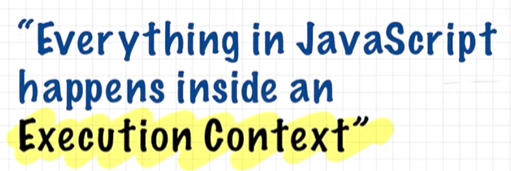
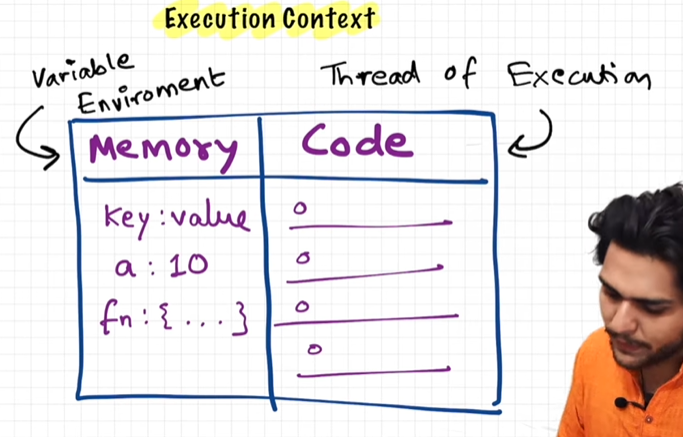
***Basics of JavaScript***



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?***

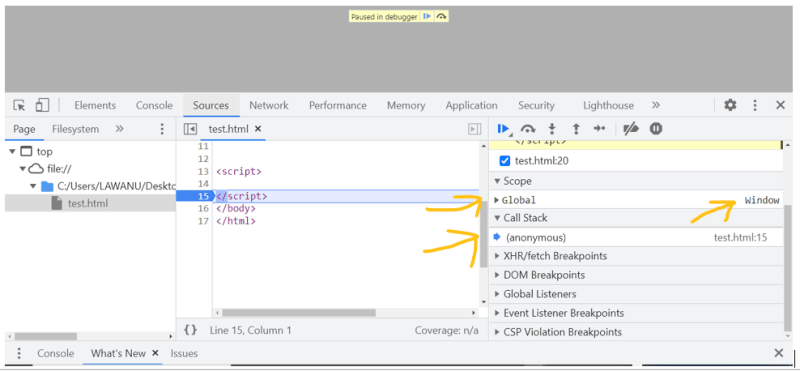
A blackboard with colorful writing

Description automatically generated

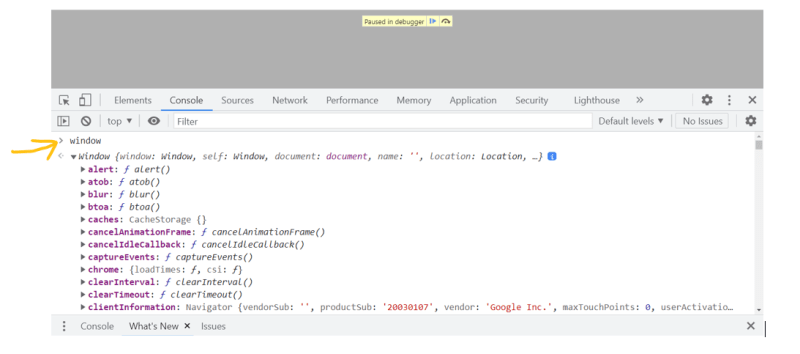
***Shortest JavaScript Program***

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

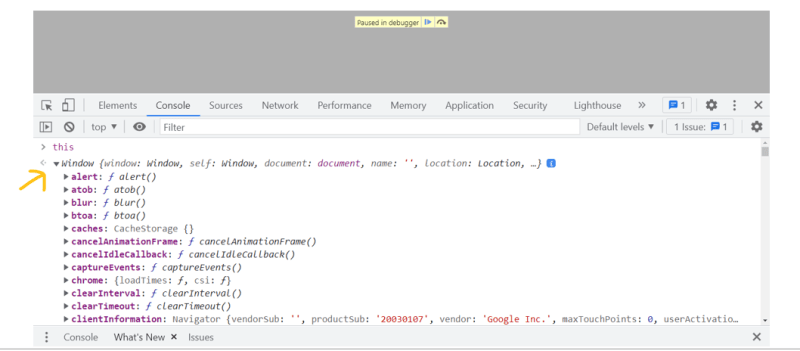
sets up the global execution context and a global memory space even though there is no code. In addition to that JS engine do something interesting, it also creates window object.



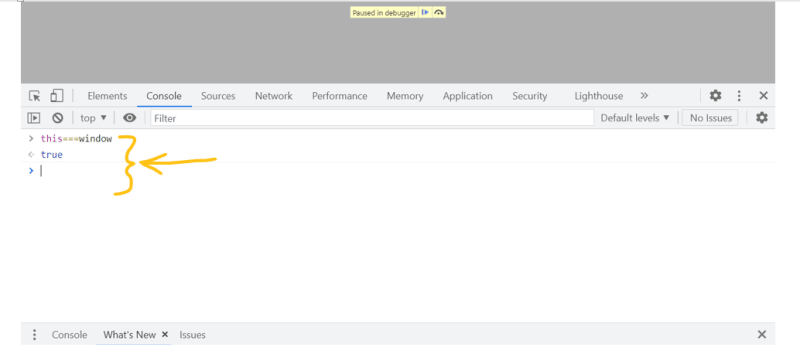
This window object is created by JS Engine which has many functions and variables. These are created in global memory space. So, we can access these variables and functions anywhere in JavaScript program.



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

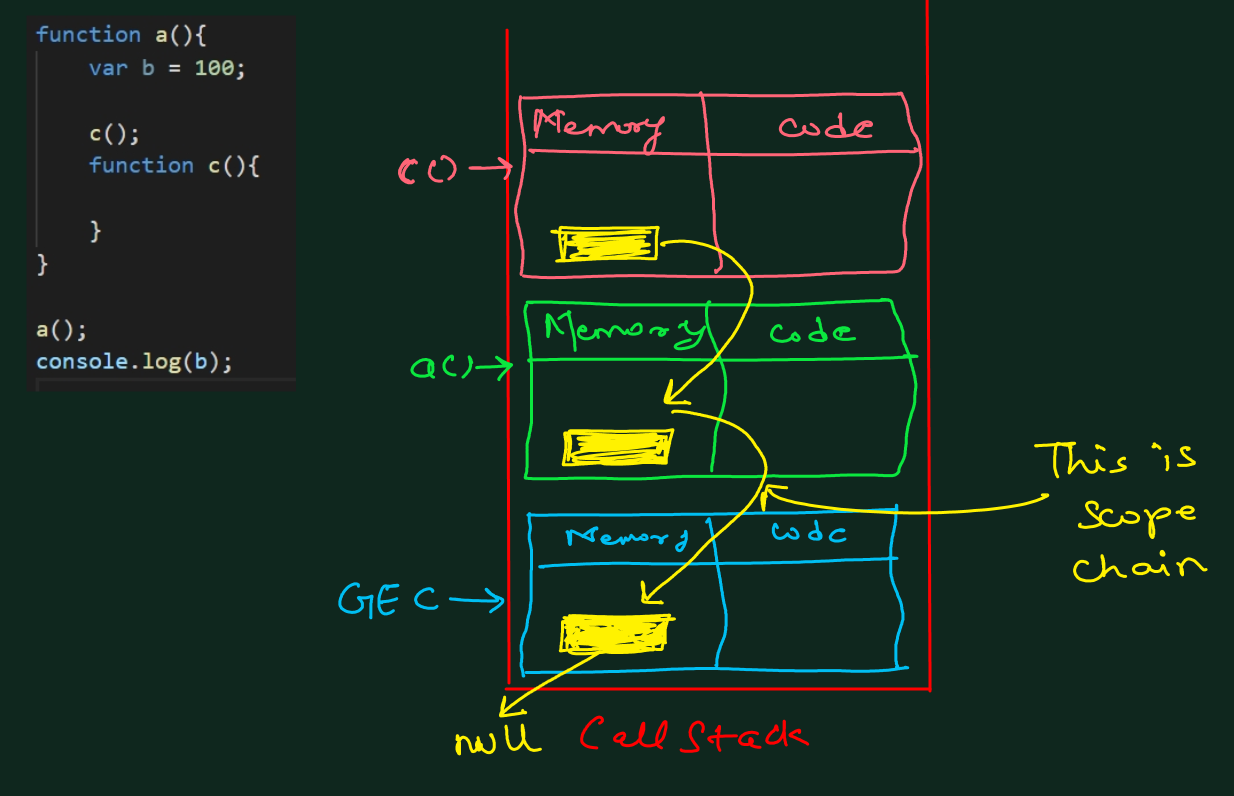
Description automatically generated

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.

***What is a Block in JavaScript?***

Block {} is used to group JavaScript statements together into one group so that it can be used anywhere in the program, where only one statement is expected to be written.

Example:

A screen shot of a computer

Description automatically generated

Above is a valid JavaScript code, where we have used block {} to combine multiple lines of code together in a single entity.

Now if we see, we have multiple things available like if else, for loop, while loop, etc. and their syntax is like below:

A black background with white text

Description automatically generated

And we want that rather than a single operation, multiple operations should be performed. So this operation is replaced by a block as below:

A computer screen shot of a code

Description automatically generated

***Block scope in JavaScript***

Block scope means, what all variables and functions we can access inside a particular block.

**Example:**

A screen shot of a computer code

Description automatically generated

**Output:**

A screenshot of a computer code

Description automatically generated

Let’s see the scope of these variables in browser:

A screenshot of a computer

Description automatically generated

The let and const variables are in the “Block Scope” and var variable is in the “Global Scope”. Thus, it is said that “let and const are block scoped”. And as soon as we move outside the block, this block scope will disappear, and we will not be able to access b and c variables.

A screenshot of a computer program

Description automatically generated

***Shadowing in JavaScript***

Now, when a variable is declared in a certain scope having the same name defined on its outer scope and when we call the variable from inner scope, the value assigned to the variable in inner scope is the value that will be stored in the variable in the memory space. This is known as Shadowing or Variable Shadowing.

In JavaScript, the introduction of let and const in ES6 along with block scoping allows variable shadowing.

// shadowing in javascript - variable example

var a = 10;

let b = 20;

const c = 30;

{

    var a = 40;     // new value assigned

    let b = 50;     // will take value from this scope only

    const c = 60;   // will take value from this scope only

    console.log(a);

    console.log(b);

    console.log(c);

}

console.log(a);     // shadowing occured, value from block scope will be taken

console.log(b);     // value taken from scope other than block scope

console.log(c);     // value taken from scope other than block scope

Output:

A screenshot of a computer

Description automatically generated

// shadowing in javascript - function example

function func() {

    let a = 'Geeks';

    if (true) {

        let a = 'GeeksforGeeks';  // New value assigned

        console.log(a);

    }

    console.log(a);

}

func();

Output:

A screenshot of a computer

Description automatically generated

Illegal Shadowing

Now, while shadowing a variable, it should not cross the boundary of the scope, i.e., we can shadow var variable by let variable but vice versa is not possible. So, if we try to shadow let variable by var, it is known as Illegal Shadowing, and this will throw an error that the “variable is already defined”.

A computer screen with text and numbers

Description automatically generated

Output:

A screenshot of a computer error

Description automatically generated

A computer screen shot of a computer program

Description automatically generated

Output:

A screen shot of a computer error

Description automatically generated

*Block scope also follows the lexical scope chain concept. When one block is defined inside another one, they behave as scope chain we studied before.*

*Arrow functions will also have the same scope rules as normal functions.*

***Closures in JavaScript***

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment).

Example:

A computer screen with text and images

Description automatically generated

In the above example we know that when function a will be invoked, then it will allocate memory to the variable x and function b, and moving forward inside this function, another function b will be invoked, so it will try to look for x in its own local memory space and when it cannot find it there, then it will check for x in its lexical environment (reference to the parent), and finally the value for x will print.

Output:

A screenshot of a computer

Description automatically generated

The above example is the closure. We can check it in the browser:

A screenshot of a computer

Description automatically generated

*Closure basically means that a function bundled or bind together with its lexical scope/environment. The function b was bind together with the variables of function a, thus a closure was formed.*

We can do multiple things with functions:

1. We can pass a function as an argument of another function.

A computer screen with text on it

Description automatically generated

1. We can return a function.

A computer screen with text on it

Description automatically generated

1. We can also define a function using a variable.

A computer screen with white and blue text

Description automatically generated

Now let’s understand closure using this a bit complex example:

function a(){

    var x = 10;

    function b(){

        console.log(x);

    }

    return b;   // inner func returned along with its lexical environment

}

var c = a();    // outer function called which is returning the inner function

console.log(c); // whole inner function will be printed

c();    // since, when the inner function was returned, it came with its lexical environment, so it will implement the logic and will print value of x i.e., 10

Output:

A screenshot of a computer

Description automatically generated

*Closure is created when a child function keep the environment of the parent scope even after the parent function has already executed.*

Note: Closure is the concept of function + lexical environment in which function it was created. So, every function declared within another function then it has access to the scope chain of the outer function and the variables created within the scope of outer function will not get destroyed.

Example 2:

function foo(outer\_arg) {

    function inner(inner\_arg) {

        return outer\_arg + inner\_arg;

    }

    return inner;

}

let get\_func\_inner = foo(5);    // argument for outer function

console.log(get\_func\_inner(4)); // this will act as argument for inner function => 5+4 = 9

console.log(get\_func\_inner(3)); // this will act as argument for inner function => 5+3 = 8

In the above example we used a parameter function rather than a default function. Not even when we are done with the execution of foo (5) we can access the outer\_arg variable from the inner function (because the closure remembers the scope chain). And on the execution of the inner function produce the summation of outer\_arg and inner\_arg as expected.

***Uses of Closures:***

1. Module design pattern
2. Currying
3. Functions like once
4. Memoize
5. Maintaining state in async world
6. setTimeouts
7. iterators
8. and many more…

Closure with setTimeout

setTimeout: The setTimeout() method sets a timer which executes a function or specified piece of code once the timer expires.

Example:

// setTimeout and closure

function a(){

    var x = 1;

    setTimeout(function(){

        console.log(x);

    }, 5000); // this console log will be executing after 5 seconds.

}

a();

Output:

A close-up of a number

Description automatically generated

This is a common mistake that we think that this setTimeout is executed here, like line by line, but what if we added a loc below setTimeout:

function a(){

    var x = 1;

    setTimeout(function(){

        console.log(x);

    }, 5000); // this console log will be executing after 5 seconds.

    console.log("hi shivani");

}

a();

So, in this case, according to me, there should be a wait of 5 seconds and then 1 must print and after that Hi Shivani must be printed. But that is not the case here. Here, “JavaScript does not wait for anyone to first finish executing”. So, hi Shivani will be printed first and then when the timer of 5 seconds will expire, then 1 will be printed.

A screenshot of a computer

Description automatically generated

What setTimeout does is, the callback function in the method forms a closure and it remembers the reference of the variable, with whom it has formed closure (in above example, with variable x). Here, the setTimeout takes the callback function and stores it at some place and attaches a timer with that, and JavaScript moves forward with the code. And once the timer expires, the function or piece of code written in setTimeout is taken and is put into the call stack and then it is executed.

***Interview question:***

You need to print numbers from 1 to 5 and each number should be printed after waiting for same number of seconds, like 1 should be printed after 1 second, 2 after 2 seconds, 3 after 3 seconds, and so on.

**Solution:**

A simple solution that comes to our mind is that we will simply run a loop from 1 to 5 and declare a setTimeout function inside that and will log value of “i” and for the timer we will just multiply 1 second (1000 milliseconds) with “i”. And we are done.

A computer screen with text

Description automatically generated

**Output:**

A screenshot of a computer

Description automatically generated

But what the output says, is very different, it prints 6 for 5 times.

**Reason:** this is because of closures. Closure is basically a function with its lexical scope/environment. So even the function is taken out from its original scope, still it remembers the “reference” to its lexical environment. What happened here is, the callback function in the setTimeout is when stored somewhere else, so it remembered the “reference to i”. So, the reference to i was remembered, not the value of i. All the five copies which were created, are referring to “i”. Also, JavaScript will not wait till the timer expires. It will quickly create 5 copies of the functions and all will be referring to the same “i”. And we know that, when the loop ended, the value of i became 6, thus, every time 6 is printed on the console.

So, now what is the fix of this problem? A very simple solution is to use “let” instead of “var” because we know that let is block scoped, so every time the loop will run, a new whole new scope is created for a new value of i and all the five closures are attached to a different “i”.

A computer screen shot of text

Description automatically generated

A screenshot of a cell phone

Description automatically generated

***Extension of above interview problem:***

Interviewer might say that we cannot use “let”, and we need to solve it using “var” only. So, here, closures will help us.

Reason why closures will help: because we know that we want to provide a new copy of “i” to the setTimeout, so that the closure is created with a new copy of “i” every time.

Solution:

function a(){

    for(let i=1; i<=5; i++){

        function closure(newValOfi){

            setTimeout(function(){

                console.log(newValOfi);

            }, newValOfi\*1000);

        }

        closure(i);

    }

    console.log("hi");

}

a();

Output:

A screenshot of a computer

Description automatically generated

Using the closure function, we tried to solve the problem, and every time a new value of “i” was provided to the setTimeout method.

Interview Questions on Closures

Functions

Terminologies:

1. Function Statement: the most popular way of creating any function.

A black screen with white text

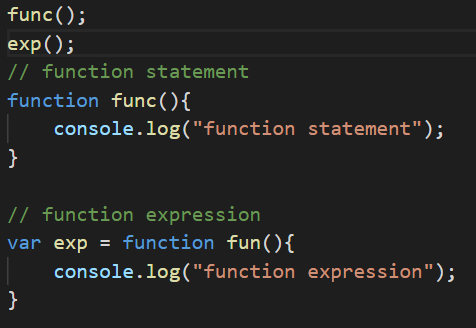
Description automatically generated

1. Function Expression: when we assign a function to a variable, and function here is acting like a value, this is known as Function Expression.

A black screen with blue and white text

Description automatically generated

Difference between function statement and function expression: the main difference between function statement and function expression is of HOISTING, the way they are hoisted. Let’s see below example:



Output:

A screenshot of a error message

Description automatically generated

Explanation:

During the memory creation phase, memory is allocated to function statement (func) and “exp” is treated as a variable, and is assigned undefined initially and then when code executes, it is assigned the function fun() to it. So, when we are trying to use exp as function before function is assigned to the exp, so it is throwing an error.

1. Function Declaration: It is a synonym of Function Statement.
2. Anonymous Function: Function without a name is called as anonymous function. These do not have their own identity.

A black background with white text

Description automatically generated

A close-up of a sign

Description automatically generated

This is because, whenever we declare a function, we need to provide a name to the function. So, why we are having the concept of Anonymous functions. The answer is, we use these anonymous “functions where functions are used as values”.

We can use these to assign them to a variable. Example below:

A black screen with white text

Description automatically generated

1. Named function expression: when we give a name to the function while assigning it as value to a variable.

A black background with blue and white text

Description automatically generated

Corner case:

|  |  |  |
| --- | --- | --- |
| Input | Output | Explanation |
|  |  | This is a normal way of calling the named function expression, it will simply perform the operations specified in the function. |
|  |  | Now that we try to access the function that is assigned to the variable directly, then it will throw an error, because JavaScript parses it as a variable, thus error is “function\_name is not defined”, because this function is not present in the global scope. |
|  |  | And if we want to access the function that is assigned to a variable, wo we can access it inside itself. |

1. Parameters and Arguments: parameters are passed while declaring a function, while arguments are passed while calling a function.

A black screen with white text

Description automatically generated

1. First Class functions: the ability of functions to be used as value, can be passed as argument to another function or returning a function from a function, is known as First Class function.

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

1. First Class Citizens: ability of functions to be used like values, passed like arguments into other functions or returned by a function makes them First Class Citizens. In other words, First Class functions are the First Class Citizens.

***Callback functions in JavaScript***

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

Callback functions give us access to the asynchronous things in “Synchronous single-threaded language”.

These functions which are passed as arguments to another function, are called as callback functions because, we give the responsibility to the other function to decide, when these functions (which are passed as arguments) will be called back.

***Examples:***

A screen shot of a computer program

Description automatically generated

Here, y is a callback function.

How are these callback functions used for achieving asynchronous behaviour in JavaScript?

A computer screen shot of text

Description automatically generated

setTimeout will take the callback function and will attach a timer of 5 seconds to it and store it somewhere. And since we know that JavaScript will not wait, and will execute its flow further, so function x will be called. And following will be output:

A screenshot of a computer

Description automatically generated

***Main thread:*** JavaScript has only one call stack, which is known as the main thread. Whatever is executed on the page, it is done through the call stack.

***Blocking main thread:*** any operation which takes more time to execute or blocks the call stack, is called as Blocking of main thread. Let’s say we are having a function, which is taking so much time to execute, so by that time the call stack will be blocked. So, it is recommended to use async things where we know that operations are going to take a lot of time. This will help us to get rid of the problem of main thread blocking. For example, we can use setTimeout, etc.

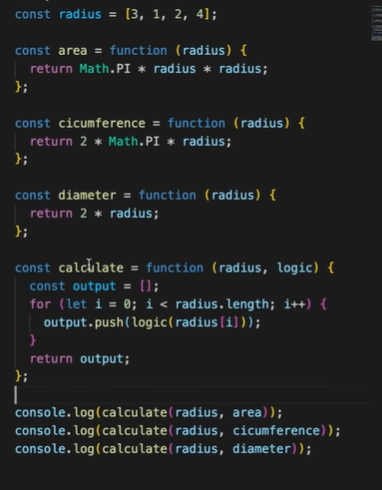
Event listeners

Asynchronous JavaScript and Event Loop

Trust issues with setTimeout

***Higher order functions***

Higher order functions are the functions that take one or more functions as arguments or return a function as their result.



Polyfill for map:

map, filter and reduce