ADDIS ABABA INSTITUTE OF TECHNOLOGY

CENTER OF INFORMATION TECHNOLOGY AND SCIENTIFIC COMPUTING

DEPARTMENT OF INFORMATION TECHNOLOGY AND SCIENTIFIC COMPUTING

JavaScript Fundamentals

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?

1.1 Definition

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?

1.2 Conclusion

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.

2. The history of "typeof null"

1.1 Definition

The History behind the confusion of the typeof null is that when JavaScript reads or tries 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

```
typeof 123
```

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:

```
typeof 123 // "number"
```

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.

1.2 Reason Behind the 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 -2^{30} 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.

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. So the JavaScript first version had to come out with as an obvious bug as this one to manage the time frame it was released in and now it is part of the original code itself.

3. Why hoisting is different with let and const?

1.1 Definition

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.

1.2 Hoisting in Let and Const

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.

```
console.log(name); // Uncaught ReferenceError: name is not defined
```

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.

4. Semicolons in JavaScript: Use or Not use?

1.1 Definition

Why are semicolons not as important in JavaScript as that of Java? Well the answer lies in the first problem JavaScript and Java are two totally different languages to begin with one taking the name of Java for popularity and marketing reason and not to do with the language itself.

That being said the importance for semicolons isn't neglected in JavaScript. The semicolon indicates the end of a line in the code, but then how does JavaScript know if the line has ended if we don't put a semi colon at the end. For this JavaScript uses something called the ASI or the Automatic Semicolon Insertion. This is the reason why semicolon are optional in JavaScript, the ASI will put the semicolon needed when there isn't one there.

Even though the ASI interprets a semicolon when it is needed and can be found it doesn't mean that the semicolon is put there in an actual presence, it means that the ASI will interpret the line as if the code has a semicolon in that line or in that specific spot needed.

1.2 ASI Rules

For ASI to do its job there are guidelines or rules that govern it because it can't be putting semicolons everywhere it wants. So, what are the guidelines to this:

1. A semicolon will be inserted when it comes across a line terminator or a '}' that is not grammatically correct. So, if parsing a new line of code right after the previous line of code still results in valid JavaScript, ASI will not be triggered.
2. If the program gets to the end of the input and there were no errors, but it's not a complete program, a semicolon will be added to the end. Which basically means a semicolon will be added at the end of the file if it's missing one.
3. There are certain places in the grammar where, if a line break appears, it terminates the statement unconditionally and it will add a semicolon. One example of this is return statements.

The guidelines above are the rules that ASI uses to put the semicolons needed in the code or the line break.

For example in the first rule it defines how a semicolon will be put if the ASI notices the that a line terminator or a "}" is reached but the line has not been put to an end or is still open.

So,

Var door

Var window

Will in turn be changed into:

Var door;

Var window:

But that doesn't mean it will always work in your favor. How you may ask:

Var a

b = 3;

will result in the ASI think that there should be a line break and put: var a;

b=3;

but it obey the grammar rules like:

a = b + c

(d+ e) = f;

Will result in:

a = b + c(d + e)=f;

In rule number 2, if the file is missing a semicolon it will trigger the ASI and it will put the semicolon to end the file and not have a syntax error.

Rule number 3, in this rule the ASI will make a line of code have a semicolon if the line seems to have a line breaker needed. For example in the case below the return statement triggers the ASI as a line breaker and will make it drop a semicolon making it a break the line of code there.

```
Function getCar{
        Return
        {
        carModel : "KIA"
}
}
```

In the above function we are trying to get the car model returned but what the ASI sees is that after the return statement there should be a semicolon because there isn't "{" making it break the line there.

```
Function getCar{
        Return;
        {
        carModel : "KIA"
```

```
}

}
```

The code above will return undefined and will not show us the model of the car as we wrote, to fix this we need to put the line breaker in the same line of the “return” to have the same outcome as we wished and not the undefined message we didn’t expect.

```
Function getCar{

        Return{

        carModel : “KIA”

}

}
```

This will now will return to us the car model instead of the undefined message we got before because the ASI now sees the “{” and will not put the semi colon to fix the issues breaking our code.

5. Expression vs Statement

1.1 Definition

JavaScript distinguishes between a statement and an expression, but before that lets see what make a statement a statement and an expression and expression.

1.1.1 Statement

A statement is used to do something or make something happen, meaning it is used to make the program flow controllable. So, JavaScript has some statements like those of:

- break, continue, for, for...in, function, if...else, new, return, var, while, with

1.1.2 Expression

Meanwhile, an expression produces a value and can be written wherever a value is expected, for example as an argument in a function call. So, an expression is roughly defined as what we give or define it as to have a certain value which will be assigned to it or will it have already been assigned.

1.2 Differences and Similarities

To see the difference between the two we will see the similar part of them and how they operate differently and point out the little thing that define them.

When a condition is needed to be written the statement version of it is:

```
var a;
   if (b <= 0) {
      a =- b;
   } else {
      a = b;
   }
```

While in the case of it expression version of the conditional operand:

```
var x = (y >= 0 ? y : -y);
```

In this case the expression inside the parenthesis does the exact job.

In JavaScript statements are chained by a semicolon which is used to know where one ends and the other statement begins. But in the case of expression the we can use a comma to chain the expression which will be returning the value of the second expression.

```
Var x = (“a”, “b”)
 x // returns
“b”
```

Then there are Function declarations and Function expressions, starting from the syntax there are differences between them:

Function declarations are written as:

```
Function c(){
            Console.log(“Something”);
}
```

Where as in the case of function expressions:

```
Var x = function c() {
 Console.log (“Something”);
```

In the case of the Function declaration the function is hoisted and is given a value of undefined in the compilation phase, but in the case of Function expression it the function isn’t hoisted but the var x is. So if we want to declared a function before defining it we would need to use a function declaration. But we can immediately invoke a Function expression but we can’t do that with a function declaration.

Evaluating an object literal using eval parses its argument in statement context. You have to put parentheses around an object literal if you want eval to return an object.

```
> eval('{ foo: 123 }')
 123
```

```
> eval('({ foo: 123 })')

{ foo: 123 }
```

The following code is an *immediately invoked function expression* (IIFE), a function whose body is executed right away (you'll learn what IIFEs are used for in Introducing a New Scope via an IIFE):

```
> (function () {return 'abc' }())

 'abc'
```

Reference

1. Medium, https://medium.com/@allansendagi/inside-the-javascript-engine-compiler-and-interpreter-c8faa638b0d9, January 24, 2021
2. Medium, https://medium.com/@almog4130/javascript-is-it-compiled-or-interpreted-9779278468fc, January 24, 2021
3. 2aliity, https://2ality.com/2013/10/typeof-null.html#:~:text=In%20JavaScript%2C%20typeof%20null%20is,it%20would%20break%20existing%20code.&text=The%20data%20is%20a%20reference%20to%20an%20object., January 24, 2021
4. Medium, https://medium.com/javascript-in-plain-english/how-hoisting-works-with-let-and-const-in-javascript-725616df7085, January 24, 2021
5. Free code map, https://www.freecodecamp.org/news/var-let-and-const-whats-the-difference/#:~:text=var%20variables%20can%20be%20updated,const%20variables%20are%20not%20initialized., January 24, 2021
6. Dev, https://dev.to/adriennemiller/semicolons-in-javascript-to-use-or-not-to-use-2nli, January 24, 2021
7. W3schools, https://www.w3schools.com/js/js_statements.asp, January 24, 2021
8. Info World, https://www.infoworld.com/article/2077317/understanding-and-using-javascript-statements.html#:~:text=Statements%20are%20used%20in%20JavaScript,independently%20of%20any%20JavaScript%20object., January 24, 2021