JS Notes:

* Everything in JS happens inside an EXECUTION CONTEXT.
* EC – Memory – key,value pair – Variables and Functions

JS is synchronous and Single Threaded Language.

One command at a time in a particular order.

Execution Context is created in two phases:

1. Memory Creation Phase – Set Variables to Undefined and Functions – Copy paste entire code
2. Code Execution Phase –

Call Stack: It’s a stack, every time at the bottom, we have global execution context. When a function is invoked, when a new execution context is created, it is pushed inside the stack, whenever execution context is deleted, it is deleted.

Call Stack maintains the order of execution of Execution Contexts

Hoisting:

Access variables and functions before defined.

Any code which is not inside any function is a global space.

var a = 10;

function b(){

var x = 10;

}

Console.log(window.a) === console.log(a)

Console.log(this.a) // This is similar to window.a because window = this at global space

Console.log(x) //Error not definted

Console.log(b.x) // Prints 10

this keyword:

1. At global level this points to the global object. Whenever any js program is run, a global context is created, a global object is created, and a this is created.
2. At global level this === window

Undefined vs Not defined:

1. Undefined is a default value of a variable when initialised. Until and unless variable is assigned any value, it is undefined. Even when variable has any value, it is set to undefined initially in Memory Allocation Phase of Execution Context.
2. Undefined is not equal to Not defined. Whenever something is not defined and we access it, it gives an error – Undefined.

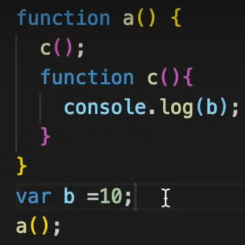
Js is a loosely/weakly typed language. Does not attach variable to any specific data type.

NEVER do this mistake – a = undefined;

Scope:

Scope in js is directly related to Lexical Environment

Where you can access a specific function/variable.



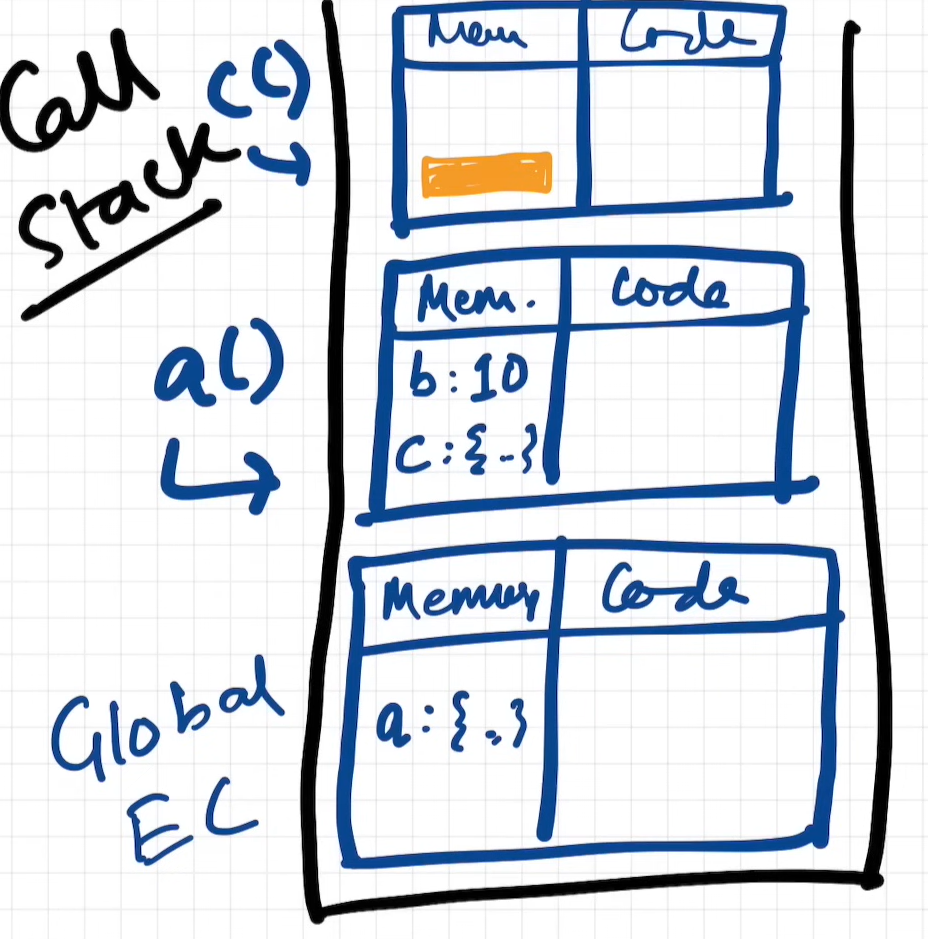
1. Variable b is accessible to a() as well as inside c().
2. Find var b inside local memory of c. If not present then go to lexical parent which is a(). Search var b inside local memory of parent which is a(). Here it found out var b inside a(). But if it doent find over here, then it goes to the lexical parent of a() which is nothing but the global context.
3. This way of finding the variable references is scope chanining.
4. Lexical env is created when global context is created.

Graphical user interface

Description automatically generated

1. b is now not defined as it is outside of the scope.

Lexical Env:



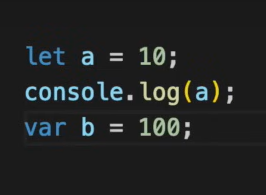
Orange part in c() EC is a reference of lexical environment in a().

Lexically c is inside a. a is inside global. So pointers become:

Null 🡪 Global 🡪 a() 🡪 c()

Let & const :

let & const declarations are hoisted.



Graphical user interface, text, application

Description automatically generated

Note the difference of scopes in a & b because of let and var.

We can access values inside global scope even before assigning actual value. But let and const will assign undefined values and take them into Script scope. We cannot access values of Script scope.

Scenario:

console.log(a);

let a = 10;

Result:

Uncaught ReferenceError: Cannot access ‘a’ before initialization.

Scenario:

console.log(a);

var a = 10;

Result:

undefined

Scenario:

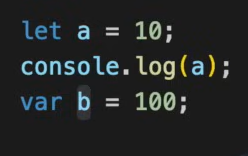
console.log(a);

Result:

Uncaught ReferenceError: a is not defined

Temporal Deadzone:

Time between when a let,const variable was hoisted and actually assigned a value



Result:

Graphical user interface, application

Description automatically generated

Means b is attached to the window object (Global space)

Value of a is stored inside some different space. Which is definitely not global space.

Graphical user interface, application

Description automatically generated

Same as window because in global context window = this

Redefining a let variable is not allowed. It will give a syntax error.

Graphical user interface, text, application

Description automatically generated

***Uncaught SyntaxError: Identifier ‘a’ has already been declared***.

A screen with numbers and letters on it

Description automatically generated with low confidence

Same error as above

But this is possible in var.

A screenshot of a computer

Description automatically generated with low confidence

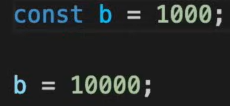
Outputs – 10

A screen with numbers and letters on it

Description automatically generated with low confidence

***Uncaught SyntaxError: Missing initializer in const declaration***.

Const b = 100 // is the only correct expression.



***Uncaught TypeError: Assignment to constant variable***.

SyntaxError: Redeclaration of let variables, declaration of const variable but not assignment.

TypeError: Assigning const variable twice.

ReferenceError: Variable in temporal dead zone and we try to access it.

const > let > var.

Always put declaration and initializations at start of the code. That is how we avoid temporal dead zone.

let & const scope.

Blocks:

{

} this is a block. It is a group of statements which can be used together. Eg. If(condition){ multiple statements }

Block Scope:

{

var a = 10;

let a = 20;

const c = 30;

}

Console.log(a) //10

Console.log(b) // ReferenceError

a is hoisted in global scope.

b & c are hoisted in block scope.

Shadowing:

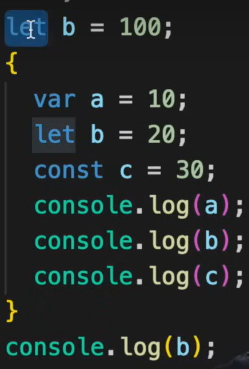
var a = 100;

{

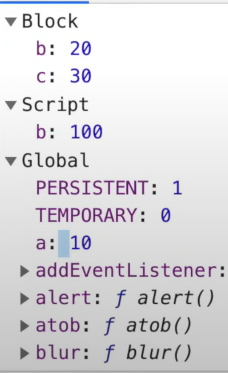
var a = 10;

}

var a inside block shadows var a outside the block and it also overrides the value. Because var runs in global scope.



Result:



Same happens in case of const.

Rule of Shadowing:

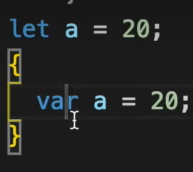
let with let is allowed

const with const is allowed

var with var is allowed

But,

let with var is ***illegal***



var with let is ***legal***

Closure:

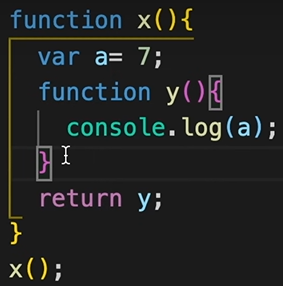
A function bundled with its lexical env/scope.

Graphical user interface

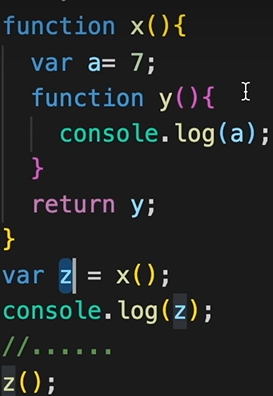
Description automatically generated with medium confidenceGraphical user interface, text, application, chat or text message

Description automatically generated

The above things are allowed in JS

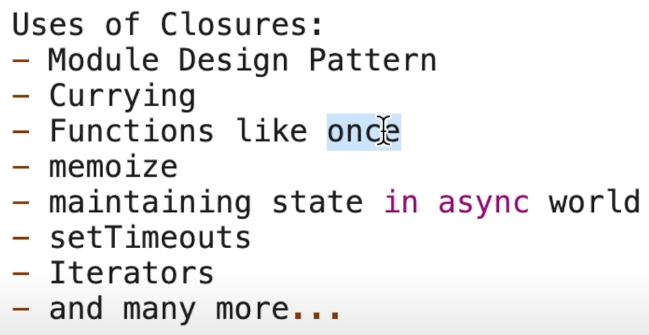


Returning a function from within a function



Here in above example. z contains function y() that is z contains code { console.log(x) }.

Although x() is no longer present in the call stack, when y was returned to z. It not only returned the code y consist of but also it returned lexical environment bundled. And that lexical environment consists of the value of x which was 7. So whenever z() is called, the inside code remember the value of x. This entire phenomenon is called closure.



First Class Functions:

Function Statement === Function Declaration:

function function\_name(){

} This is statement

Function Expression

var b = function(){

} This is a function expression to assign a function to a variable.

Difference: Hoisting

Calling a function before creating it. 🡪 Here function statement will be executed and in case of function expression variable is assigned a value undefined being a variable

Text

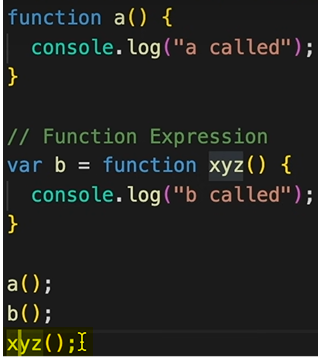
Description automatically generated

Anonymous Function:

1. Function without any name is anonymous function.
2. An anonymous function cannot be defined directly. It has to be used as a value to a variable.
3. Used in a place where functions are used as values

Named Function Expression:

Anonymous function with a name is named function expression.



Call xyz() returns into an error

First Class Functions:

Ability of JS to use a function as a value and to pass a function as an argument to another function or returned from another function is called as First-Class Function.

Above ability of a function makes Functions first class citizens

Call-back function:

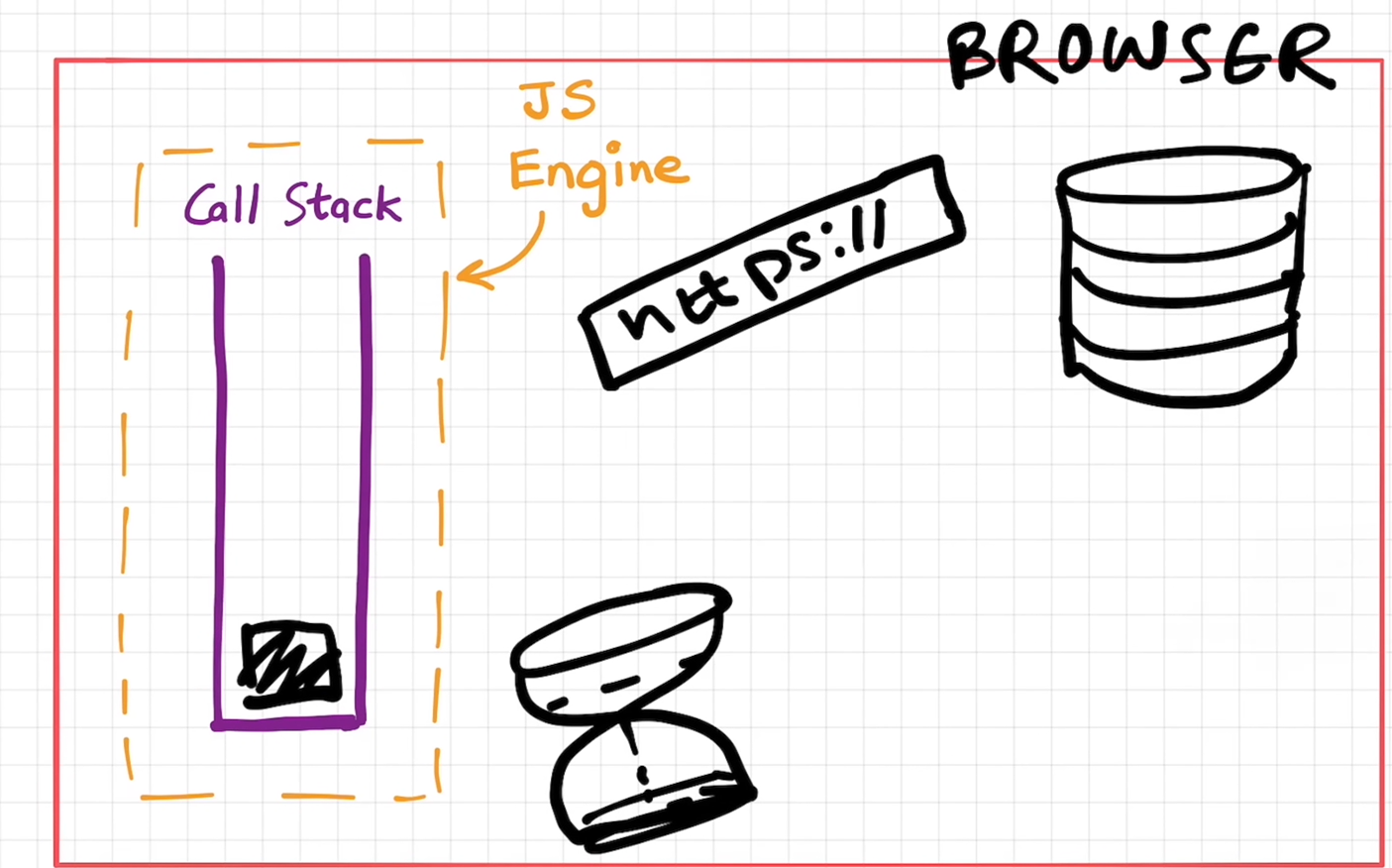
A function which is passed as an argument in another function is called a call-back function.

x(function y(){

}) y is call back function

Event Loop:

Call Stack does not have a timer.



Global Object – window:

setTimeout function lies inside window object which is a part of browser.

Window.setTimeout === setTimeout

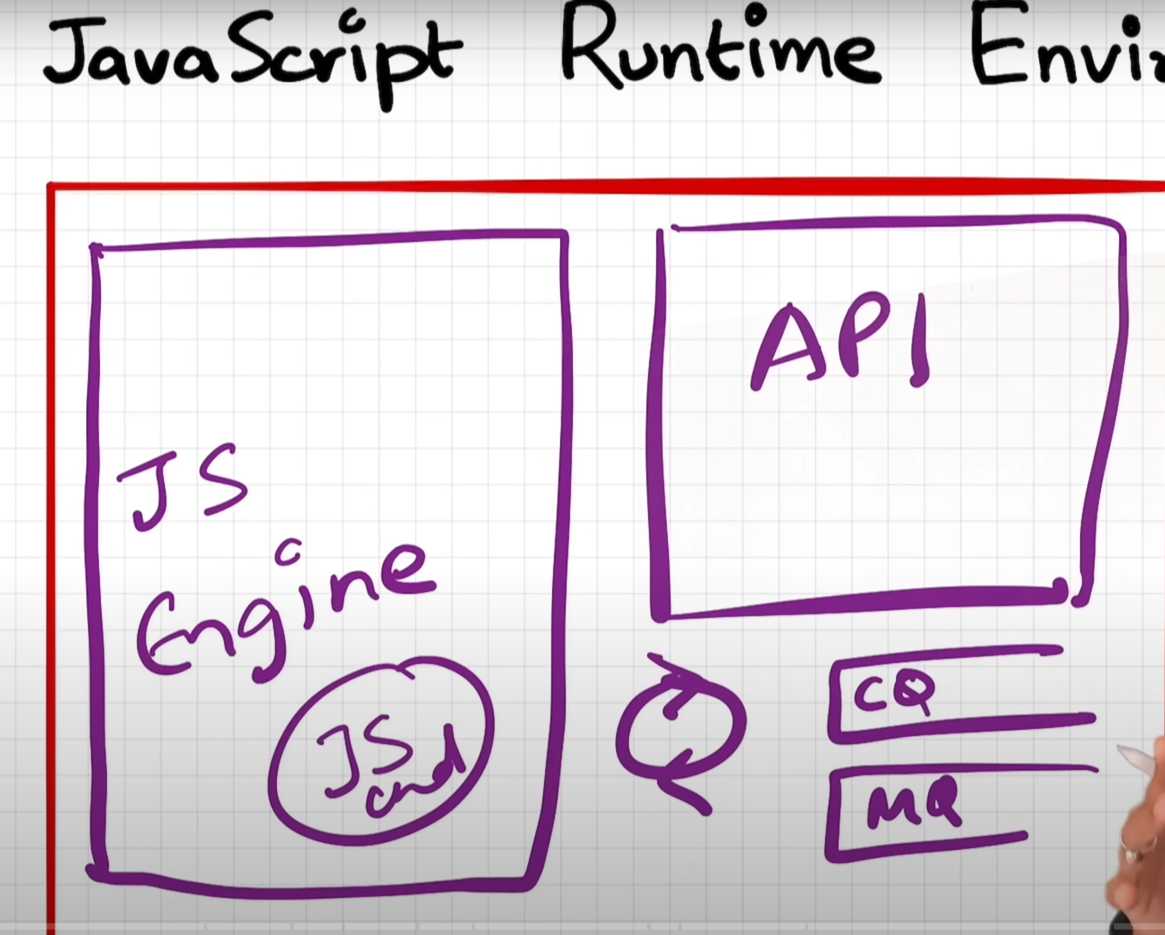
Window.location === location

Window.fetch() === fetch

Microtasks?

1. All the callback functions which come through Promises come under Microtask Queue
2. Mutation Observers

Other callbacks etc. SetTimeout/DOM API goes under Callback Queue == Task Queue JS Engine:



JS can run in anything – PC, browser,

1. JSRE –
   1. JS Engine – Heart of JSRE
   2. API
   3. Event Loop
   4. Callback Queue
   5. Microtask Queue

Want to run JS inside a water cooler? We must have JSRE inside a water cooler lollzz

API –

1. LocalStorage
2. Console
3. SetTimeout
4. Fetch

JS engine should follow ECMAScript standards. You can define your own JS engine.

Node Js uses v8 engine. V8 is written in C++.

First JS engine was developed by its developer. Which is also known as SpiderMonkey today.

CODE 🡪 Parsing 🡪 Compilation 🡪 Execution.

Code is broken down into Tokens:

Syntax Parser:

Conversion to AST – Abstract Syntax Tree.

Text

Description automatically generated with medium confidence

This is AST

JIT – Just In Time Compilation

JS can behave as compiled as well as an interpreted language. Initially it was designed to run as interpreted language. It is upto JS engine to run as interpreted or compiler or to use both.

AST goes to Interpretter

Interpretter gets the byte code of the stubs..

Interpreter takes help of Compiler to prettify the code. (Memory heap and call stack)

Callbacks:

JS waits for none. But what if we want to execute something after some time?

Callback Hell

Inversion of control