technical);

console.log(technical.f\_name);

**HOW TO CHANGE PROPERTY VALUE**

technical.age = 25;

console.log(technical);

**HOW TO ADD A NEW PROPERTY WITH VALUE**

technical.m\_name = “R”;

console.log(technical);

**HOW TO DELETE PROPERTY NAME**

delete technical.m\_name;

console.log(technical);

**REVERSE STRING (Realtime) in JAVASCRIPT**

<h2 style="text-align: center;">Reverse The String in Real Time with Decrementing For Loop.</h2>  
 <input type="text" name="" id="reversTheName" onkeyup="reverseString()">  
 <input type="text" name="" id="showData">  
  
 <script>  
 function reverseString(){  
 var str = document.getElementById('reversTheName').value;  
 var newString="";  
  
 for(var i=str.length-1; i>=0; i--){  
 newString += str[i];  
 }  
 document.getElementById('showData').value = newString;  
 }  
 </script>

**PRIMITIVE vs REFERENCE DATATYPE in JAVASCRIPT**

Primitive values are data that are stored on the stack. Primitive value is stored directly in the location that the variable accesses. Primitive types include Undefined, Null, Boolean, Number, or String. Primitives are values, they have no properties. Updated: JavaScript has 6 primitive data types: String, Number, Boolean, Null, Undefined, Symbol (new in ES6). With the exception of null and undefined, all primitives values have object equivalents which wrap around the primitive values, e.g. a String object wraps around a string primitive. All primitives are immutable.

Reference values are objects that are stored in the heap. Reference value stored in the variable location is a pointer to a location in memory where the object is stored.

The Basics: Objects are aggregations of properties. A property can reference an object or a primitive.

[] === [] **//return false as they are reference datatype**

1 === 1 **//return true as they are primitive datatype**

obj1 = {val: 5}

obj2 = obj1

obj3 = {val: 5}

Here obj1 and obj2 are equal (===) not obj3 because the data stored by these objects are at different position.

**ECMASCRIPT 6 (ES6)**

ES 6: 2015

ES 7: 2016

ES 8: 2017

**BABEL** is a transpiler that can covert ES6 code into ES5.

**LET and CONST**

var is function scoped and let is blocked scoped.

|  |  |
| --- | --- |
| function check(){  **let** a = 10;  if(true){  **let** a = 20;  console.log(a); //a will give 20  }  console.log(a); //a will give 10  } | function check(){  **var** a = 10;  if(true){  **var** a = 20;  console.log(a); //a will give 20  }  console.log(a); //a will give 20  } |

**const PI = 3.14 (value is not changed due to const)**

PI = 3.24 (it will gives an error due to const)

**DESTRUCTURING**

It simply breaking down a complex structure into simpler parts.

const biodata = { name: ‘aayush’, age: 30, born: ‘India’ } **//it will create an object**

If we want to access any value of an object, we have to do:

biodata.age;

But if we don’t want this method, we can do destructuring as:

const name = biodata.name; **//in this method, we have to do for all keys again and again**

**OR**

const {name, age, born} = biodata; //**//in this method, we can do for all once**

name; **//it will give aayush as an output**

**TEMPLATE STRING**

let show = “My name is “ +name+ ”My age is “ +age+ ”I was born in ”+born; **//old method to access values of an object**

let show = `My name is ${name}. My age is ${age}. I was born in ${born}.`; **//new method using template string (usage of back tick ` `)**

**DEFAULT ARGUMENTS**

Variable pass at the time of function definition is **parameter**.

Variable pass at the time of function calling is **argument**.

function funAge(age=18){

return age;

}

funAge(); //it will give 18 as output

funAge(20); ////it will give 20 as output

**OBJECT PROPERTIES**

const biodata = { name: ‘aayush’, age: 30, born: ‘India’ }

const shortobj = { name, age, born }

shortobj; **//it will return name as aayush, age as 30 and born as India**

**ARROW FUNCTIONS**

function add(){ **//can be written as const varadd = function add()**

let a = 5;

let b = 10;

console.log(`The sum of $(a) and $(b) is $(a + b)`);

}

add();

**OUTPUT: The sum of 5 and 10 is 15**

**ABOVE FUNCTION WRITTEN IN ES5**

const funadd = () => {

let a = 5;

let b = 10;

console.log(`The sum of $(a) and $(b) is $(a + b)`);

}

funadd();

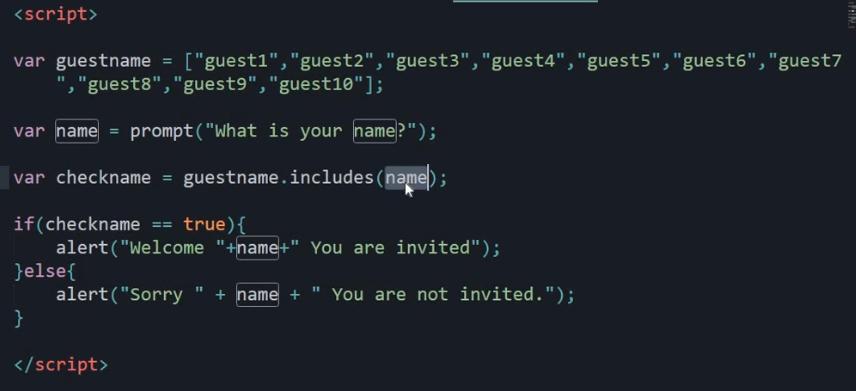
**OR**

const funadd = (a, b) => console.log(`The sum of $(a) and $(b) is $(a + b)`);

funadd();

**ECMASCRIPT 7 (ES7)**

**INCLUDES (method in array)**



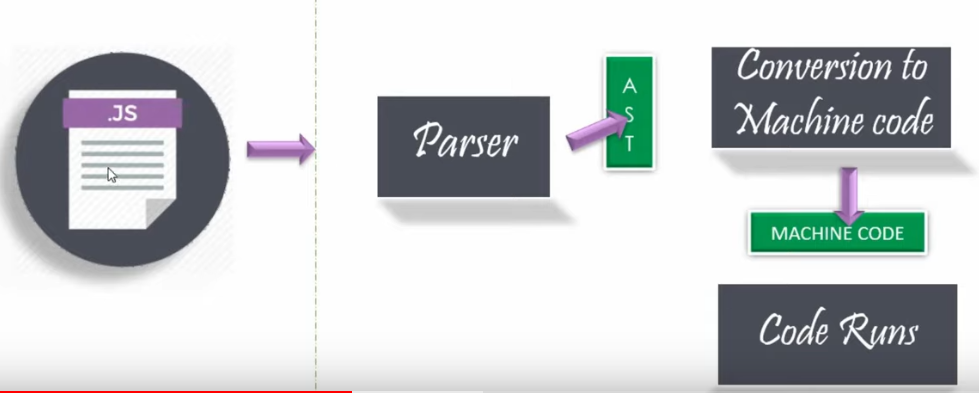
**EXPONENTIATION (to give power function like Math.pow)**

let square = 2 \*\* 2

square; **//it will give 4 as an answer**

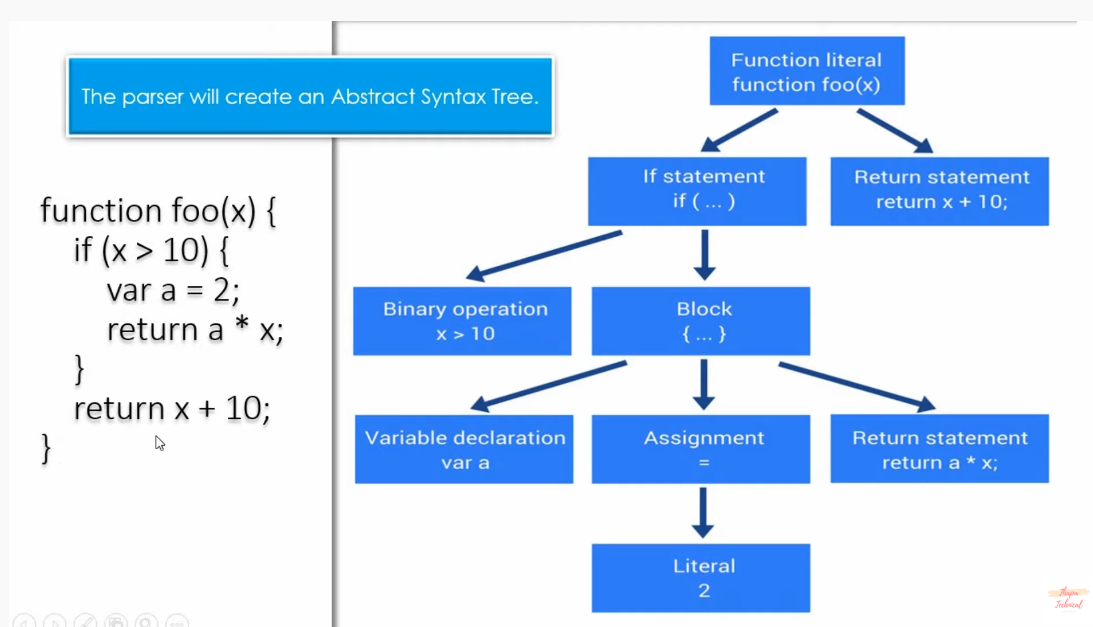
# **JAVASCRIPT EXECUTED PARSER & JS ENGINE**

How JavaScript code runs in the browser JS Engine. JavaScript is what called a Client-side Scripting Language. That means it is a computer programming language that runs inside an Internet browser (a browser is also known as a Web client because it connects to a Web server to download pages). When the browser loads the page, the browser has a built-in interpreter that reads the JavaScript code it finds on the page and runs it.



Parser will check your code line by line. After checking, it will produce a data structure called as AST.

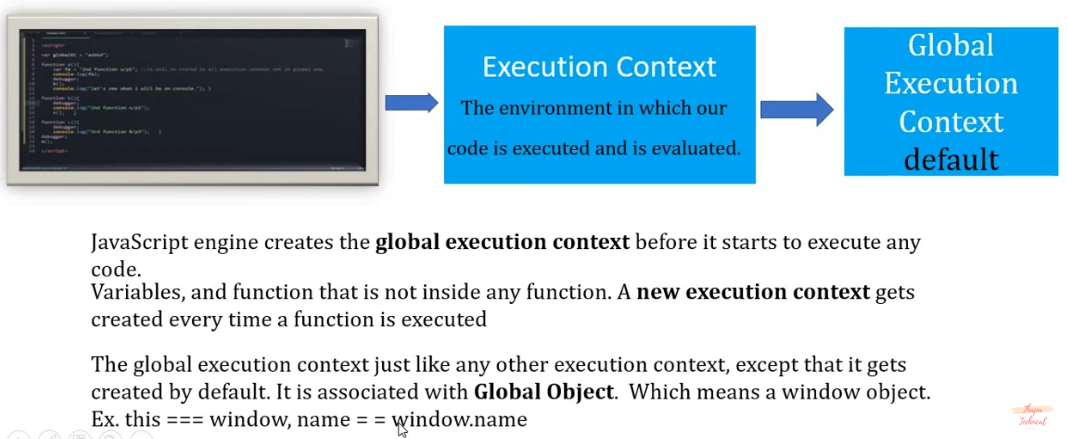
AST represent for Abstract Syntax Tree.



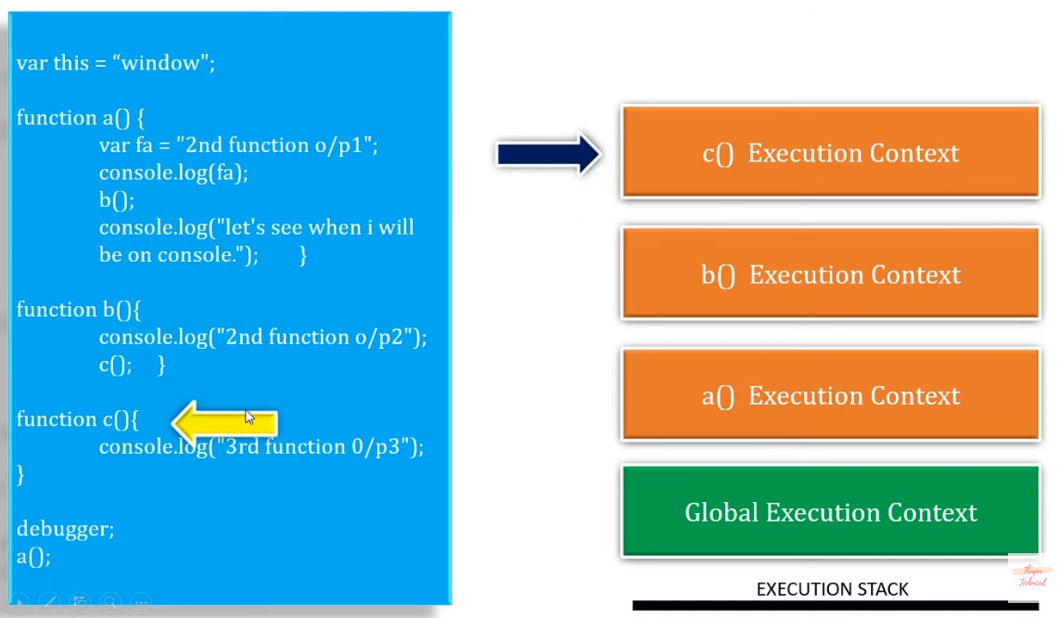
# **EXECUTION CONTEXT AND EXECUTION STACK IN JAVASCRIPT**

In JavaScript, the **Execution context** is an abstract concept that holds information about the environment within which the current code is being executed. Remember: the JavaScript engine creates the global execution context before it starts to execute any code. Execution context has three properties:

* Variable Object
* Scope Chain
* this Variable

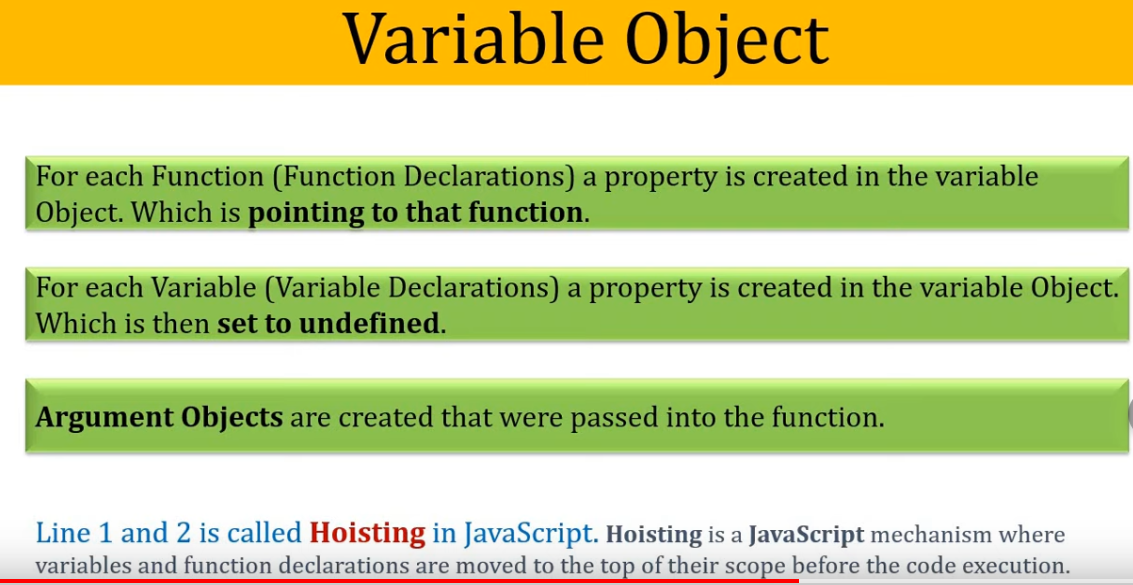


**Execution stack**, also known as “calling stack” is a stack with a LIFO (Last in, First out) structure, which is used to store all the execution context created during the code execution.



# **CREATION PHASE & EXECUTION PHASE & HOISTING IN JAVASCRIPT**





**FUNCTION HOISTING**

|  |  |
| --- | --- |
| **sum(5, 10); //HOISTING**  **function sum(a, b){**  **var add = a + b;**  **console.log(add);**  **}**  **In this case, due to hoisting function is moved to top of their scope before execution.** | function sum(a, b){  var add = a + b;  console.log(add);  }  sum(5, 10); |

**Exception in Function Hoisting**

|  |  |
| --- | --- |
| sum(5, 10); //Reference Error  (function sum(a, b){  var add = a + b;  console.log(add);  }) | sum(5, 10); //Type Error  var sum = function (a, b){ //var sum = undefined;  var add = a + b;  console.log(add);  } |
| sum(5, 10); //Reference Error  const sum = (a, b) => {  add = a + b;  console.log(add);  } | |

**VARIABLE HOISTING**

|  |  |
| --- | --- |
| **console.log(a); //undefined**  **var a = 10;**  **in this case, due to hoisting variable is moved to top of their scope and set to undefined.** | var a = 10;  console.log(a); |

**Example:**

<script>

console.log(job);

var job = "tester"

console.log(job);

function test(){

console.log(job);

var job = "programmer"

console.log(job);

}

test();

</script>

**Output:**

**Undefined**

**Tester**

**Undefined**

**Programmer**

**SCOPE CHAIN**

The scope chain is used to resolve the value of variable names in javascript. Without a scope chain, the Javascript engine wouldn't know which value to pick for a certain variable name if there are multiple defined at different scopes. The scope chain in javascript is lexically defined, which means that we can see what the scope chain will be by looking at the code.

At the top of the scope, the chain is the global scope, which is the window object in the browser (global in NodeJS). Besides from the global scope, functions have their own scoping of variables. The scope chain can be determined by looking at where functions are defined.

When resolving a variable, inner functions first look at their own scope. If the variable cannot be found in its own scope it will climb up the scope chain and looks for the variable name in the environment where the function was defined.

**“USE STRICT”**

**This statement instruct the browser to use the Strict mode which is a reduced and safer feature set of Javascript.**

<script>

function add (a, b){

sum = a + b; **//we have not defined sum as variable still it will display output**

console.log(sum);

}

add(10,20);

</script

So, we use **“use strict”** above the code so that this variable cannot show display and set to not defined.

<script>

function add (a, a){

sum = a + 20; **//duplicate parameters error will show if we use “use strict”**

console.log(sum);

}

add(10,30);

</script>

**THIS KEYWORD IN JAVASCRIPT**

this keyword refers to an object, that object which is executing the current bit of javascript code. In other words, every javascript function while executing has a reference to its current execution context, called this. Execution context means here is how the function is called.

**console.log(this);** //alone it refers to global object i.e. window object.

function add (a, b){

sum = a + b;

console.log(sum);

**console.log(this);** //it will also refers to global object i.e. window object.

}

sum(10,20);

const **student** = {

name: “aayush”,

age = 30,

**sum**: function(){

var add = 2+2;

console.log(add);

**console.log(this);** //in a method under object, this refers to object owner i.e. **student**

}

}

thapa.sum();

**TYPES in JAVASCRIPT**

JavaScript makes an arbitrary distinction between values: Primitive values and Objects. Primitive values include boolean, numbers, strings, null and undefined. While everything else in a JavaScript is said to be an object which means **window, JSON, Math, and even functions and arrays are Objects** as well.

let a = 1; **//number**

let b = “aayush” **//string**

let c = true; **//boolean**

let d; **//undefined**

let e = null; **//object**

const youtube = {

name: “aayush”,

age: 30

}

console.log(typeof( {} )); **//it will give an object**

console.log(typeof( [] )); **//it will give an object as we have declared array**

console.log(typeof( function\_name() {} )); **//it will give an function**

**ARRAY.ISARRAY() METHOD**

The isArray() method in JavaScript determines whether an object is an array. This function returns true if the object is an array, and false if not.

const array = [1,2,3];

console.log(array);

const **objectarray** = {

0: 1,

1: 2,

2: 3

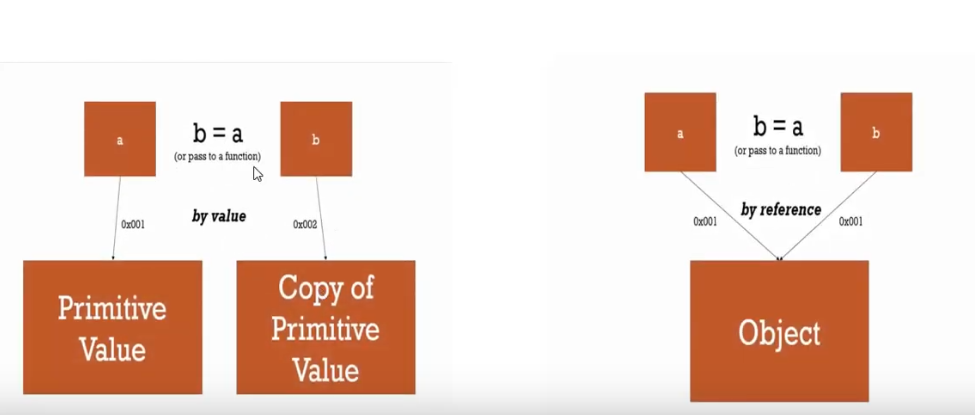
}

console.log(objectarray);

console.log(typeof(array)); **//it will return object as type**

console.log(Array.isArray(**objectarray**)); **//it will return false**

**PASS BY VALUE VS PASS BY REFERENCE IN JAVASCRIPT**



**PASS BY VALUE (Primitive Values)**

let a = 5;

let b = a;

**b = b + 5; //it will change only value of b not a**

console.log(a); **//it will give 5**

console.log(b); **//it will give 10**

**PASS BY REFERENCE (Non-Primitive like Objects)**

const youtube = {

name: “aayush”,

age: 30

}

const twitter = youtube;

**twitter.age = 35; //this will change value of age in both object**

console.log(youtube); **//it will give the same output as twitter**

console.log(twitter);

const arr1 = [1,2,3,4]

const arr2 = arr1;

arr2.push(5); **//it will change both array as array is an object too**

console.log(arr1);

console.log(arr2);

**NOTE: if you don’t want to change one array if another array is changed, do:**

**const arr2 = [].concat(arr1);**

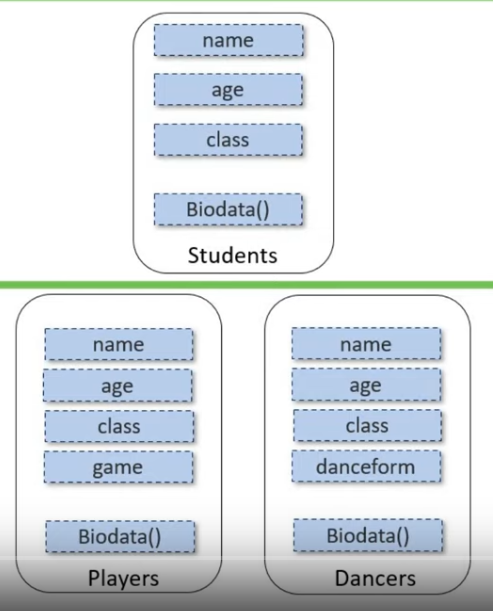
# **CLASSES, OBJECT & INHERITANCE IN JAVASCRIPT**

An object is an instance of a class. Using a class, we can create many objects which all share methods and properties.

A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword **class**, and the **properties are assigned inside a contructor() method**.

This technique does not support **hoisting** which means that firstly you need to create a class then only you can create instance of those class.

ECMAScript6 provides the keyword **class** to create a class.



**INHERITANCE**

**class classname{**

**constructor(){ //defines properties of class**

**}**

**method(){ //defines method of a class**

**}**

**}**

class Student{

constructor(**name, age, branch**){ **//defines properties of class**

**this.names**=name;

this.ages=age;

this.branches=branch;

}

biodata(){ **//instance method of class(if we write static biodata(), the subclass cannot access that method of parent class)**

console.log(`Hello ${**this.names**}. Your age is ${**this.ages**}. Your Branch is ${**this.branches**}.`);

}

}

class Player extends Student{ **//child class of Student(parent class): Inheritance**

constructor(**name, age, branch**, game){

super(**name, age, branch**); **//it will ensure that this child class can access properties of parent class using super method**

this.games=game;

}

biodata(){ **//method of parent class Student**

console.log(`Hello ${**this.names**}. Your age is ${**this.ages**}. Your Branch is ${**this.branches**}. You play ${**this.games**}. `);

}

play\_biodata(){ **//different method of accessing method of parent class**

**console.log(`${super.biodata()}`);**

}

}

let obj1 = new Student(**‘aayush’, 30, ‘IT’**); **//create an object of class Student**

let obj2 = new Player(**‘rahul’, 30, ‘CSE’, ‘Cricket’**); **//create an object of class Player**

obj1.biodata();

obj2.biodata();

obj2.play\_biodata();

# **SYNCHRONOUS VS ASYNCHRONOUS PROGRAMMING IN JAVASCRIPT**

Synchronous javascript code is executed in sequence – each statement waits for the previous statement to finish before executing. Asynchronous (Ajax Web Application Model)) javascript code doesn't have to wait – your program can continue to run. Asynchronous request does not block the client i.e. browser is responsive.

**SYNCHRONOUS**

const fun2 = () => {

console.log(“fun2 is starting”);

}

const fun1 = () => {

console.log(“fun1 is starting”);

fun2();

console.log(“fun1 is ending”);

}

fun1();

**OUTPUT:**

fun1 is starting

fun2 is starting

fun1 is ending

**ASYNCHRONOUS**

const fun2 = () => {

setTimeout( () => {

console.log(“fun2 is starting”);

}, 3000); **//it will accept two parameters, one is function which will perform its operation after some time which is given in second parameter in milliseconds.**

}

const fun1 = () => {

console.log(“fun1 is starting”);

fun2();

console.log(“fun1 is ending”);

}

fun1();

**OUTPUT:**

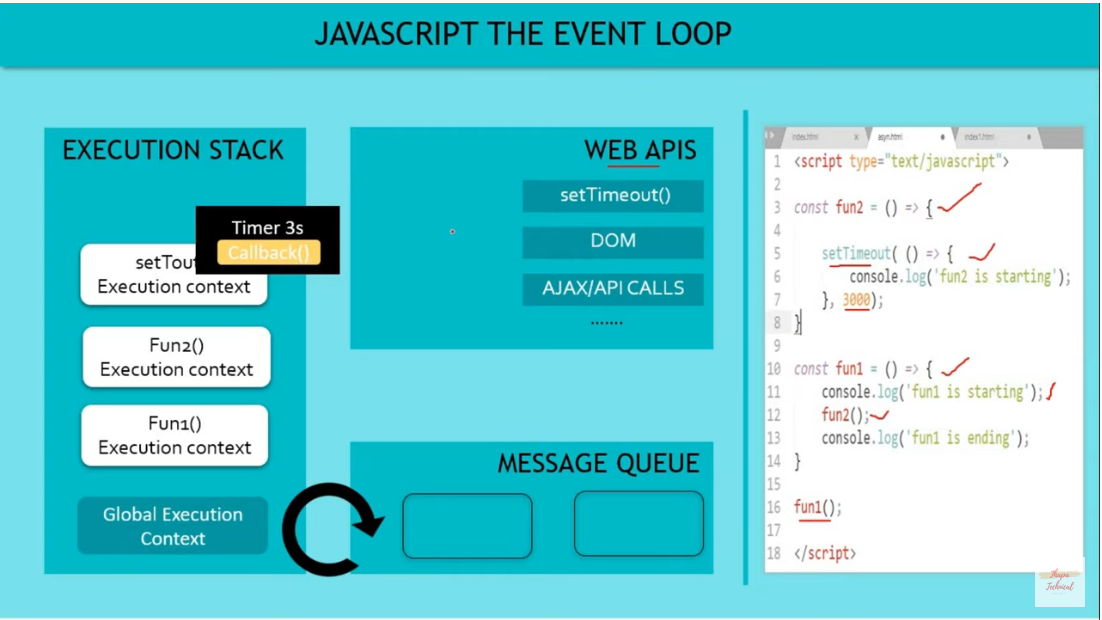
fun1 is starting

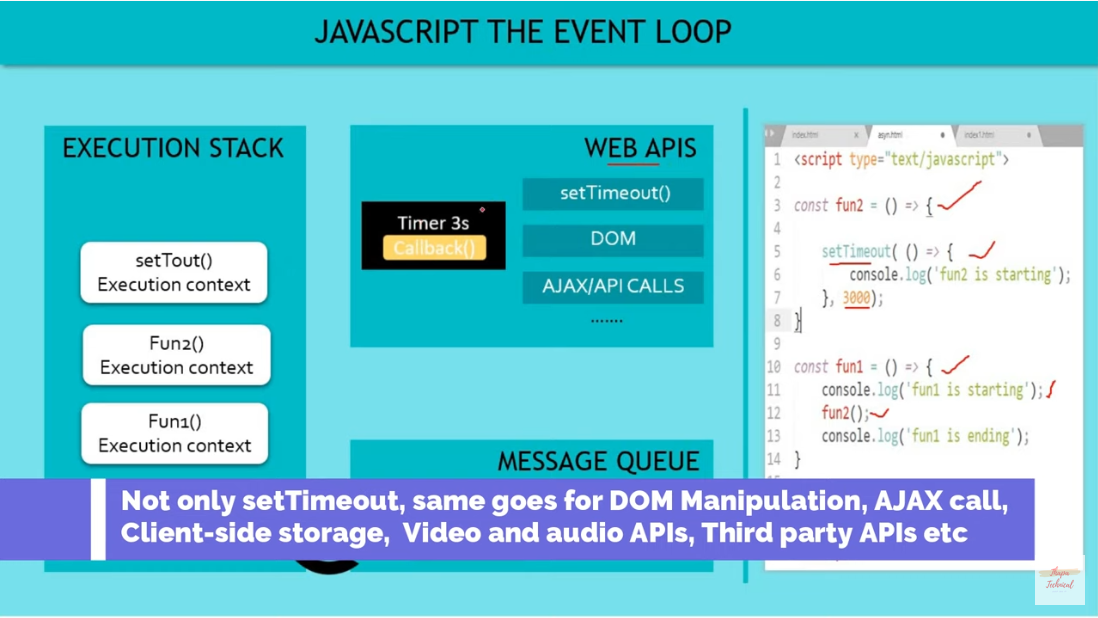
fun1 is ending

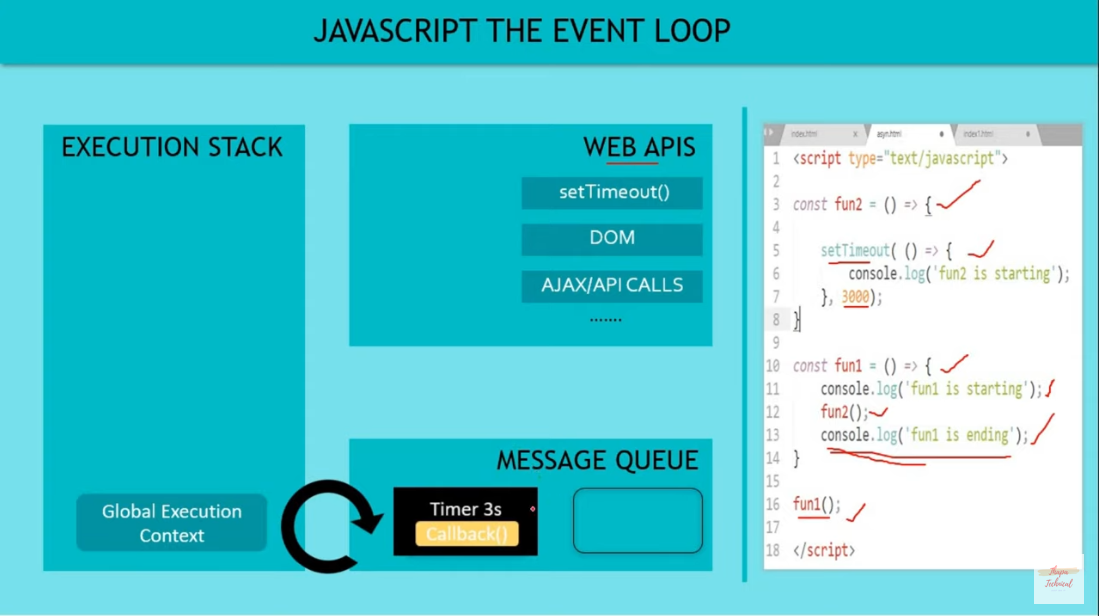
fun2 is starting

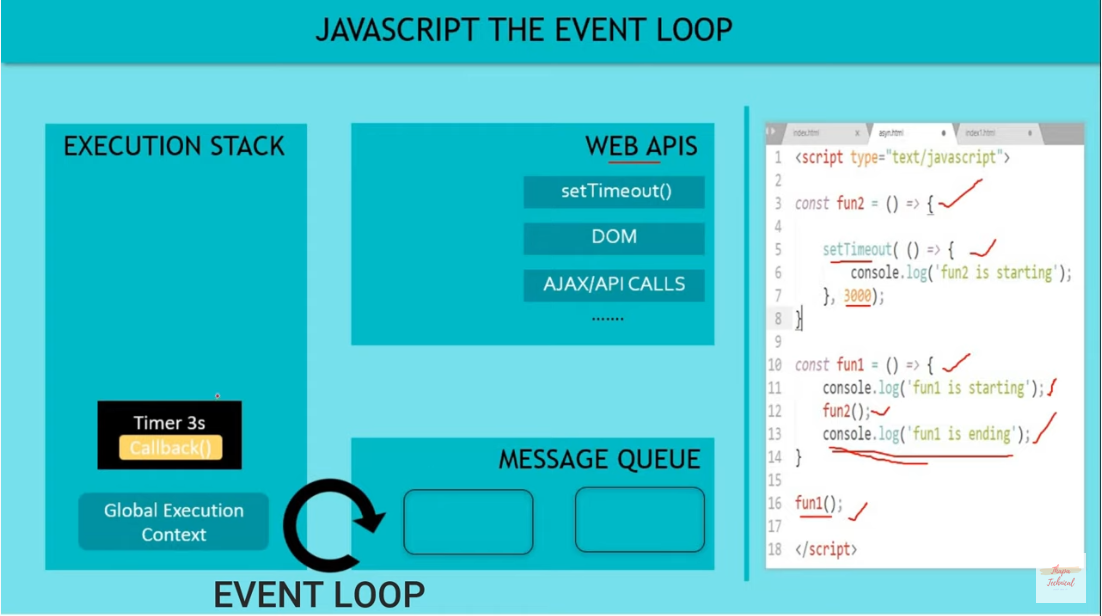
# **EVENT LOOP IN JAVASCRIPT**

The Event Loop is one of the most important aspects to understand about Asynchronous JavaScript. Any JavaScript code that takes too long to return back control to the event loop will block the execution of any JavaScript code in the page, even block the UI thread, and the user cannot click around, scroll the page, and so on.

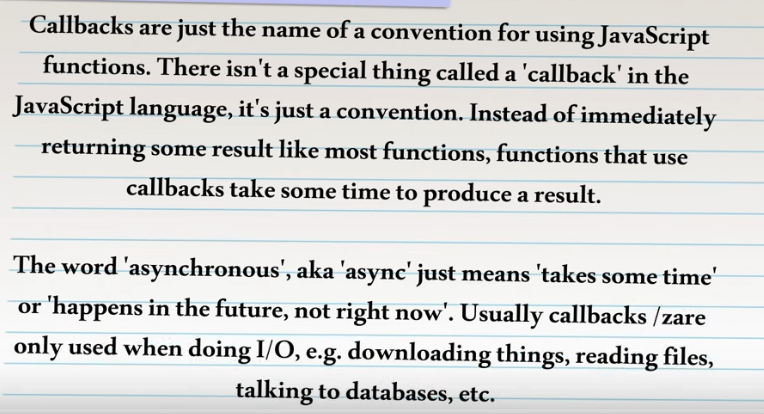








# **CALLBACK HELL IN JAVASCRIPT**



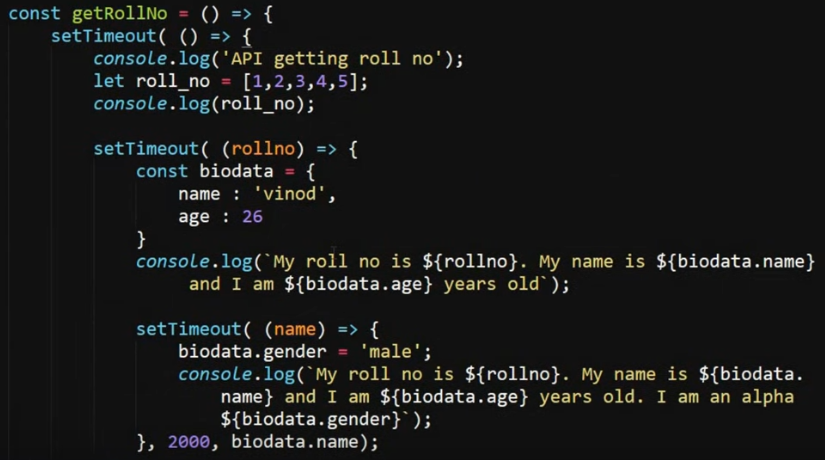


**Example:**

**Get student Roll Number after 2s**

**Get name and age of 2nd Roll Number after 2s**

**Get gender of 2nd Roll Number after 2s**



This becomes very complex when more function includes. To overcome this problem, we use **Promises** in JavaScript

**PROMISES IN JAVASCRIPT**

Promises are used to handle asynchronous operations in JavaScript. A Promise has Three states: **fulfilled**: Action related to the promise succeeded.

**rejected**: Action related to the promise failed.

**pending**: Promise is still pending i.e not fulfilled or rejected yet.

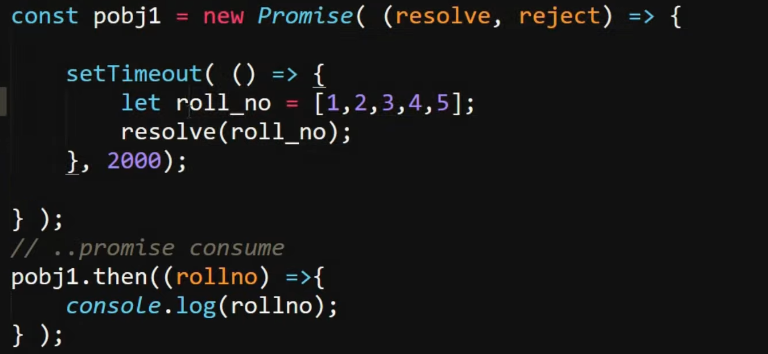
Promises can be consumed by registering functions using .then and .catch methods.

**then()** is invoked when a promise is either resolved or rejected.

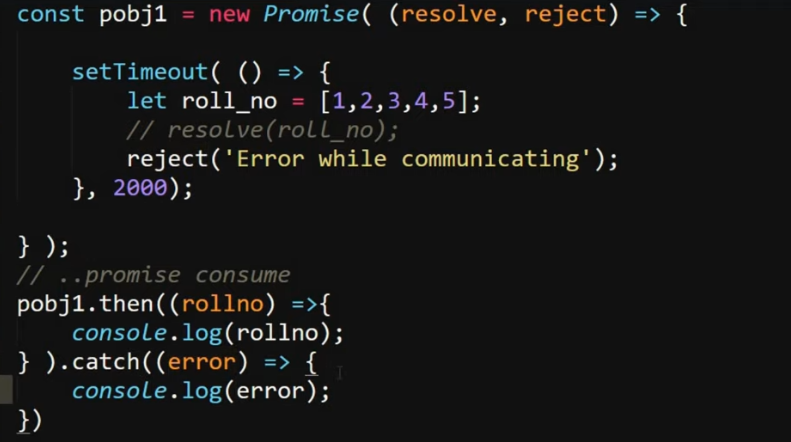
**catch()** is invoked when a promise is either rejected or some error has occurred in execution

**A Promise is an object that keep track about whether a certain event has happened already or not. It also determines what happens after the event has happened.**

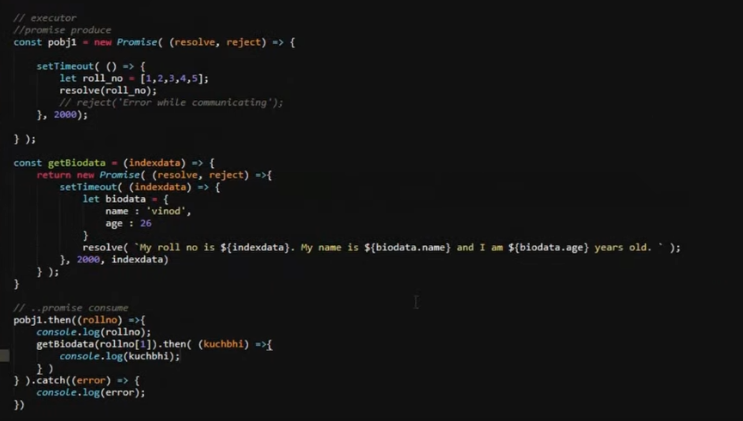
**Example**: On Success



**Example**: On Reject



**Example**: Two connected events



# **ASYNC AWAIT IN JAVASCRIPT**

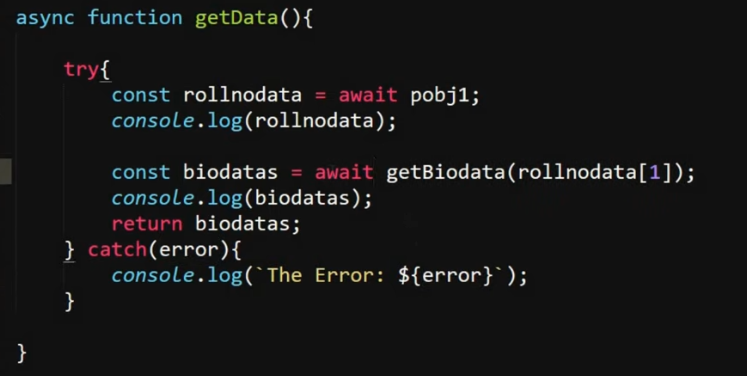
# The async function declaration defines an asynchronous function, which returns an AsyncFunction object. When an async function is called, it returns a **Promise**. When the async function returns a value, the Promise will be resolved with the returned value. When the async function throws an exception or some value, the Promise will be rejected with the thrown value. An async function can contain an await expression, that pauses the execution of the async function and waits for the passed Promise's resolution, and then resumes the async function's execution and returns the resolved value.

**There is a special syntax to work with promises in a more comfortable fashion, called “async/await”.** It is easy to understand and use.

**Difference**: refer above example which is used in Promise. Main difference is in consumer part

|  |  |
| --- | --- |
|  |  |

**ERROR HANDLING IN JAVASCRIPT**



# **LOAD JSON FILE [API] WITH AJAX CALL USING FETCH API IN JAVASCRIPT**

The Fetch API provides a **fetch()** method defined on the window object, which you can use to perform requests. This method returns a **Promise** that you can use to retrieve the response of the request.

**Example**: To fetch data using fetch() function



**.json()** is used to get data in json (object) format

actualdata.Countries[101]; **//to fetch India data**

**Example: To show data on your webpage**

<p id = “apiindia”> </p> **//HTML**

const mydata = actualdata.Countries[101].Country;  **//JAVASCRIPT**

document.getElementById(‘apiindia’).innerHTML = mydata;