Priya Pandey · Follow
Jul 27, 2021 · 7 min read

JavaScript Execution Context And Hoisting

I will cover What is context, What is a thread of execution, Execution context creation, and its 2 phases: The memory Allocation Phase the Code Execution phase. Is javaScript synchronous or asynchronous? Is JavaScript a single-threaded language or multi-threaded? What is hoisting in JS and how its works?

What is the context?

Context is Explain why the weirdness of Javascript is going on.

The Whole idea of context is to get information collected. context means the environment in which we are Executing the code.

Example:

```
var num = 2;

function sayMe(){
console.log("sayMe");
}

SayMe();
```

// one line of Execution context will kick in and goes out. but, when we call a function sayME() that means the entire function will be Executed. that means the Execution function can come which will be responsible for running a function.

So **Where does the JavaScript code get Executed?**

first things JavaScript Engine do as soon as we Executed a code snippet the JavaScript engine create a ***Global Execution context***. so Global Execution context can only get

Global Execution context contains a specific section is used for storing data also known as memory.

What is Execution context? ✋

Execution context is something that you need to know to understand how JavaScript code runs. The Environment in which our code is Executed and is evaluated. Global Execution context is by default.JavaScript engine creates a Global Execution context before it starts to Execute code.

Everything in JavaScript happens inside an execution context.

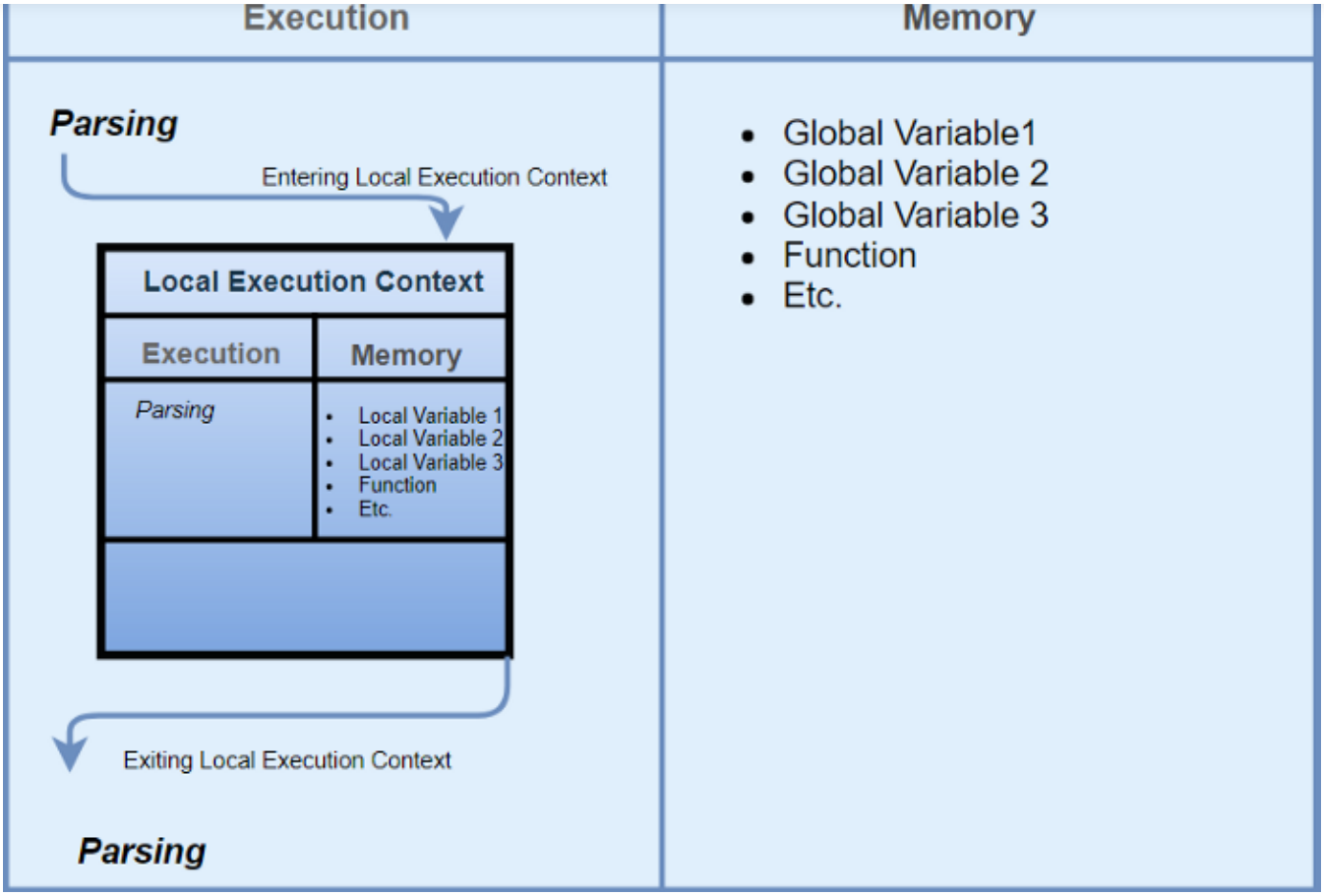
Execution context happens for every line of code, not only for a function its form top to bottom there will come and goes away. considered this as a memory stack, there as soon as the memory is needed it keeps on stacking of each other and then they go away.

There is two types of execution context

one gets created only onces throughout the program the other one is gets created whenever you are executing any function.

we can assume the execution context in javaScript is a big-box or a container in which whole JavaScript code is executed.





> *Parsing meaning:*
>
> *Parsing means analyzing and converting a program into an internal format that a runtime environment can actually run, for example, the JavaScript engine inside browsers.*

Let's walk through the **image**. When the JS engine starts reading your code, it creates the global execution context. It starts parsing line by line and adds your variables to memory also known as global variable environment.

 # The Difference between Global and Function context and what point they get created.

The **Global Execution Context** is the first execution context that's get created by the JavaScript engine whenever it is running your code. and it gets created only once when the JavaScript engine is running your code. after it's created it goes through each line of code one after another and execute the code.

To sum up briefly:

1. Global execution context is created first.
2. Whenever a function gets invoked, called, or executed, a new local execution context gets created.
3. The JavaScript engine starts parsing the code in the global execution context.
4. When the engine comes across a function call, it enters a local execution context and continues parsing in there.
5. When it handles function execution, it exits from that local execution context, returns to the previous one.

Phases of the Execution context.

Different phases of Execution context

Every Execution context goes through two phases.

1- Declaration phase: In the declaration phase engine only look for Declaration.

Two types of Declaration phase :

1-Function Declaration. function Declaration is scanned code and made available so we were able to call the definition first then we were having the definition of the function of what we gonna do.it was all possible because of The ***Global Execution Context.***

2-Variable Declaration — variable Declaration also Scanned the code and made it **undefined.**

#What is the Difference between Variable Declaration Function Declaration?

In the declaration phase, the variable Declaration doesn't know what value it holds.it has because it's only getting initialized with undefined so, it doesn't know what value it holds.



2-Execution phase — During the execution phase, the JavaScript engine executes the code line by line. In this phase, it assigns values to variables and executes the function calls.

For every function call, the JavaScript engine creates a new **Function Execution Context**. The Function Execution Context is similar to the Global Execution Context, but instead of creating the global object, it creates the `arguments` an object that contains a reference to all the parameters passed into the function.

> ***Note:***
>
> *how the function was executed completely, but we have* ***undefined*** *for the variable. This is because* **Hoisting** *is performed differently for functions vs variables. The function as a whole is picked up into the memory, but for the variables, space is reserved as a* placeholder *with the value of* ***undefined****. The actual value is then replaced when the engine executes* your *code line-by-line.*

What is Hoisting in JavaScript?

I already write this thing on top but to understanding hoisting, I go through the short intro to all of the things.

When you execute a piece of JavaScript code, the JavaScript engine creates the ***Global Execution Context.***

> *The* **Global Execution Context.** *has two phases: creation and execution.During the creation phase, the JavaScript engine moves the variable and function declarations to the top of your code. This feature is known as* ***hoisting*** *in JavaScript.*

undefined vs ReferenceError

Before we begin in earnest, let's deliberate on a few things.

```
console.log(typeof variable); // Output: undefined
```



Our second point is:

```
console.log(variable); // Output: ReferenceError: variable is not
defined
```

> In JavaScript, a ReferenceError is thrown when trying to access a previously undeclared variable.

The behavior of JavaScript when handling variables becomes nuanced because of hoisting. We'll look at this in-depth in subsequent sections.

Variable hoisting

Variable hoisting means the JavaScript engine moves the variable declarations to the top of the script. The following example declares the `counter` variable and sets its value to 1:

```
console.log(counter); // undefined
var counter = 1;

Code language: JavaScript (javascript)
```

However, the first line of code doesn't cause an error because the JavaScript engine moves the variable declaration to the top of the script. Technically, the code looks like the following in the execution phase:

```
var counter;

console.log(counter); // undefined
counter = 1;

Code language: JavaScript (javascript)
```



undefined.

Let's cover some important points from MDN

Only declarations are hoisted

- JavaScript only hoists declarations, not the initialization. If a variable is used in code and then declared and initialized, the value when it is used will be its default initialization (`undefined` for a variable declared using, otherwise uninitialized). For example:

```
console.log(num); // Returns 'undefined' from hoisted var
declaration (not 6)
var num; // Declaration
num = 6; // Initialization
```

The example below only has initialization. No hoisting happens so trying to read the variable results in `ReferenceError` an exception.

```
console.log(num); // Throws ReferenceError exception
num = 6; // Initialization
```

Variables declared with `let` and `const` are also hoisted, but unlike for `var` the variables are not initialized with a default value of `undefined` . Until the line in which they are initialized is executed, any code that accesses these variables will throw an exception.

Hoisting var variables:

Let's look at some examples to understand hoisting in case of `var` variables.



Remember JavaScript only hoist declarations, not initializations. That is, during compile time, JavaScript only stores function and variable declarations in the memory, not their assignments (value).

But why `undefined` ?

When JavaScript engine finds a `var` variable declaration during the compile phase, it will add that variable to the lexical environment and initialize it with `undefined` and later during the execution when it reaches the line where the actual assignment is done in the code, it will assign that value to the variable.Variables and constants declared with let or const are not hoisted!

JavaScript Initializations are Not Hoisted

Hoisting let and const variables:

Let's first take a look at some examples:

```
console.log(a);
let a = 3;
```

Output:

```
ReferenceError: a is not defined
```

So are `let` and `const` variables not hoisted?

The answer is a bit more complicated than that. All declarations (function, var, let, const) are hoisted in JavaScript, while the `var` declarations are initialized with `undefined` , but `let` and `const` declarations remain uninitialized.

They will only get initialized when their assignment is evaluated during runtime by the JavaScript engine. This means you can't access the variable before the engine evaluates

If the JavaScript engine still can't find the value of `let` or `const` variables at the line where they were declared, it will assign them the value of `undefined` or return an error (in case of `const`).

> ***Conclusion***
>
> *So now we know that during hoisting our code is not physically moved by the JavaScript engine. Having a proper understanding of the hoisting mechanism will help you avoid any future bugs and confusion arising due to hoisting. To avoid possible side effects of hoisting like undefined variables or reference error, always try to declare the variables at the top of their respective scopes and also always try to initialize variables when you declare them.*

🔚.

That's it Thankyou for Reading my article. if you found this article helpful, please click the clap 👏button below 😃.