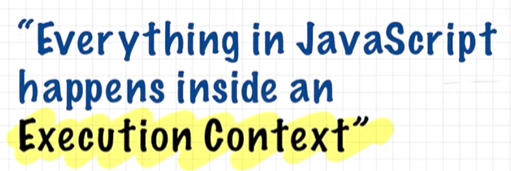
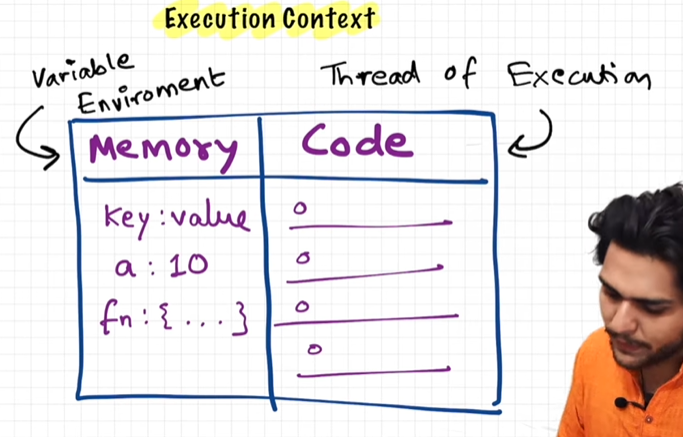
***Basics of JavaScript***



We can imagine “execution context” as a big box, which has two components. First, Memory component (also known as Variable Environment) and second, Code component (also known as Thread of Execution). In the memory component all variables and functions are stored in the form of key-value pair. And in the code component, whole code is executed line by line.



JavaScript can execute only one command at a time. By “synchronous single-threaded” it is meant that, only one command at a time is executed and that too in a particular order. This means, it can move to the next line, only when the current line has completed its execution.

A close-up of a computer script

Description automatically generated

Example:

A computer screen with colorful text

Description automatically generated

Let’s try executing this program in JavaScript. As soon as we run the program, a “global execution context” will be created, which will have two components – memory component and a code component. Now the JavaScript Engine skims through the program, and in the first phase “Memory Creation Phase” the memory will be allocated to all the variables and functions in the program.

1. Global execution context creation.

A blackboard with writing on it

Description automatically generated

1. Memory allocation: all the variables will be allocated a memory and will be assigned a special keyword called as “undefined”, and for the functions, whole code will be copied to the memory component.

A blackboard with writing on it

Description automatically generated

1. Now the second phase will take place. The “Code Execution Phase”. Now JavaScript runs through the program again and start executing the code line by line. Here in this phase, all calculations and actual value assignment take place.
2. If we see in the code, line number 1, var n = 2; so now “2” will be assigned to “n” in the memory component. And moving from line 2 to 5 we don’t have any operation to perform. So, we will move to line number 6.

A blackboard with writing on it

Description automatically generated

1. In line number 6, function “square” is being invoked. As soon as a function is invoked in JavaScript, a whole new Execution Context is created inside the Code Component.

A blackboard with yellow squares

Description automatically generated

1. Again, there will be two phases, the memory creation phase and the code execution phase, where firstly memory will be allocated to all variables and functions, and then code will start executing line by line. Below is the ss of memory creation phase.

A chalkboard with writing on it

Description automatically generated

1. Now in the code execution phase, the value of num will be replaced by the value of “n” i.e., 2 that is passed as argument, while invoking the function.

A chalkboard with writing on it

Description automatically generated

1. Now it will move to the next line, where calculation will take place – “var ans = num\*num;” so whatever will be the value we will get after “num\*num” will be assigned to “ans”.

A chalkboard with writing on it

Description automatically generated

1. Moving to the next line of code “return ans;” this will tell the function that your work is now over, and give the control back to the code where you (function) were invoked. So the return statement will give the control back to the line of code where the function “square” was invoked, i.e., “var square2 = square(n);” and the value that is returned i.e., “4” will be the answer for “var square2”. And as soon as the return statement is executed, and control is given back to the main program, then the execution context will was created inside the code component will be deleted.

A chalkboard with writing on it

Description automatically generated

1. In the same way, “var square4 = square(4);” will be calculated.

A blackboard with writing on it

Description automatically generated

1. Now that whole program is executed, so the whole global execution context will also get deleted.

Call Stack in JavaScript: we saw that for each time whenever a function is invoked a new execution context is created. What if there was a function invocation inside a function, so every time a new execution context was created, so for managing the whole creation and deletion, JavaScript uses a Stack, which is called as the Call Stack.

A blackboard with writing on it

Description automatically generated

Whenever we run a JavaScript code, the call stack is populated with a Global Execution Context, and whenever a new “local execution context” is created, it is pushed on the top of the stack, and as soon as its execution is over, it is popped out of the stack and control goes to the execution context which is at the top of the stack at that time. And at last, when all execution contexts are popped out, the control goes to the global execution context and after the whole main program is implemented, the global execution context is also popped out and the call stack becomes empty.

A close-up of words

Description automatically generated

Different names of call stack are as follows: A close-up of a list of text

Description automatically generated

***Hoisting in JavaScript***

Hoisting is a concept that enables us to extract values of variables and functions even before initializing/assigning value without getting errors and this happens during the 1st phase (memory creation phase) of the Execution Context.

* In JavaScript, hoisting is the default behaviour of moving all the declarations at the top of the scope before the code execution. Basically, it gives us an advantage that no mater where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local.
* It allows us to call functions before even writing them in our code.

Note: JavaScript only hoists declarations, not initializations.

JavaScript allocates memory for all variables and functions defined in the program before execution.

Let’s understand hoisting with some examples:

|  |  |  |
| --- | --- | --- |
| Input | Output | Explanation |
|  |  | When we see this code, it is clear that after the variable and function declaration, they will be allocated some memory, and after that when console log is called then value of “x” will print, and then function is called so whatever operations are there in the function will get executed. In this case “hey Shivani” is printed. |
|  |  | Now, in this case, we tried printing the value of “x” even before we declared it, and in the same way we are calling the function before its declaration and definition. So, if we consider any normal programming language scenario, this will throw an error. But in JavaScript, the output for value “x” will be undefined and “getName” function will execute normally and “hey Shivani” will be printed. Why is this happening? So, we have studied before that as soon as we run the JavaScript program, memories are allocated to the variables and functions at the Memory Creation Phase. Thus, the value for x is “undefined” and function somehow was invoked and performed the necessary operations. |
|  |  | Now in this case, we removed the declaration of variable “x”, so we can see, that, function has been invoked but there is a “Reference Error” which says that x is not defined. This is because, in the program we tried accessing a value “x” which was never initialized in whole code. |
|  |  | In this case, when we are trying to get the value of this function (remember we are not invoking the function), so here, the whole code for the function will be printed in the console. |
|  |  | In this example, we are having an arrow function. And it is called after the declaration of the function. So, output will be as expected. It will print 7 and hey Shivani. But what if we call the function before its declaration? |
|  |  | In this case, value for x will be undefined (we discussed before) and when getName is called, then there will be a “Type Error” which says that getName is not a function. This is because, at the time of Memory creation phase, getName was treated as a variable, and “undefined” value was assigned to it.  Conclusion: whenever a proper function is declared, it will be allocated the memory as a function, and whole function will be copied in the execution context’s memory component (variable environment) but if we are trying to declare arrow functions and want to access those before, then they will be treated as variables, and will be assigned “undefined” at the Memory Creation Phase. |

This is all about Hoisting in JavaScript.

***Call Stack Demo in JavaScript***

We understood while studying the basics that – “Call Stack maintains the order of execution of Execution Contexts”. And whenever a function is invoked, a new execution context is created, and it appears on the top of the Call Stack. As soon as the execution of function is over, its execution context is popped out of the Call Stack and control goes back to the previous Execution Context.

How the above concept looks like in Browser?

When we run the below program and put a debugger at line number 27 “var x = 7;”, we can see in the call stack that a global execution context has been created, by the name “(anonymous)”.

A computer screen with text

Description automatically generated

Putting the debugger:

A close-up of a math problem

Description automatically generated

Call stack:

A screenshot of a phone

Description automatically generated

This tells us that, control is now on line number 27 in index.js file.

Now, let’s try putting debugger on the function getName, this means we are trying to invoke the function, and we are expecting that a brand-new execution context will be created, let’s see what’s happening:

A computer code with text

Description automatically generated

Call stack will look like:

A white rectangular object with a black border

Description automatically generated with medium confidence

This blue arrow shows that control is now on the function getName. This simply implies that a new execution context was created when the function was invoked, and it appeared onto the top of the stack. And as soon as the debugger is put on another line, this execution context will get deleted. And when the program is over, call stack will become empty.

***How functions work in JavaScript?***

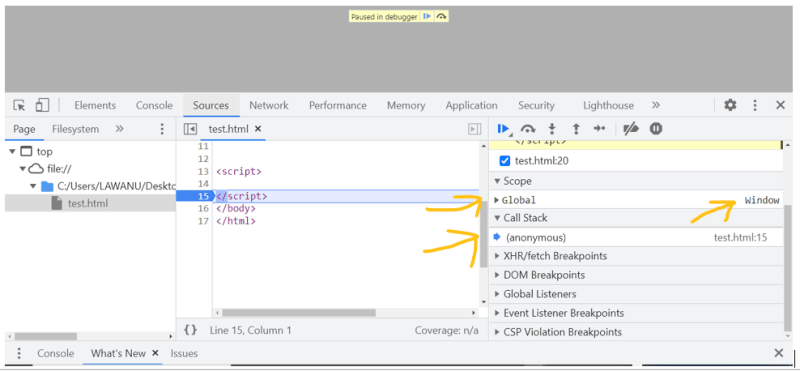
A blackboard with colorful writing

Description automatically generated

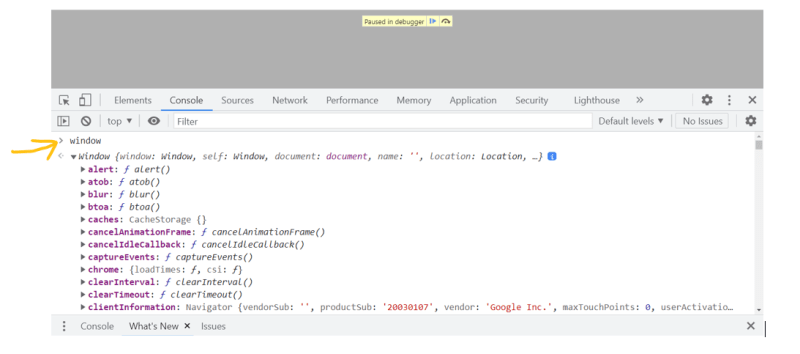
***Shortest JavaScript Program***

The shortest JavaScript program is an empty program. When we run an empty JavaScript code, a global execution context is created. The JS engine

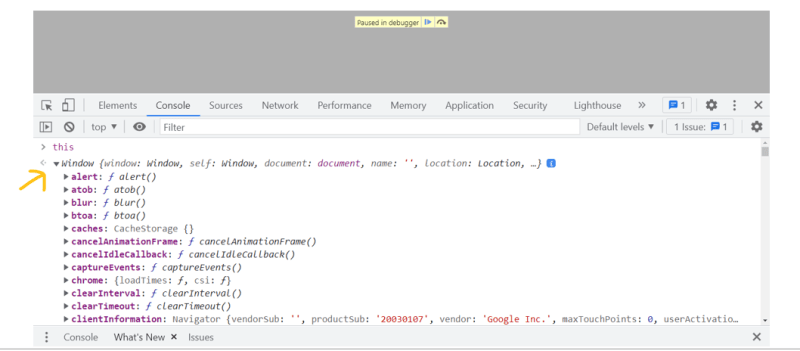
sets up the global execution context and a global memory space even though there is no code. In addition to that JS engine do something interesting, it also creates window object.



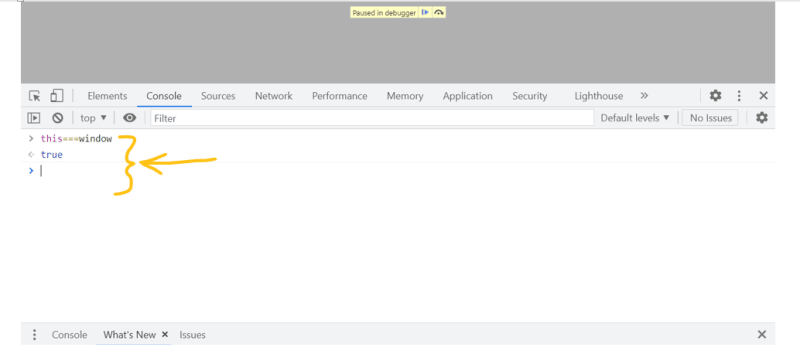
This window object is created by JS Engine which has many functions and variables. These are created in global memory space. So, we can access these variables and functions anywhere in JavaScript program.



In addition to that JS Engine will also create “this” variable. At the global level “this” points to “window” object.



Window is a global object which is created along with the global execution context. Whenever any JS program runs, a global execution context is created, window object and along with the global execution context a “this” variable is created.



Any variable or function we create in a global scope, gets attached to the “window” object and we can access it by the reference of “window” object.

var a = 10;

console.log(window.a); // 10

console.log(a); // 10

function b() {

    var x = 100;

    console.log(x);

}

console.log(x); // x is not defined - because it is in function scope

console.log(b.x); // undefined

***Undefined vs. not defined in JavaScript***

**undefined**: undefined is a special keyword, which is used when we are allocating a memory to any variable in JavaScript. It acts as a placeholder, which is attached to the variable, just to reserve its memory in the Execution Context, until we assign any value to the variable.

// example 1

console.log(a); // undefined

var a = 10;

We know that, whenever we run a JavaScript code, memory is allocated to all variables and function before the execution of the program, and there the special keyword “undefined” is attached with the variables.

// example 2

var a;

console.log(a); // undefined

***JavaScript is a loosely typed language or weakly typed language***

There is no such data type in JavaScript, we can use it in any type we want. See the example below:

var a;

console.log(a);  // undefined

a = 10;

console.log(a);  // 10

a = "hello shivani";

console.log(a);  // hello shivani

JavaScript is that flexible language, that it allows to change the data type, whenever and however we want.

***Never do this mistake***

var a = undefined;

It is not a good practice to do so, because “undefined” has its own purpose (to check whether any variable has been assigned with any value or not).

**not defined**: when we haven’t defined any variable or function, but we’re trying to access it somehow, so JavaScript will throw an error, that the variable is not defined.

A computer screen shot of a code

Description automatically generated

A screen shot of a message

Description automatically generated

***The Scope Chain, Scope and Lexical Environment***

“Scope in JavaScript is directly related to the Lexical Environment”.

Let’s see some examples:

// example 1

function a(){

    console.log(b);

}

var b = 100;

a();  // it will print 100

// example 1

function a(){

    console.log(b);

}

a();  // it will print undefined

var b = 100;

// example 2

function a(){

    c();

    function c() {

        console.log(b);

    }

}

a();  // it will print undefined

var b = 100;

// example 2

function a(){

    c();

    function c() {

        console.log(b);

    }

}

var b = 100;

a();  // it will print 100

In the above examples, when we invoke a function, it is somehow able to fetch the value of “b” from the global memory space. Let’s see an example below:

// example 3

function a(){

    var b = 100;

    c();

    function c() {

        console.log(b);

    }

}

a();  // it will print 100

console.log(b);  // Reference Error: b is not defined

In this example, when we have defined the variable inside a function and we are trying to access it in global part, then it is throwing a reference error, which says that b is not defined.

Here comes Scope into the picture.

**Definition of Scope**: *A space where you can access a specific variable or a function in the code.*

There are two aspects or views we can consider while looking for scope of a variable or function:

1. Is variable/function being inside the scope (of any function)?
2. What is the scope of a particular variable/function?

***What is Lexical Environment?***

Let’s understand this with a visual representation:

A blackboard with writing on it

Description automatically generated

When we run above program, a global execution context will be created, and memory to function a will be allocated, after that we see a function invocation for a, so a new execution context for a will be created, and all the variables and functions inside a will be allocated a memory, after that we come through another function c, which is inside a, a new execution context for that will also be created. Since we don’t have any variable or function inside c, so leave the execution context empty.

Now comes the Lexical Environment.

*Whenever an Execution Context is created, a Lexical Environment is also created. Lexical Environment is the local memory, along with the Lexical Environment of its parent.*

The literal meaning of “Lexical” is “hierarchy or sequence”. In the terms of JavaScript, keeping the above code in mind, we can say that “function c is *lexically sitting* inside function a”.

Now what is the Lexical Environment of the parent, so whenever an Execution Context is created, along with the memory allocation to variables and functions, we get one more thing, that is the reference to the “Lexical Environment of the Parent”, that means it will point its parent (which is its parent in the physical code, here we can see that a is the parent of c and a’s parent is the global execution context). We can understand this with below diagram:

A blackboard with writing on it

Description automatically generated

Here, c is pointing to the lexical environment of a, and a is pointing to its lexical environment, i.e., global execution context, and since the global execution context does not have any parent, so it is pointing to null.

How Lexical Environment works?

Let’s say we have a piece of code as below:

A screen shot of a computer program

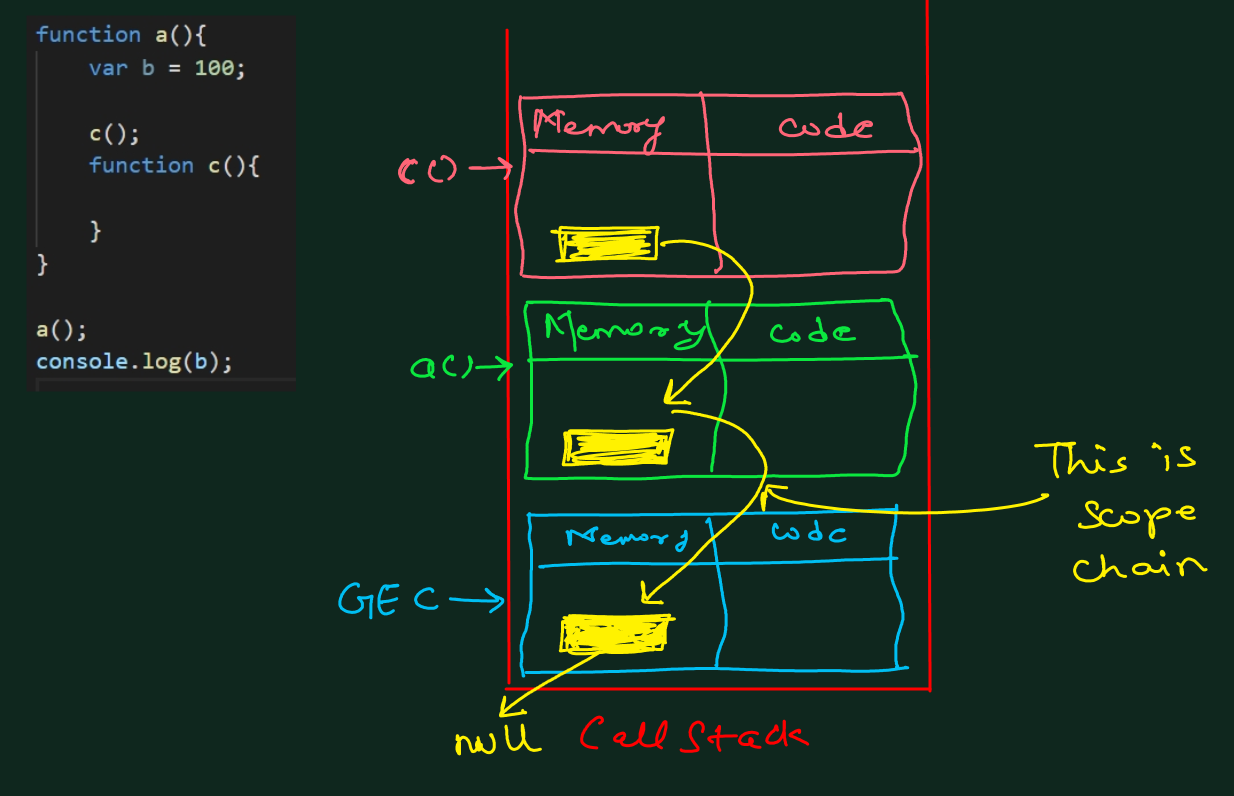
Description automatically generated

Now when the function c is invoked, it is trying to print the value of b, so firstly it will look into its own local memory space, whether the variable b is present or not. Since b is not present in the local memory of function c, so it will try watching out for b in the Lexical Environment of its parent, i.e., function a, and there we found variable b with value 100, so since c has the reference to the lexical environment of a, so it will now print 100.

***What if we didn’t find variable b in lexical environment of a?***

In that case, a will look out for b in its parent’s lexical environment, i.e., in the global execution context’s memory space. If b will be found there, then it will be printed, otherwise there will be a reference error, that b is not defined.

***The Scope Chain:*** The scope chain is nothing, but what we did above, mechanism of finding the variable in different lexical environments. The chain of the lexical environments with the parent references is the Scope Chain.



***let and const in JavaScript***

let and const were introduced in ES6 version of JavaScript. “let” allows reassignment of values to it, but “const” creates constant variables that cannot be reassigned another value.

let and const are hoisted, but in a very different manner as compared to the var.

***Interview tip:***

**Ques**. Are let and const hoisted?

**Ans:** Yes, let and const are hoisted, but they are in temporal dead zone for the time being.

|  |  |  |
| --- | --- | --- |
| Input | Output | Explanation |
|  |  | In this case, program flow will be as expected, we declared two variables with different values and tried to print those, and we got the expected output. |
|  |  | We have seen before, that in case of var, whenever and wherever we try accessing it, we get two values either undefined (in case, where we haven’t initialized it) or the value assigned to it. So here also, we will get undefined. But let works differently. It is hoisted, but not in the global memory space, so we cannot use it before its initialization.  The second ss shows that “a” is hoisted, as it has been assigned “undefined”, but since it is not in the global memory space, we cannot use it before it is assigned with some value. |

***Temporal Dead Zone:*** *The time since when the let variable was hoisted and till it is initialized with some value.*

***ReferenceError:*** the ReferenceError object represents an error when a variable that does not exist (or hasn’t yet been initialized) in the current scope is referenced.

|  |  |  |
| --- | --- | --- |
| Input | Output | Explanation |
|  |  | This clearly states that we cannot access a before a value is assigned to it. This means that a is in temporal dead zone for now. |
|  |  | When we try to access a variable which is not there in the current scope, then we get a reference error stating that the particular variable was not defined anywhere in the scope. |

***Important difference between let and const***

const declaration is even more strict than the let declaration. In the hoisting part, they behave the same way, const also takes up memory in the memory space other than global memory space and also goes through the temporal dead zone.

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

In case of let, we can declare it and can initialize it later in the program, but this is not the case with const. We need to declare and initialize the const variable right away, otherwise it will throw a SyntaxError.

***SyntaxError:*** A SyntaxError is a type of error that is thrown when there is a typo in the code, creating invalid code – code which cannot be interpreted by JS Engine. If there is a SyntaxError in the program, then the program does not run, it is just rejected upfront.

|  |  |  |
| --- | --- | --- |
| Input | Output | Explanation |
|  |  | For the const variable declaration, we need to initialize it on the same time while declaring it. |
|  |  | No duplicate values are allowed while using let declarations. |

***TypeError:*** A TypeError object represents an error when an operation could not be performed, typically (but not exclusively) when a value is not of the expected type. It may be thrown when: an operand or argument passed to a function is incompatible with the type expected by that operator or function.

|  |  |  |
| --- | --- | --- |
| Input | Output | Explanation |
|  |  | We cannot reassign a new value to const variable. And when we try to do so, it throws a type error stating that we are trying to assign a value to const variable afterwards. |

***How to avoid Temporal Dead Zone?***

For this, we can follow a simple tip, that all the declarations and initializations must be done at the top of the code, and then we can dive into the logic part. Moving all the declarations and initializations on the top will help the JS Engine to not to go in the temporal dead zone, and we will be able to minimize the time window of the temporal dead zone.