JAVASCRIPT (loosely typed language)

(Single thread, Synchronous, Execution context, Call Stack, callback, microtask queue and event loop)

Js working = Js call stack ,Js engine & browser.

JS engine

Call stack

Browser

Multitask queue

Event loop

callback queue

Everything in JS happens in an execution context (inside the box which is made up of two components, 1) Variable environment (memory): all values to datatypes(key-value pair) and functions are kept here.

2) Thread of execution(code): all code is executed line by line)

JavaScript is a SYNCHRONOUS SINGLE-THREADED(can only execute one command at a time) language.

In the first phase of memory allocation, JS declares variables and stores them with “undefine” even though is value is provided. For functions, the exact code will be copied.

In the second phase where code is being executed, the assigned values(if any) will be given to respective declared variables. Whenever a function is invoked, a whole new execution context is created for that function.

After the code is executed for the function or the main code, every execution context is deleted.

All this execution context is managed by a call stack, which push & pop the functions as they are executed on top of main execution context(Global execution context). In the end when code ends the main execution context is pushed out of stack.

Loosely typed/weakly types, means any data type can contain any data. No strict rules like char for character or string for strings etc floats or doubles for fractions etc.

(Hoisting => Execution context, undefined, not defined for variables and functions)

Allocation of memory to a variable or function is called hoisting.

A variable not defined(reference error) is not the same as undefined. Not defined means the variable hasn’t been declared or initialized, while undefined simply means the variable is called before its declaration.

when a variable is called before declaration it will be undefined and if that variable hasn’t been declared it will throw an error(not defined).

When a function in normal standard <fname> {..} is invoked before the function code itself, It will work fine, If it's called inside a console.log(), it will print the entire function, however if we get arrow convention var <fname> = () => {..}, it will be undefined.

(this keyword and window)

Whenever a program is run, a global execution is created, a global object(window) is created and ‘this’ variable is created.

window is a global object which is created along with the global execution context. All memory is stored in this object, hence when we say console.log(a) it will fetch variable ‘a’ from this window, we can also use console.log(window. a) to get a similar result.

At the global level, this points to the window object.

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

(scope, scope chain & lexical environment)

The scope is accessing variables, we can access global variables anywhere and in any deep functions, but we can't do vice versa.

Scope chain is a Lexical environment, where Lexical environment is a referencing chain that follows a hierarchal pattern (parent-child relationship).

(Let and Const, temporal dead zone, ref error, syntax error and type error)

(Strict scale) = Const > Let > Var

Let and const are declared in memory of script and not global.

Const has to be declared and initialized in the same line, while var and let can be initialized anywhere.

Temporal dead zone is associated with hoisting let and const. Time btw hoisting(declaring) let & const and it being initialized a value (f(Time),(undefined -> 10)) is called the temporal dead zone.

Let and const are stored in a different memory space but not in global(var is stored in global). We can’t access these variables with window and this keywords too.

When we try to access a variable in the temporal dead zone, it gives us a reference error.

Can’t use the same variable name for let and const (or) didn’t initialised const in same declaration line will throw a syntax error. (var is allowed)

The value of const cannot be changed anywhere after its initialization resulting in a type error.

The benefit of using let and const over var is we can understand any errors arising. Var might cause some unexpected errors.

(Block, Block Scope, shadowing and illegal shadowing)

Block is a space where code can be written( between {}). Ex conditional statements, loops, functions etc. Block is also known as compound statement. Generally used to execute multiple statement instead of one.

Block scope is basically scope of a block, scope = accessibility, hence accessibility of a block. let and const inside a block cannot be used outside, however, var can be used outside.

Shadowing is basically to overwrite(PERMANENTLY for var) the value of variables in a block. Ex a=10(global) and a=100(block), console.log(a) will print 100.(only for var because var is present in global context). For Let as its in the script not in global and it retains its value there even if block is updated.

Illegal shadowing is when try to shadow a variable with different variable types, ex global(let a = 10;) block(var a = 100;), syntax error; NOTE: vice versa will work in the above case as Var is pretty flexible. NOTE: if we use a function as a block, then above case is fine.

Closures, setTimeout, Encapsulation.

A function along with its lexical scope = closure, in simple words its like the summary of the function with reference to its outer enviroment, what values were captured in this function is called closure. So even after a function is ended, its been popped out of the call stack while entering the child function to that, it still holds a closure window.

Inside child function it Closure created a summary of the parent function for values to be used.

A SetTimeOut is a function that can be used to manipulate the timing of output of a function, it generally works well with closures.

Constructor function, garbage collector,

It is a function which has the capability to host multiple uses of a particular function and all these instances can be accessed whenever needed. this and new keywords are used for this method.

Example: function Counter() {

let count = 0;

this.increaseCounter = function () {

count++;

console.log(count);

};

this.decreaseCounter = function () {

count--;

console.log(count);

};

}

var newCounter = new Counter();

newCounter.decreaseCounter();

garbage collector takes unused variables and takes it out of memory, this is automatic in Javascipt, unlike C and C++ where developers are allowed to manually allot memory. If we use closures we cannot free the memory of a variable in closure that hasn’t been used yet. These unused variables in closure can build up and cause memory leak. To overcome this modern editors have smart garbage collectors where if a variable is not used it is dumped even inside a closure.

Function statement, function expression, Anonymous functions, named function statement, parameters and arguments,

//Function statement (or) declaration

function a() {

console.log("Hello");

}

//Function expression

var b = function () {

console.log("Hello");

};

//Anonymous function : Error: A function statement without function name

// function () {

// console.log("Hello");

// }

//Named function expression, Note: buzz(); cannot be accessed on a global level.

var b = function buzz() {

console.log("Hello");

};

//parameters and arguments, parameters are local to function while arguments are global.

function calc(parameter1, parameter2) {

let result = parameter1 + parameter2;

return result;

}

calc(argument1, argument2);

//First class functions/citizens, functions as values in return statements or arguments.

a(function(){

console.log();

})

Function statement can be hoisted but expressions can’t.

Anonymous functions are purely used to assign values, they are used in function expression.

The ability to use functions as values(in return statement, as arguments etc)

SetTimeout uses anonymous fucntions.

Callback function = SetTimeOut(), Blocking the main thread, event listeners(removal of event listeners)

//callback funcions, calling a function as argument is called as callback fucntion.setTimeOut uses this feature alot.

function x(){

}

x(function y(){

})

Block of main thread: main thread is also called as call stack, and as we know JS has a single call stack to manage all operations, so when a very heavy function is being executed and it takes say around 20-30 sec that function will occupy the stack and won't let it run further or clear it hence causing a delay in output, to avoid this we use callback functions which are this asynchronous functions that are called back(executed later) after a defined interval of time, which means in mean time code will continue to run and stack won't be blocked. Without these callback functions, we wont be able to do this asynchronous operation.

Event listeners are these block of code that captures an event over an element.

These event listeners form a closure, hence holding memory and they are heavy in nature, hence can slow down the code and cause memory leak, hence they need to be removed when not using.

Power of Browser = Event loop, callback queue(task queue), microtask queue, fetch and promise, mutation observer, starvation of the callback queue

The call stack can only execute one line/function at a time, once executed it is popped out. This call stack has no way of working asynchronous or access to many real-world features like time, servers, UI(DOM), location or Bluetooth etc. Hence in order to fully use these feature’s that are all provided by browsers, the java script engine has to use something called as an API. setTimeOut is not a part of JS, it is a browser API. Event listener accessible by .document are DOM APIS. Console.log is console API. Fetch APIS for server connection.

Settimeout functions are stored in browser(temp) and added to a callback queue with a timer(timer specified in the function), after the time expires(or) event is occurred(click,close,drag etc) this function is pushed back into the stack via an event loop, event loop acts like a gatekeeper btw callback queue and call stack.

Precedence = Main code > microtask queue > call back queue.

Promise: when a callback function expects and output. “If this is successful then perform this, analogy.”

Mutation observer: Changes in the DOM tree.

All the callback functions that come through promises and mutation observer will be stacked in microtask queue.

When microtask queue keeps on creating more and more microtask in this queue, this prevents the callback queue tfrom performing this delay is called starvation of the callback queue.

JS Engine, ECHMA script, compilers.

All javascript code is executed in JS Engine. Which is basically a compiler; compiler = tokenizer, parser , compiler & execution.

Interepter: compiles line by line, fast execution.

Compiler: compiles the whole code first in an optimized version then runs it, more efficient.

JIT: Interpreter + compiler, best of both worlds. AST -> interpreter ( code -> byte code) while being optimized altogether.

AOT: Ahead of time, the Compiler takes a piece of code which is to be used later & tries to optimize it.

The first JS engine(spiderMoneky) was an interpreter based compiler but all most recent compilers like Googles V8, Edge’s Chakra, Opera’s Caraken etc. are JIT(Just In Time) compilers.

The execution stage of JS Engine holds the call stack. Memory heap and garbage collector(mark and sweep algo) is also linked in this execution stage with this call stack.

Trust issues,Concurrency model, trust issues

Say code takes 10s to execute and setTimeOut for a function is given 5 seconds, even after 5 seconds is expired the SetTimeOut will not work as it waits 10s for main code to be executed. This is called as concurrency model or trust issues. Trust issue a bit different as it states that callback isn’t acting when specified as main thread is blocked.

When we put setTimeOut timer to 0ms, its generally known as defer. This code will be executed still after main code but main idea of doing is to make sure some code/feature is implemented at the end or later.

Higher order functions.

A function which takes another function as an argument or returns a function is called as higher order function.

Using these higher order fucntions greatly decrease number of lines of code, make code reusuable and avoid DRY code.

Map is a higher-order fucntion.

//Write a code that calculates area, circumference and diameter of circle,

//given the radius is in array.

//radius

const radius = [ 1, 2, 3, 4];

const radiusLen = radius.length;

//formulas

const area = function (radius){

return (Math.PI \* radius \* radius);

}

const circumference = function (radius){

return (2\* Math.PI \* radius);

}

const diameter = function (radius){

return (2 \* radius);

}

//manual iteration and output assignment.

Array.prototye.calculate = function(formula){

const output = [];

for(let I = 0; I < radiusLen; i++ ){

output.push(formula(this[i]));

}

return output;

}

//calling with map

// console.log(radius.map(area));

// console.log(radius.map(circumference));

// console.log(radius.map(diameter));

//calling with manual loop

// console.log(calculate(radius, area));

// console.log(calculate(radius, circumference));

// console.log(calculate(radius, diameter));

//caling with prototype array

// console.log(radius.calculate(area));

// console.log(radius.calculate(circumference));

// console.log(radius.calculate(diameter));

Map, filter and reduce higher order fucntions.

Map is used to transform(change content) of whole array.

Filter is used to find specific elements inside and array, ex Even, odd, positive, zero, negative, divisible or multiple of etc..

Reduce(or call it summarize), it is used to perform something on all available elements of an array, like sum, average , mean etc.

Reduce takes 2 parameters, accumulator(acc) and current(curr). Current is the present value of the array, accumulator is the needed value, set to 0 by default.

const values = [1, 2, 3, 4, 5];

//double, triple and binary values of this array

// const double = function(values){

// return ( 2 \* values);

// }

const triple = (values) => {

return values\* 3;

}

const binary = function(values){

return values.toString(2);

}

console.log(values.map((values) => {

return(2 \* values);

}))

console.log(values.map(triple));

console.log(values.map(binary));

MAP.

const arr = [1,2,3,4,5,6,-2,-4,0];

//find even numbers;

console.log(arr.filter((value) => {

return value % 2 == 0;

}))

//find odd numbers;

console.log(arr.filter((value) => {

return value % 2 != 0;

}))

//numbers greater than zero

console.log(arr.filter( (value) => {

return value > 0;

}))

//numbers less than zero

console.log(arr.filter( (value) => {

return value <= 0;

}))

//numbers divisible by 2 and negative

console.log(arr.filter((values) => { return values % 2 === 0 && values < 0}));

FILTER

const arr = [1, 2 ,3];

//const arrL = arr.length;

//sum of numbers

console.log(arr.reduce((acc, curr) => {

acc = acc + curr;

return acc;

}));

//max of numbers

console.log(arr.reduce((max,cur) => {

if (cur > max) {

max = cur;

}

return max;

},0));

//average

const sum = (arr.reduce((avg, curr) => {

avg = avg + curr;

return avg;

}, 0));

console.log("Average: " + sum / arr.length);

Reduce.