**JavaScript**

* JavaScript also need a compiler which convert JS code into machine understandable code. Browsers like chrome already comes with this compiler that’s why we are able to write a JS file and run into these browsers using html file.
* We can use JS engine (v8 [google chrome], spider monkey[firefox]) to run these files individually. These engines follow a standard in order to run/execute our code known as ECMAScript / ES. ECMAScript is a Standard for scripting languages.
* ECMA: European Computer Manufacturer's Association.
* Google chrome is environment and v8 is engine for our JS code.
* People took engine [v8] out of environment [chrome] and put it separately in backend server with few additional things like nodeJS.
* JS is work based on JIT [just in time] compilation concept where code will be compiling fast and running. JIT concept is tied to JS engines, so JS is compiler language or interpreted language is depends on which functionality engine has where we are running the code.
* Nodejs, Deno, etc. are some implementation which can convert our JS code to machine understood language, so we don’t need to attach JS code into HTML to run it like we have to do in browsers.
* **Datatypes:**
* primitive datatype: [string, number, null, undefined, boolean, symbol]. Primitive datatypes are immutable that mean they can’t be altered.
* Object datatype: [arrays, functions, objects] mutable.
* **Difference between JS mutable and immutable datatypes:**

Values are immutable, variables are not. They hold a reference to their (primitive) values

var str = "testing";

var str = "testing, testing";

console.log(str); // testing, testing

var fruits = ["apple", "banana", "orange"];

fruits[0] = "mango";

console.log(fruits); //["mango", "banana", "orange"]

The values associated with both mutable and immutable types can be changed through reassignment as the above examples with strings and arrays show. But then, these data types have associated functions(methods) that are used to manipulate the values belonging to each data type. This is where mutability/immutability is seen. Since arrays are mutable, any manipulation by an array method affects the array directly. For example,

var fruits = ["mango", "banana", "orange"];

fruits.pop();

console.log(fruits) //["mango", "banana"]

The array.pop() method deleted "orange" from the original fruits array, but with strings for example,

var name = "Donald Trump";

name.replace("Donald", "President");

console.log(name)//Donald Trump

the original string remains intact!

Immutability disallowed any altering of the original string by the string method. Instead, the method produces a new string if the method operation is assigned to a variable like so:

var name = "Donald Trump";

var newName = name.replace("Donald", "President");

console.log(newName);//President Trump

<https://stackoverflow.com/questions/16115512/understanding-javascript-immutable-variable>

<https://javascriptweblog.wordpress.com/2010/09/27/the-secret-life-of-javascript-primitives/>

We can see from above example that we are able to access properties for primitive datatypes, for example name is a variable and we can do name.toUpperCase ; this is allowed because behind the scene primitive datatypes are wrapped in object datatypes, do operations as toUpperCase here and once this name change to upper case, it turns back to primitive datatype.

* **variables:** To reserve some memory. There are 4 ways to create a variable: without datatype, using var, using let, using const.
  + **var**: We can store String, number and Boolean in var datatype. If we declare a variable and don’t assign any value to it and print it, it will show undefined so until we assign any value to a variable its value will be undefined.

Whenever we create any variable in JS, it will be assigned to window object. Which is global context in JS. We should prevent creating variable in global context until it’s really required so suppose if we need a variable limited to a function so we should create it inside function only. If we create a variable using var in an empty block {}, it is still available outside this block as it assigned to global context.

Note: if we don’t give datatype like var to any variable in a function the JS will put this variable in window global context because we can declare variable without datatype globally but not inside function.

* + **let**: use to create block level variable which scope is limited to the block like a variable in function scope.
  + **const**: const is similar to let in case of scoping but doesn’t allow to change value.
* Number in JS are having a range and if we go behind that we will get below 3 output.

**Infinity:** if number is going to beyond limit positively.

**-Infinity:** If number is going beyond limit negatively.

**NaN:** If we try to do operation on non-number input as below.

Console.log(“ashvani”\*2); //output: NaN

* JS really throw an error for numbers if we are trying to do unprocessed operations as:

Console.log(10/0); //output: undefined

* To take input from user, we can use **@alert** or **prompt()** but we need HTML page to use them. [so, prompt () is a function which will be taking input from user with the help of html page]
* If we are printing String with any variable on console, we can use either + or , in between them as below. Here comma (,) will give a space automatically between variable and String.

console.log("my name is: "+name);

console.log("my name is:",name);

* There is another way if we want to take values on run time. We can use backticks ` ` and ${variable\_name} to print it as below.

console.log(`

Here we are printing values at runtime

    my name is = ${name};

    the value is = ${value};

`)

This concept is knowns as **interpolation**.

* We can get type of any variable like Boolean, var etc., using below reserve JS keyword.

**typeof** variableName

* **Functions:**
* Whenever we want to write block of code and reuse it, we can write it as function. Syntax are:

/\*In JS we have two type of functions: Regular functions and Arrow functions.  Arrow functions is a new concept which introduced with ES6 \*/

//regular functions

function square(x) {

    return x \* x;

  }

  console.log(square(10));

//arrow functions

  var squareVal = (a) => a\*a;

  console.log(squareVal(3));

function fun1() {

    console.log("inside fun1");

}

**Call a function:** fun1();

**Reference of a function:** fun1;

* JS functions don't have a return type but if we can return data from functions without return type.
* In JS, a function can be called without giving required argument.
* JavaScript supports overriding not overloading, meaning, that if you define two functions with

the same name, the last one defined will override the previously defined version and every time a call will be made to the function, the last defined one will get executed.

* **Context**

We have a concept of context in JS, especially global context. If we try to code in browser console and try to write below comparison statement, it will give true as in browser **window** is global context. Node has a similar context with different name.

* + var name = “ashvani”;

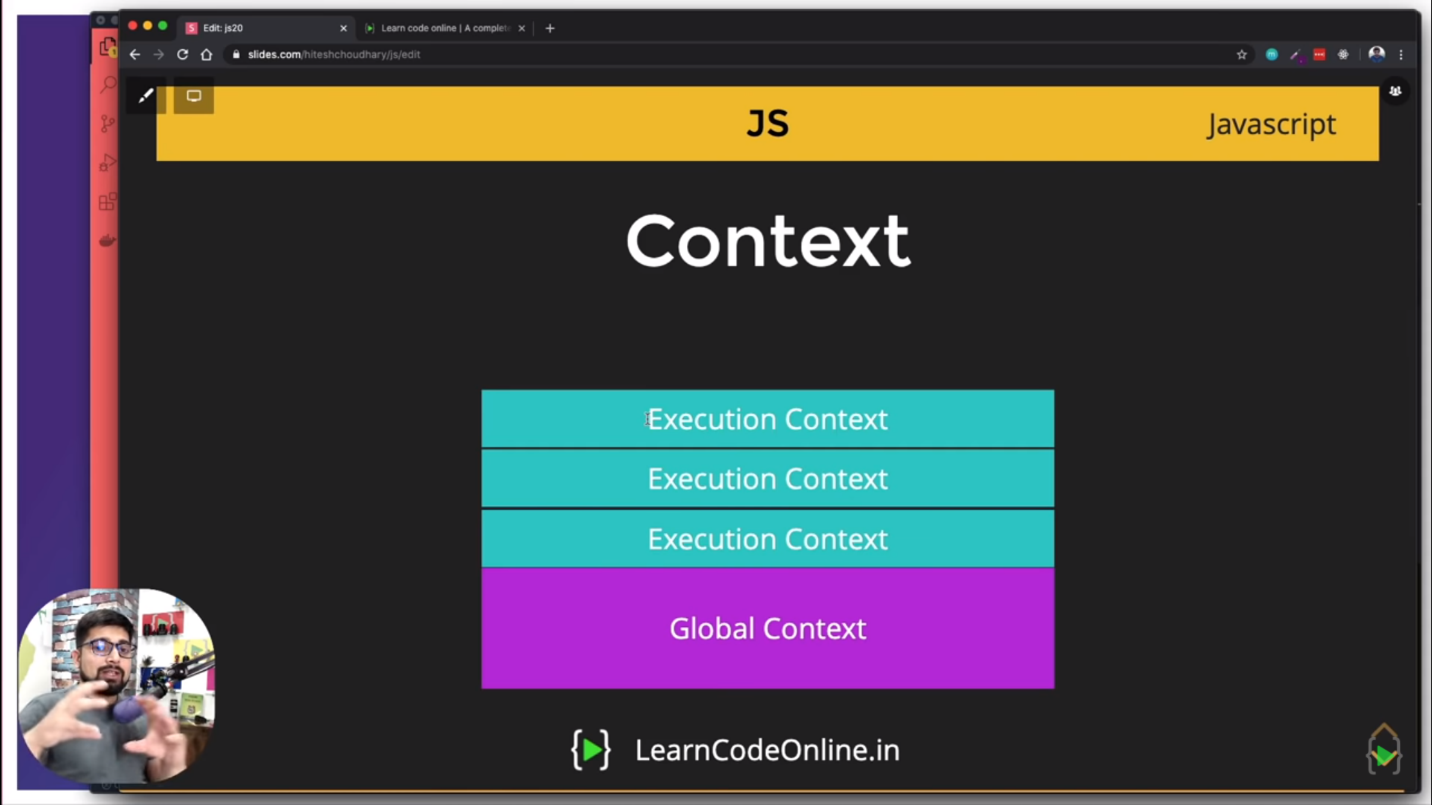
If(window.name === name) {

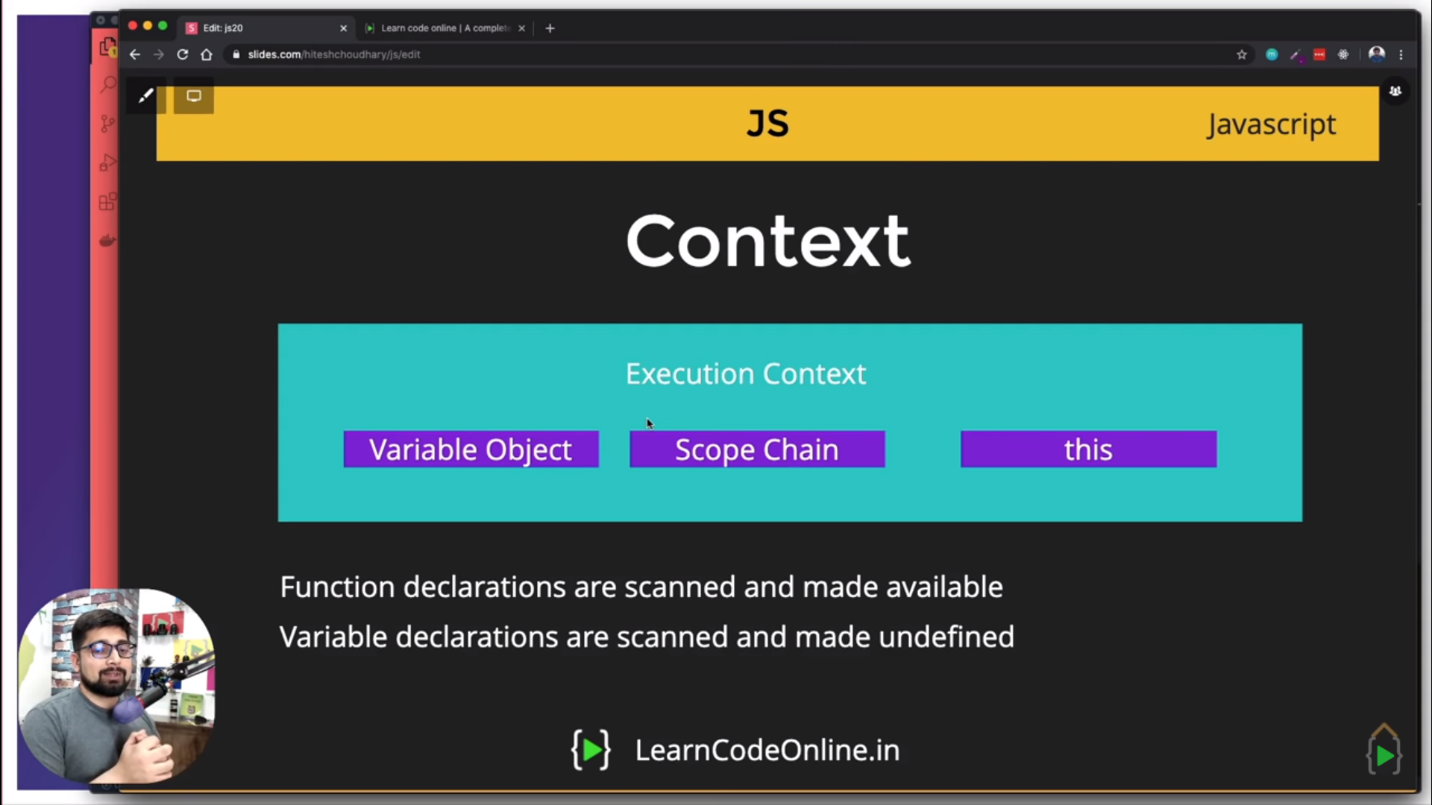
Console.log(“this is a true statement”);

}

Majorly we have global and execution context, global context is for whole process while we write the code and execution context starts once we start calling out methods.

So JS engine will be creating a Stack of execution contexts top on global context. These execution contexts will be added in stack while calling a method or executing a line of code and will be getting removed from stack once execution done.





<https://medium.com/@happymishra66/execution-context-in-javascript-319dd72e8e2c#:~:text=Global%20execution%20context%20(GEC)%3A,inside%20the%20global%20execution%20context.&text=Each%20function%20has%20its%20own%20execution%20context.>

* **Hoisting:**

Declaration –> Initialization/Assignment –> Usage

//code hoisting : Declaration –> Initialisation/Assignment –> Usage

contextFunction(100);

function contextFunction(a) {

    console.log(`contextFunction() can be called from anywhere in this JS file as function declaration are scanned and made available ${a}`);

}

var hoistvariable = function hoist1(a) {

    console.log(`this function is assigned to a variable so we can't called it before declaration and it will throw an error if called before assignment`);

}

hoistvariable(10);

console.log(`the value of b {${b}} will be undefined as for global context, variable declaration is scanned and made undefined`);

var b = 100;

/\* To understand the difference between global context and function context \*/

console.log(`my full name is printing before declaration is: ${name} ${sirName}`);

var sirName = "sharma";

var name =  "ashvani";

console.log(`my full name is printing after declaration is: ${name} ${sirName}`);

function getName() {

    var name = "aaryan";

    var sirName = "vashist";                                                     /\*This variable is limited till this function\*/

}

getName();

var name = "aayush";

console.log(`sirName will come from global context and its value is: ${sirName}`);

console.log(`name is: ${name}`);

/\*The concept behind this is:

If a variable is getting declared within function then it will be part of  function context [while code scan].

for global context it will consider variable which was declared globally.

if we assign different value to these variables in globally then it will override previous value.

if we assign different value to these variables in function then scope will be limited to funcion only.

\*/

* **this keyword:**

this keyword gives us access to global context and global context differs in different situations. If we write same code to print value of this keyword in browser console and on VS code running top on node, it will give us different result as global context differs between them.

Console.log(this);

In case of node, it gives us an empty scope represented by {}. In case of browser, it gives us access to window object.

*For all regular function calls,* ***this*** *points to* ***window***object*.* Binding for this keyword is available only for regular functions but not for arrow functions.

* **Object:**

It’s a structure in Key-Value pair where keys don’t have any specific data type as String and all. We just give Key as a variable/Object name. Object can have any datatype as value. We can have methods and arrays in object as well.

var object = {

    firstName : "Ashvani",

    lastName : "Sharma",

    address : "Bijnor",

    role : "Admin",

    loginCount : 21,

    fbSignIn : true,

    courseList : [],                                    //array in object

    buyCourse : function (courseName) {                 //function in object

        this.courseList.push(courseName);

    },

* **Difference between let and var:**

The main difference between let and var is that scope of a variable defined with let is limited to the block in which it is declared while variable declared with var has the global scope. So, we can say that var is rather a keyword which defines a variable globally regardless of block scope. For example, if we declare a variable with let in for loop then it’s scope will be limited to the loop.

* **forEach:**

gone through each element of array and work according to implementation. For ex:

let i = 0;

var newArray = [];

array1.forEach(

    (i,j) =>

    newArray.push(i)

);

* ***for in* and *for of* loop:**

"for of" loop is mostly used for Array and "for in" loop is mostly used for Objects.

for(const n of array1) {

    console.log(n);

}

for(const n in object1) {

    console.log(`key is: ${n} and value is: ${object1[n]}`);

}

* **How to grab web element in JS:**

There are multiple ways to grab web elements in JS and few are as below:

Suppose we have an index.html page where we have some DOM elements mentioned in it so if we want to access any element by its tag we can use below code for same.

<!DOCTYPE html>

<head>

    <title>This is for DOM</title>

</head>

<body>

    <div class = "nav"></div>

    <div class = "body">

        <h1>This is Header <a href= #> click me </a></h1>

        <img src="">

    </div>

    <div class = footer></div>

</body>

</html>

* **To grab HTML from header:**

*var title = document.getElementsByTagName("h1")[0].innerHTML*

output: "This is Header <a href="#"> click me </a>"

* **To grab Text from header:**

*var title = document.getElementsByTagName("h1")[0].innerText*

output: "This is Header click me"

* **IDs are unique for an element, but classes can be repeated.** So, we can have same class assigned to different div, but single id can be assigned to only 1 div.
* We can define document query selector using class or Id.

When we mentioned id in document, we represent it using **#id** , when we represent a class, we represent it using **.class**

**Ex:**

*var number = document.querySelector(".container")*

* If we have a HTML link in a document referred by a class “*container*” and if we want use querySelector for this document as above then,

**number.innerText:** It will show only Test except URL/link

**number.innerURL:** It will show both text and URL as well

* We can manipulate above fetched data at any given time.

For ex:

*number.innerText*

"1000

Followers"

*number.innerText = "ashvani"*

*number.innerText*

"ashvani"

So, modification of this is super easy once we grab the hold of these values.

* **Prototype:**

Prototype is an object that is associated with every functions and objects by default in JavaScript, where function's prototype property is accessible and modifiable and object's prototype property (aka attribute) is not visible.

Prototype chain is basically JS inheritance like for any object data type as Array if we check it’s prototype \_\_proto\_\_ we will find property that behind the scene array is also an object using prototype chaining.

* **Lexical Scoping:**

A lexical scope in JavaScript means that a variable defined outside a function can be accessible inside another function defined after the variable declaration. But the opposite is not true; the variables defined inside a function will not be accessible outside that function.

* **Node Element in JS:**

Whatever starts with <angular brackets> is known element in JS world. For ex:

<ul>, <li>, <span> etc.

Whatever we write without <> is known as text in HTML.

* **Scope problem in JS:**

**{ } =>** this is empty scope in all languages except JS, so in all other languages a variable declared in { } will have a limited scope. But in JS if we have a variable declared with var, it can be accessed outside of this scope. If we want to get rid of this issue, we should declare these variables using let keyword.

* If we see default for loop created by JS developers, they have used let for index variable because if they have declared it using var and we want to use same variable outside of this for loop then it might create an issue for us/
* JS consider function as a scope.

If we are writing JS code for backend, we should use let keyword instead of var. The reason is if we declare any variable with var and try to fetch it before initialization then we will get undefined as output but in case of let, we will get proper error message as:

console.log(lastName);

var firstName = "Ashvani";

let lastName = "Sharma";

error:

ReferenceError: Cannot access 'lastName' before initialization

console.log(firstName);

var firstName = "Ashvani";

output: undefined

<https://www.javascripttutorial.net/es6/difference-between-var-and-let/>

<https://www.javascripttutorial.net/javascript-execution-context/>

* **String literal:** Defined by ` `

This is new way to define a String which is helpful whenever we have ‘ or “ in any String as below:

Whenever we have “ or ‘ in a String then we have to use it along with \ but is case of String in ` `

There is no use of \ .

* **Map in JS:**

Map is a collection of elements where each element is stored as a Key, value pair. ... Map object can hold both objects and primitive values as either key or value. When we iterate over the map object it returns the key, value pair in the same order as inserted.

We can use traditional for-of loop and latest forEach loop to traverse through MAP. Both gives slightly different results.

console.log(`\*\*\*\*\*\*\*\*\*\*\*\*Traverse map for key and value using traditional for of loop\*\*\*\*\*\*\*\*\*\*\*`);

for(let [k,v] of map1) {

    console.log(`key is ${k} and value is ${v}`);

}

console.log(`\*\*\*\*\*\*\*\*\*\*\*Traverse map using for each loop : It will return value first then key\*\*\*\*\*\*\*\*`);

map1.forEach((val, k) => {

    console.log(`value is ${val} and it's key is ${k}`);

* **De-structure in JS:**

The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

We can apply this concept on Array and Object both as below.

Left and right hand side data should be of same type and name/keys should also be same.

var {name, lastName} = object1;   //here key should be same as object's key

const [name1, lastName1] = array1;                                  //Here name1 and lastName1 are index so it's value can be anything

* **Spread and Rest operator in JS: (…args)**

...args is a rest parameter. It always has to be the last entry in the parameter list and it will be assigned an array that contains all arguments that haven't been assigned to previous parameters.

**spread:**take input from array and seprate them out as individual values

**rest:** take individual values and make array out of it.

function spread(a, b) {

    return a + b;

  }

  var myA = [5, 4, 100];

  console.log(spread(...myA));

  function rest(a, b, c, d, ...args) {

            return(a+b+c+d);

  }

  console.log(rest(1,2,3,4,5,6,7));

**Visual Studio Code**

* **h1 + enter:** header for html code will appear on .html file in VS code
* **script:** script for html code will appear
* **ctrl + ~ :** PowerShell terminal will appear
* **! + enter:** Boiler plate code for HTML page will appear
* Command to run a .js file: Ex, to run file1.js

**node folder1/file1.js** (full path of file/filename)

* **Shift + alt + down arrow :** to get duplicate of a line.
* **Shift+Alt+F** : format js code