

ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

CENTER OF INFORMATION TECHNOLOGY AND SCIENTIFIC COMPUTING

**Assignment 1 JavaScript**

**Prepared By:**- Sintayehu Sermessa

**Id:- ATR/8798/11**

**Section:- IT**

January 2020

Contents

[1. Is JavaScript Interpreted Language in its entirety? 2](#_Toc62248366)

[2. The history of “typeof null” 2](#_Toc62248367)

[3. Explain in detail why hoisting is different with let and const? 2](#_Toc62248368)

[4. Semicolons in JavaScript to use or Not to Use? 2](#_Toc62248369)

[5. Expression vs Statement in JavaScript? 2](#_Toc62248370)

[6. Reference. 2](#_Toc62248371)

1. Is JavaScript Interpreter Language in its entirety?

What is difference between Interpreter and compiler?

The interpreter is a computer program that directly executes instructions written in a programming without requiring them previously to have been compiled into a machine language program. It translates one Statement at a time.

A compiler is computer software that converts computer code written in one programming language (the source language, like JavaScript, … etc) into alternative programming language (the target language, like machine code (byte Code)).

“JavaScript used to be purely interpreted but that was many years ago. Nowadays, it is JIT (Just in Time)-compiled to native machine code in all major JavaScript implementations. JavaScript is *not* an interpreted language. I wish people would stop answering questions, saying that it is. In some cases, parts of a JavaScript program might be interpreted briefly, see below for explanation.”

Exactly when it’s compiled to machine code varies based on implementation. In the current V8 (used in Chrome and Node.js), it starts out using an interpreter since there is little reason to spend time compiling code that only runs once. However, if a function gets executed more than a couple of times, it’s immediately compiled into optimized native machine code.

For example: -

console.log ('Hoppity Hoppity');  
oops oops;

In this example, we should understand that the interpreter reads the code directly line by line, and by running this code here it shows this “Uncaught Syntax Error: unexpected token” error message, in theory, the “Hoppity Hoppity” should’ve printed the first line of code and throw the Syntax Error but instead show the error message without ever running the code. This means that JavaScript optimized it to the native machine code.

Another example on Hoisting: -

max (1, 2);  
// 2

function max (num1, num2) {  
 return num1 > num2 ? num1 : num2;  
}

This example also shows that JavaScript optimizes into compiling the code before run time, this shows that JavaScript knows the “max (1,2)” means in the first place, also shows that after compiling the code it interprets the code of the Function line by line so my conclusion is that All programming language are human-readable. Then translates to machine language. And the JavaScript engines have both compiler and interpreter to translate the JavaScript code on the web browser. The flow of the engine is that it interprets the JavaScript code and simultaneously uses the compiler to optimize the code being interpreted. So, it’s kind of both and Mach closer to compiler this days. First the interpreter reads the code and produces bytecode, secondly parsing takes place and turn it into AST (Abstract Syntax Tree), and then the code initially goes to an interpreter and spits out byte code and the compiler optimizes it and converts that part to machine code. It answers really depends on the implementation of that code.

1. The history of “typeof null”

In JavaScript, typeof null is 'object', which incorrectly advocates that null is an object (it isn’t, it’s a primitive value, this is a bug and one that unfortunately can’t be fixed, because it would break existing code.

This is a **bug** that exists since the beginning of JavaScript. The reason that this bug exists is simple. Each JavaScript value has a type tag. First 1-3 bits of each value are reserved for its type. The type tags for different types were as follows: -

* 000 - object
* 001 - int
* 010 - double
* 100 - string
* 110 - boolean
* To define **undefined,** they used a special number 2 -30 (which of range for integer)
* To define **null** NULL pointer was used.

In the initial version of JavaScript, values were storied in 32-bit units. The first 3 bits represented the data type tag followed by the remaining bits that represented the value.

For all objects it was 000 as the type tag bits. null was considered to be a special value in JavaScript from its very first version. null was a representation of the null pointer. However, there were no pointers in JavaScript like C. So null simply meant nothing or void and was represented by all 0’s. Hence all its 32 bits were 0’s. So, whenever the JavaScript interpreter read null, it considered the first 3 bits as type “object”. That is why typeof null returns “object”.

1. Explain in detail why hoisting is different with let and const?

Hoisting- During compile phase, just microseconds before your code is executed, it is scanned for function and variable declarations. All these functions and variable declarations are added to the memory inside a JavaScript data structure called **Lexical Environment**. So that they can be used even before they are actually declared in the source code.

Lexical environment – a lexical environment is place where variables and function live during the program execution.

For example: -

LexicalEnvironment = {  
 Identifier: <value>,  
 Identifier: <function object>  
}

helloWorld(); // prints 'Hello World!' to the console function helloWorld(){  
 console.log('Hello World!');  
}

Hoisting Function

function declarations are added to the memory during the compile stage, so we are able to access it in our code before the actual function declaration.

So the lexical environment for the above code will look something like this:

Example:- LexicalEnviroment ={

helloworld:<fun>

}

So when the JavaScript engine encounters a call to helloWorld(), it will look into the lexical environment, finds the function and will be able to execute it.

Only function declarations are hoisted in JavaScript, function expressions are not hoisted. For example: this code won’t work.

helloWorld(); // this here is Undefined

var helloWorld = function(){  
 console.log('Hello World!');  
}

As JavaScript only hoist declarations, not initializations (assignments), so the helloWorld will be treated as a variable, not as a function. Because helloWorld is a var variable, so the engine will assign is the undefined value during hoisting.

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

So, the initial lexical environment for the above code will look something like this:

lexicalEnvironment = {  
 a: undefined  
}

Hoisting Let and Const variables

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

or

const a = 3;

The output is Uncaught Reference Error: can't access lexical declaration 'a' before initialization

Which begs the question **is let and const variable not hoisted?**

So, the answer is All declaration (function, var, let, const) are hoisted in JavaScript, while the var declaration initialized with **Undefined**, but let and const declarations remain uninitialized.

They will only get initialized when their lexical assignment is evaluated during runtime by the JavaScript engine. This means they can’t access the variable before the engine evaluates its value at the place it was declared in the source code. This is what we call “**Temporal Dead Zone**”, A time span between variable creation and its initialization where they can’t be accessed.

For example: -

Const =a;  
console.log(a)  
a = 3;

In this case with const we con not do this because, it raises an error of Uncaught ReferenceError: a is not defined why? Before run time In lexical environment the a variable is set to uninitialized so, for const that I bit of a problem.

Const- we set a variable const if we don’t want to assign it after declaration or initialization. In this case it we can not change it so it will stack in the temporal dead zone we can’t access its value because it wasn’t initialized first this means that with const it must be initialized before run time.

1. Semicolons in JavaScript

References

1.https://www.quora.com

2. https://medium.com

3. <https://stackoverflow.com>