

ADDIS ABABA INSTITUTE OF TECHNOLOGY

CENTER OF INFORMATION TECHNOLOGY AND SCIENTIFIC COMPUTING

DEPARTMENT OF INFORMATION TECHNOLOGY AND SCIENTIFIC COMPUTING

Internet and Website

Submitted by: - Dan Mekonnen

Student Id: - ATR/8274/11

Section: IT

Submitted to: - Fitsum Alemu

January 2021

Contents

[1. Is JavaScript Interpreted Language in its entirety? 3](#_Toc62321623)

[2. The history of “typeof null” 3](#_Toc62321624)

# Is JavaScript Interpreted Language in its entirety?

To answer that we have to first consider what the meaning behind the words is, or how one language can be consider an interpreted language or a compiled one. Before that we have to know what, they mean.

what is an interpreted language?

A language is consider interpreted when it converts one line of code into machine language at a time not being in need of a compiler to convert the content into machine code before being executed. It achieves this by using an interpreter running over a virtual machine, executing the code line by line.

What if it is a compiled one?

Compiled languages use a compiler to compile the language to a machine code before being executed themselves. This makes it that when a language is trying to be executed by the machine, it gets the full translated version of the programming language.

To put it roughly as an example, when a programming language is an interpreted one its like when a person is speaking in Russian or some other language you don’t understand and its being translated to you in every pause he takes, processing every word as you go. When it’s a compiled language it is the equivalent of buying the translated version of a book or novel to your language and you can read it continuously as one.

What if we combine the two?

So, is JavaScript Interpreted in its entirety?

No. Maybe many years ago yes but with modern browser engine which use engines like V8 translate the code into machine code, dropping the interpreter. It achieves such a task by changing the JavaScript code into machine code at the execution by implementing a JIT compiler. So, in simpler terms JavaScript might be an interpreted language but it gets compiled, for the machine we run the language on to understand the code or set of commands we gave it, it needs it to be translated. Hence the reason for it to be compiled.

## The history of “typeof null”

The History behind the confusion of the typeof null is that when JavaScript reads or trys to tell what type null is it see that it is an object and not NULL itself. To understand the reason behind why typeof null is an object we first have to know what typeof does.

The typeof operand is used to see what data type something is or to know the data type of a certain operand. So, if one chooses to see the data type of 123 just write



Which will return the data type of 123 as a string to you, being that 123 is a number so the data type string returned to you will be:



Accordingly, what do we get when we prompt or ask for the data type of null, it might come to a surprise to you but in JavaScript the data type of null in not going to show you a string saying it’s NULL, but tells you that null is actually an object.

So, how is the typeof null data type an object. To answer that question, we have to go to the beginning of JavaScript, in the first version of JavaScript. In this version JavaScript stored values in a 32-bit unit, and gave 1-3 bits for the type tag and were stored on the first of the bits. So, JavaScript would use these bits to determine what the data type was of the value. There were 5 of these type tags ranging from object to Boolean.

* 000: object. Data type object.
* 1: int. The data is a 31-bit signed integer.
* 010: double. This data was floating-point number.
* 100: string. This means it was a string.
* 110: Boolean. And last one was a Boolean.

But other than those there were 2 other special data type. These being:

* Undefined – was the integer outside the possible integer value being -230 or -1073741824.
* Null - was the machine code NULL pointer or it was also the object that referenced zero.

Now that we know how JavaScript Typeof know what data type a value is, we can see how the Typeof null is an object. So, when JavaScript tries to know what the data type is and sees the type tag there is a zero, ultimately making the data type of null an object in JavaScript’s eyes.

But where there any things to make a fix for this? Yes, there was a fix was proposed to change type of null == “Object” typeof null == “null”' but was [rejected](https://web.archive.org/web/20160331031419/http:/wiki.ecmascript.org:80/doku.php?id=harmony%3Atypeof_null).

But the reason behind this obvious bug is that JavaScript was made is the littlest time possible making every error right as small or as big wasn’t as a concern as time was.

## Why hoisting is different with let and const?

Before seeing the difference between hoisting in let and const let us see what hoisting itself is. Hoisting is described as the moving of variable and function declarations to the top of their (global or function) scope. So, in JavaScript the interpreter goes through the cod twice, the steps are defined in two phase the compilation phase and the execution phase.

The compilation phase, in which is where variables and functions are stored in memory before the rest of your code is read, by this it means that the declared variables already have been put in store before being executed, creating the illusion of “moving” to the top of their scope hence they have been hoisted. While the variables are hoisted the interpreter gives them a value for initialization purposes as undefined. After the variables mentioned above have been hoisted the second step proceeds, it being the execution.

In the execution phase JavaScript’s Interpreter will start again from the first line of code but this time instead of storing them in a memory, the interpreter works its way down the code assigning the variables hosted in the first phase values of their specific data types, and in the same phase processing the functions.

So now that we know how a variable can be hoisted, “how does it being hoisted using let and const make a difference?” or “does it have one?”. The answer to that is a simple yes, when a variable is hoisted using “var” during the compilation phase the variable is given an initialized value of undefined until the execution phase starts. In the execution phase however the value of thus variable will be changes from undefined to its true or assigned value in the code.

When we come to variables hoisted using let and const, even though they are hoisted none the less the value of the variables vary from that of to variables hoisted using var. when the interpreter sees the var in front of the variable it automatically gives it an undefined initialization as we saw earlier, but in the case of the variable having the let or const declaration the interpreter leaves it without declaring it or without giving it any initialization.

Then how does it work if the variables are not initialized? Well JavaScript will initialize it in the second phase or the execution phase when the interpreter is running the code. In the runtime the variables will have been initialized and will have the value given to them when they are declared or hoisted in let or const. So, in the compilation phase the variables will have not been initialized and will not be accessible, meaning one cannot access these variables in this time. The time between them being declared and being evaluated has a term called the “Temporal Dead Zone”. It has been given this name because the is you try to access them with in the temporal dead zone you will get the following reference error.



This being said where can we reference let and const? To reference any variable declared by let or const we have to make sure that we don’t get a Reference Error. In order to do that we must not make the execution before the declaration of thus variables. In other terms if the variables aren’t executed in the execution phase before the declaration, we can still hoist them using let and const and not have a Reference Error, come up to us.

We can achieve that by either putting the variable before a function is called or executed. So, if we call a function before the variable is declared we will run in the Reference error but we can still have the function be hoisted and use let and const to declare the variable and still have the execution after the declaration by simply calling or executing the function in hand after the variables are declared and when the execution phase commences it first initializes the variables before executing the function it has hoisted and we will not have the runtime error ReferenceError.