

ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

CENTER OF INFORMATION TECHNOLOGY AND SCIENTIFIC

COMPUTING

Assignment I

Name: Mikiyas Daniel Tefera

Id: ATR/1876/11

Section:It

To: Fitum Almu

January 2020

Contents

[IS JAVA INTERPRETED LANGUAGE IN IT ENTIRETY? 4](#_Toc62314273)

[How does javascript work? 4](#_Toc62314274)

[Just-in-time compiler (JIT) 4](#_Toc62314275)

[Conclusion 5](#_Toc62314276)

# IS JAVA INTERPRETED LANGUAGE IN IT ENTIRETY?

To answer this question we must look at how javascript works in the first place.

## How does javascript work?

For starters, javascript is supposed to run on browsers so for the browser to understand the language it has a javascript engine.

Javascript engine is a program that goes through the javascript code, character by character, and transforms it to a language that the computer CPU understands and executes which is machine code.

The javascript engine executes the javascript line by line which makes it go through the whole code blindly consequently making it redundant and inefficient, which means the javascript engine has to keep retranslating the same code like when you are in a loop which makes the execution slow. This where JIT comes.

### Just-in-time compiler (JIT)

As a way of getting rid of javascript interpreter inefficiency browsers started mixing compilers in.

Different browsers use different ways but the idea is still the same. They added a new part to the engine, called a monitor or also known as a profiler.

The profiler is used to watch the code as it runs making a note of how many times it runs and types used, so at first, the monitor just runs everything through the interpreter.

If the same lines of code are run multiple times the segment of codes is called hot.

When the part of the code is getting the getting very hot the monitor will send it off to the optimizing compiler which will make it complied and stored consequently making a faster version of the code, to do that the compiler has to make some assumptions.

For example, if it can assume that all objects created by a particular constructor have the same shape-that is, that they always have the same property names and that those properties were added in the same order – then it can continue to be true.

But this might not always be true a code can have 99 objects that all have the same shapes, but the 100th might be missing a property.

So the compiled code needs to check before it runs to see whether the assumption is valid. if they are, then the compiled code runs .but if not, the jit assumes that it made the wrong assumptions and trashes the optimized code. which makes the execution go back to the interpreter. this process is called deoptimization.

Even if JIT is supposed to make the code run faster it can cause unexpected performance problems, to avoid that browsers have limits to break out of this optimization. deoptimzation cycle when needed.

## Conclusion

So, in my opinion, even if JIT acts as an optimizer for the javascript code we can't deny the fact that the java engine uses basic definitions of compiled language along the way, which paves the way for usage of both compiled and interpreted, so to address this question javascript isn’t interpreted language in its entirety but **depends on its implantation,** which means it depends on how the java engine choose to implement it, it could be solely interpreted, compiled or both at the same time, which most java engines do nowadays.

# The history of “typeof null”

In javascript, typeof null returns an object, which suggests that its an object when it is a primitive value.

In the first version of "typeof null," the bug comes from the JavaScript error achieve. In this version, with a 32-bit value is stored, the actual data comprising a type of marker (1-3) and a representation of the value. On the lower, a total of five kinds of tag storage type:

* 000: **object**. The data is a reference to an object.
* 1:**init**.the data is a 31 bit signed integer.
* 010:**double**. The data is a reference to a double floating-point number.
* 100:**string**.the data is a reference to a string.
* 110:**Boolean**. The data is a Boolean.

If the lower bit is 1, then only one type of flag bit long; if 0, then labeled with a 3- bit type, two additional bit provide four types

Two special values:

* To define **undefined**  (JSVAL\_VOID) integer -230(An integer number outside the range).
* To define **null**  (JSVAL\_NULL) the machine code is a null pointer. or: an object type tag plus a reference that zero.

**Code of tyeof**

**JS\_PUBLIC\_API(JSType)**

**JS\_TypeOfValue(JSContext \*cx, jsval v)**

**{**

**JSType type = JSTYPE\_VOID;**

**JSObject \*obj;**

**JSObjectOps \*ops;**

**JSClass \*clasp;**

**CHECK\_REQUEST(cx);**

**if (JSVAL\_IS\_VOID(v)) { // (1)**

**type = JSTYPE\_VOID;**

**} else if (JSVAL\_IS\_OBJECT(v)) { // (2)**

**obj = JSVAL\_TO\_OBJECT(v);**

**if (obj &&**

**(ops = obj->map->ops,**

**ops == &js\_ObjectOps**

**? (clasp = OBJ\_GET\_CLASS(cx, obj),**

**clasp->call || clasp == &js\_FunctionClass) // (3,4)**

**: ops->call != 0)) { // (3)**

**type = JSTYPE\_FUNCTION;**

**} else {**

**type = JSTYPE\_OBJECT;**

**}**

**} else if (JSVAL\_IS\_NUMBER(v)) {**

**type = JSTYPE\_NUMBER;**

**} else if (JSVAL\_IS\_STRING(v)) {**

**type = JSTYPE\_STRING;**

**} else if (JSVAL\_IS\_BOOLEAN(v))**

**type = JSTYPE\_BOOLEAN;**

**}**

**return type;**

**}**

The code executes like this :

* In (1), the engine is first checked whether the undefined value V(VOID), By checking whether the comparison value is equal to;

#define JSVAL\_IS\_VOID(v) ((v) == JSVAL\_VOID)

* Next check (2) if there is a value of the object tag. If it can be called. (3) or its internal property [[class]] mark it as a function of (4), then v is a function. Otherwise, it is an object. This is the result of typeof null generated.
* The next check number, string, and Boolean values. Not even explicitly check if it is null.

#define JSVAL\_IS\_NULL(v) ((v) == JSVAL\_NULL)

This bug isn’t removed yet and even for the foreseeable future as it will break the existing code that relies exactly on this principle, which means that every web application out there will need to undergo a refactoring.

# Why hoisting is different with let and const?

The question arises how hositing with let and const is differenent from var, to look at that lets look at what hosting is and how it differes between them.

## What is hosting?

During compilr phase, just before a code is excuted , it is scanned for function and variable declaration. Which all these functions and variables are added to a memory inside javascript data structure called Lexical Enviroment. This is done so that they can be used even before they are declared in the source code.

A lexical environment is a data structure that holds **identifier-variable mapping**. (here **identifier** refers to the name of variables/functions, and **the variable**is the reference to actual object [including function object] or primitive value).

This is what a lexical environment conceptually look like:

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

Hoisting is essentaily a javascript mechanism where variables and functions declarations are moved to the top of their scope before code execution.meaning is we do this

console.log (greeter);

var greeter = "say hello"

its is interpreted as this

var greeter;

console.log(greeter); // greeter is undefined

greeter = "say hello"

## Scop

It essentially means where these variables are available for use. For instane , var based variables are function scope meaning they are not available outside the fuction , while let and const varables are block scoped, meaning cannot be accessed from outside the block.we will dig in discrptions leater.

## Variables lifecycle

When the engine works with variables, their lifecycle consists of the following phases:

1. **Declaration phase** is registering a variable in the scope.
2. **Initialization phase** is allocating memory and creating a binding for the variable in the scope. At this step the variable is automatically initialized with undefined.
3. **Assignment phase** is assigning a value to the initialized variable.

## var variables lifecycle

lets look at this code

function multiplyByTen(number) {

console.log(ten); // => undefined

var ten;

ten = 10;

console.log(ten); // => 10

return number \* ten;

}

multiplyByTen(4); // => 40

When JavaScript starts executing multipleByTen(4) and enters the function scope, the variable ten passes declaration and initialization steps, it will add that variable to the lexical environment and initialize it with undefined, before the first statement. So when calling console.log(ten) it is logged undefined. And when the engine reaches the line (during execution) where the actual assignment is done which is ten = 10, it will update the value of the variable in its lexical environment. So the lexical environment after the assignment will look like this:

lexicalEnvironment = {  
 ten: 10  
}

After assignment, the line console.log(ten) logs correctly 10 value.

Variables declared with **var** keyword can be redeclared at any point in the code even within the same execution context.

Var x = 1;

Var x = 2;

Console.log(x): //prints 2

## Let and const

Both Let and const are block scoped meaning a varabile declared in a block with let or const is only available for use within that block.

let greeting = "say Hi";

let times = 4;

if (times > 3) {

let hello = "say Hello instead";

console.log(hello);// "say Hello instead"

}

console.log(hello) // hello is not defined

using hello outside its block (the curly braces where it was defined) returns an error. This is because let variables are block scoped .

let variables are processed differently than var. The main distinction is that declaration and initialization phases are **split**.

while the var declarations are initialized with undefined, but let and const declarations remain uninitialized.

They will only get initialized when their lexical binding (assignment) is evaluated during runtime by the JavaScript engine. This means you 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.

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).

let a;  
console.log(a); // outputs undefined  
a = 5;

Here during the compile phase, the JavaScript engine encounters the variable a and stores it in the lexical environment, but because it’s a let variable, the engine does not initialize it with any value. So during the compile phase, the lexical environment will look like this:

lexicalEnvironment = {  
 a: <uninitialized>  
}

Now if we try to access the variable before it is declared, the JavaScript engine will try to fetch the value of the variable from the lexical environment, because the variable is uninitialized, it will throw a reference error.

During the execution, when the engine reaches the line where the variable was declared, it will try to evaluate its binding (value), because the variable has no value associated with it, it will assign it undefined.

So the lexical environment will look like this after execution of the first line:

lexicalEnvironment = {  
 a: undefined  
}

And undefined will be logged to the console and after that 5 will be assigned to it and the lexical environment will be updated to contain the value of a to 5 from undefined.

We can reference the let and const variables in the code (eg. function body ) even before they are declared, as long as that code is not executed before the variable declaration.

For example, This code is perfectly valid.

function foo () {  
 console.log(a);  
}let a = 20;  
foo(); // This is perfectly valid

But this will generate a reference error.

function foo() {  
 console.log(a); // ReferenceError: a is not defined  
}  
foo(); // This is not valid  
let a = 20;

## the difference between let and const

let can be updated within its scope.  But not re-declared within its scope

So while this will work:

let greeting = "say Hi";

greeting = "say Hello instead";

this will return an error:

let greeting = "say Hi";

let greeting = "say Hello instead"; // error: Identifier 'greeting' has already been declared

However, if the same variable is defined in different scopes, there will be no error:

let greeting = "say Hi";

if (true) {

let greeting = "say Hello instead";

console.log(greeting); // "say Hello instead"

}

console.log(greeting); // "say Hi"

Why is there no error? This is because both instances are treated as different variables since they have different scopes.

const cannot be updated or re-decleared.

So if we declare a variable with const, we can neither do this:

const greeting = "say Hi";

greeting = "say Hello instead";// error: Assignment to constant variable.

nor this:

const greeting = "say Hi";

const greeting = "say Hello instead";// error: Identifier 'greeting' has already been declared

Every const declaration, therefore, must be initialized at the time of declaration.

## Conclusion

While they all are hosited to the top of their scope. But wile var variables are initialized with undefined, let and const variables are not initialized.

# Semicolons in javascript: to use or not to use?

The official guidelines say that JS uses a semi colon to terminate a statement.

The ECMAScript spec describes how, if a statement is not explicitly terminated with a semicolon, sometimes a semicolon will be automatically inserted by JavaScript engine (called “Automatic Semicolon Insertion” (ASI)). if you don’t physically put them in, the engine will do it for you. It just happens behind the scenes. ASI helps the parser to determine when a statement ends.

## Conclusion

I say its advisable to use semicolons rather than dependening on automatic insertion . plus it is a good habit to use it often casue it will develop ones skill for other languges that depend on  an explicit semicolon.