

ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

CENTER OF INFORMATION TECHNOLOGY AND SCIENTIFIC COMPUTING

**JavaScript and its unusual behaviors**

**Prepared By:** - Eyob Alemu

February 2020

Table of Contents

[Introduction 2](#_Toc62068390)

[Is JavaScript interpreted in its entirety? 3](#_Toc62068391)

[The history of “typeof null” 5](#_Toc62068392)

[Resources 8](#_Toc62068393)

# Introduction

JavaScript a really powerful language and it can be used in many different areas. But for all its powerfulness and usefulness, some the languages behaviors are a little bit odd. This document tries to see some of those odd behaviors and reason about how they are the way they are.

The behaviors of JavaScript seen in this document are:

* Whether or not JavaScript is interpreted or compiled language
* Why is the type of null an object when it should be primitive data type
* Hoisting with var vs let and const
* The usage of semicolons
* Expressions and statements in JavaScript

The reasoning in this document is based on different sources mentioned in the references page. But when there are personal opinions of the writer, it will be mentioned right along the assumption that it is the writer’s opinion.

The document uses “Courier New” font for code snippets.

# Is JavaScript interpreted in its entirety?

Every program is a set of instructions, this instructions can be written in any high level programming language. But these high level programming languages are mainly designed to be human readable which makes them hard for machines to understand as they are. Therefore, in order for machines to understand them, these programs need to be converted to machine readable format also known as machine code. To do this there are two mainly used methods interpretation and compilation.

Compilation is the process of converting an entire source code to the target machine code and executing. Compiled languages will have a build step where the compiler takes the source code and converts it to the respective machine code. Since the entire source code is going to be converted to machine code once, compiled languages are fast at execution time. But modifying the source code results in recompilation.

In interpretation the source code is read and executed by a program call the interpreter. This interpreter reads the source code line by line and executes each command one at a time. Due to this line by line execution, interpreted languages are relatively slower. But when editing the source code there won’t be any need for recompilation or rebuild steps since the interpreter is reading line by line.

According to the MDN documentation, “JavaScript is a lightweight, interpreted, or just-in-time compiled language.” If interpretation and compilation are as described above, then what is just-in-time compilation?

Just-in-time compilation (JIT) is a method used to improve performance of interpreted languages. While interpreted program is being executed the JIT compiler determines the most frequently used code and compiles it to machine code. This machine code is an optimized code for the target CPU architecture then every time that same block of code is needed, this already compiled machine code will be used. This way JIT helps to avoid the inefficiency caused by recompiling a code block over and over again.

A JavaScript engine is a program that executes a JavaScript code and it can be implemented as a plain old interpreter or as JIT compiler.

Some of the engines that implement plain old interpretation are,

* Futhark : ECMAScript engine for Opera browser versions 9.5 – 10.10
* Jscript : engine used in Internet Explorer for version up to IE9
* Rhino : one of several engines from Mozilla using the Java platform
* V4 (QJSEngine) : Qt’s newer ECMAScript engine

Some of the engines that use JIT are,

* Carakan: engine used by Opera browser in versions 10.5 – 15
* V8: an open source engine used in Google Chrome, Node.js, Deno and V8.NET
* Chakra (Jscript9): and engine used in IE after IE9 also used in Microsoft Edge
* SpiderMonkey: the engine used in Mozilla Firefox and other Gecko applications

The above mentioned engines are just an example and there are lots of other engines implemented either as an interpreter or JIT. What all this shows is that classifying the JavaScript language is more or less dependent on the implementation. It is the writer’s opinion that JavaScript can be both compiled and interpreted language depending on the implementation used by the engine. Therefore, it is hard to say that JavaScript is just an interpreted language specially with the introduction of JIT.

# The history of “typeof null”

In JavaScript all values are either Objects or Primitives. The difference between objects and primitives is that primitives are immutable and properties can’t be added to them. Primitive values are equal if they have the same value.

var a = “String”;

a.foo = 123; // this wouldn’t do anything

console.log(a.foo); // prints undefined

var b = “string”;

console.log(a == b) // true

Primitive values in JavaScript are:

* undefined
* Booleans
* Numbers
* Strings
* Null

All non-primitive values in JavaScript are objects. Objects are mutable and properties can be added to them. When comparing objects, the objects need to be the same for equality even if two objects have the same value, unless they are the same object they won’t be equal.

var a = {};

a.foo = 123; // this would do add property foo

console.log(a.foo); // prints 123

var b = {};

console.log(a == b) // false

console.log(b == b) // true

JavaScript has multiple ways to check a type of certain value. One of these methods is typeof. This method distinguishes between primitive types and objects. Below is the result of typeof with different values.

|  |  |
| --- | --- |
| Operand | Result |
| undefined | "undefined" |
| null | "object" |
| Boolean value | "boolean" |
| Number value | "number" |
| String value | "string" |
| Function | "function" |
| All other values | "object" |

As shown above typeof returns object for null. This is a bug that dates back to the first days of JavaScript. On the first version of JavaScript, values were represented by a 32 bit units, which consisted a type tag (1-3 bits) and the actual data. There were five type tags and they were stored in the lower bits of the units.

* 000: object
* 1: int
* 010: double
* 100: string
* 110: boolean

The two values undefined and null were special values represented by the integer -230 and the machine code NULL POINTER reference respectively. In most platforms the NULL POINTER reference is 0x00 or an object type tag and value zero. This makes the typeof function treat null as an object rather than a primitive value.

So if this is a bug, why not fix it? There reason is, introducing a breaking change to JavaScript has a huge consequences. Many sites might be running on code written years ago and they might be dependent on this bug. If a breaking change is introduced to the language, all the sites that used to rely on the way it used to work will break and that’s not good for the internet.

# Hoisting with let and const

JavaScript code is interpreted in the execution context it is written in. The JavaScript engine interprets the code written in this execution context in two separate phases, compilation (creation) and execution phases.

In compilation phase all declared variables and functions are put into memory and this is called hoisting. In hoisting it is not the code that gets rearranged or the variables don’t change place from where they were originally written. Instead any declared variable or function is moved to a special place in memory before the rest of the code is read. Assigning the actual values to variables and processing the function happens on the execution phase, which happens at run time.

When declaring variables with the keyword var the variables are hoisted and initialized with the value undefined. Allowing something like the following to work just fine.

console.log(a); // prints undefined

var a = 2;

But when it comes to let and const, the variables will be hoisted but won’t be initialized with any value putting them in Temporal Dead Zone (TDZ). TDZ is the state where variables are un-reachable in other words they are in scope but they aren’t declared. For example, the following code raises an error because the local a declared by let takes precedence over the global one declared with var. But when the local one is hoisted it wasn’t given any value and it is TDZ; therefore, the engine throws an error saying that a is not defined.

var a = 2;

{

console.log(a); // this raises ReferenceError

let a = 4;

}

The same is true for const. Any value declared with const will be hosted and put in TDZ in other words it won’t be assigned any value. Therefore, trying to access the value before it is assigned results in an error.

The problem with the hosting of let and const variables is, the variables can’t be accessed before they are assigned a value. This means a code that is trying to access a variable declared with let and const can’t run before the assignment of this variables.

function sayName() {

console.log(name); // this is fine, prints Jhon Doe

}

let name = “Jhon Doe”;

sayName();

function sayName() {

console.log(name); // this throws ReferenceError

}

sayName();

let name = “Jhon Doe”;

# 

# Resources

* Interpreted vs Compiled programming languages - <https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/>
* Just In Time Compilation - <https://www.freecodecamp.org/news/just-in-time-compilation-explained/>
* List of ECMAScript engines - <https://en.wikipedia.org/wiki/List_of_ECMAScript_engines>
* The history of “typeof null” - <https://2ality.com/2013/10/typeof-null.html>
* Categorizing values in JavaScript - <https://2ality.com/2013/01/categorizing-values.html>
* Why “typeof null === object” will stay in JavaScript - <https://softwareengineering.stackexchange.com/questions/371136/why-typeof-null-object-will-stay-in-javascript>