
# ADDIS ABABA INSTITUTE OF TECHNOLOGY
## CENTER OF INFORMATION TECHNOLOGY AND SCIENTIFIC COMPUTING
### DEPARTMENT OF INFORMATION TECHNOLOGY

# JavaScript Reading Assignment

**Prepared By:** - Betemariam Moges, ATR/0536/11

**Submitted To**: Mr. Fitsum Alemu

January 2021

# Contents

# List of Tables

# 1. Is JavaScript Interpreted Language in its entirety? Check this Link and Make Up your justification.

The short answer is no. JavaScript is not an interpreted language in its entirety.

It was designed in by Brendan Eich and introduced in the Netscape browser in 1995 with the goal of adding interactivity to web pages. Its purpose was to work inside the browser and provide API to manipulate the DOM. Performance wasn't the first concern. It was fully interpreted - read and executed line by line by an interpreter without needing it to be compiled to machine language.

But it was slow and needed improvement for different reasons, of which, one was the fact that it was unoptimized unlike compiled languages. It was redundant and inefficient. For example, when a loop containing a function call that invokes a calculation and returns a result is ran, it would read the code line by line, interpreting and executing each and every iteration even if it was the same function call with the same arguments, unlike complied languages where there'd be complier optimizations. For this case, one simple optimization could be replacing the function call with its return value, avoiding going to the function every time. Another issue was error handling. When a code is being executed in a line-by-line basis, say line 6 will executed before line 7 is read, which could potentially be a line with an exception and so this makes every execution a possible failure, a doomed partial execution, which might be undesirable. Additionally, since JavaScript was dynamically typed the type checking and computation made it very slow - iterating over an Array would be an example in point here showing, considering how arrays can consist of multiple data types.

Due to the resulting slowness of JavaScript, Google was having a hard time providing a Google Maps that was highly interactive, smooth and dynamic. And so, they come up with better approaches and build the modern JavaScript engine V8 in 2008 and had made huge progress since then.

Today, the modern V8 engine and other browser engine with similar mechanisms run our JavaScript codes and they had improved running efficiency a whole lot. It uses lots heuristics (essentially self-teaching, data and result based) and optimizations to fasten up the process. Here are the steps from parsing () to execution:

1. The script is 'parsed' into tokens that are like the units or alphabets of the language and are used to make an Abstract Syntax Tree (AST) of the code, which is a tree-like structure representing the code and their relationships. Here the parser makes optimization choice like not parse functions that aren't called and will only parse them when they're called like with a button click.

2. The engine interpreter (V8 Ignition Interpreter for Chrome) interprets the AST to bytecode, which is a bit more abstract form of the low-level machine code. One optimization that's done here is the using of only one shared copy of the same meta-data/object properties of multiple instances to avoid wasting memory with duplicated information. Here a component called the profile monitors the codes that runs and watches if a certain snippet of code is run more than a certain number of times. Then it passes it to the 'optimizing' compiler which then tries to form an optimized machine code of it so as to save it and easily run it when needed without going through the whole process again. But that is assuming things like the type of a variable is not changed. The engine also has a cache where such repeated bytecodes and locations of objects in the bytecode are stored for the same purpose of make the process faster and more efficient.
3. The compile (called Turbofan) takes in the bytecode and changes it machine code. When using those saved code snippers, the compiler checks if the assumptions it made were true and if not, the engine will go back to using the bytecode being generated. Finally, the JavaScript Virtual Machine executes the program.

These sets of optimizing techniques are called Just-in-Time compiling (JIT). And the steps make it clear that JavaScript engines use compilers and thus JavaScript is not a fully interpreted language. It would seem more of a hybrid - as there can be different implementations (like one that mostly use interpretation) - but leaning to compiled language as interpretated language hardly touches the half of what JavaScript is and how it shares similar execution process with Java and how it reports static (compile-time) errors (before execution).

## 2. **The history of “typeof null”.**

The "typeof" keyword allows one to classify primitive data types and tells them apart from objects. The table below shows its output for different operands.

Table 1 typeof outputs

| OPERAND | RESULT |
|---|---|
| **UNDEFINED** | "undefined" |
| **NULL** | "object" |
| **BOOLEAN VALUE** | "boolean" |
| **NUMBER VALUE** | "number" |
| **STRING VALUE** | "string" |
| **FUNCTION** | "function" |
| **ALL OTHER VALUES** | "object" |

The result for "null" is a bug. It is a primitive data type, not an object. This bug traces its roots the first version of JavaScript when Brendan Eich had very little time to write the source code. He wrote it in 10 days.

He was writing in C and wrote typeof to check every of the above possible data types except the null. Values used to be stored in 32-bit units, 1-3 of which were used to tell the data type. They were 1, 000, 010, 100, and 110. The 000 reference was for an object. But here's the catch, the null was an object type tag with a reference of zero. Because of this the typeof saw the type tag of null and read it as an object.

## 3. **Explain in detail why hoisting is different with let and const?**

Hoisting is the when variable and function declarations are put into memory during the compilation phase. With var for example, the interpreter scans the entire code and puts its declaration in the memory (in the VO (Variable Object) to be exact). And after it does so, it will initialize it with a value of undefined, and so if you used a var before its declaration it won't throw an error. Instead, it will work normally with the value undefined.

What changes with let and const is that they won't be initialized with undefined or anything like var. Take the following example:

```
console.log(x); //throws an error

let x;

console.log(x); //gives undefined

x = 1;

console.log(x);//gives 1
```

The interpreter will first put the declaration of x to memory without initializing it and before going to the execution phase and starts executing each line. And it will throw a throw an error while running the first log as x isn't initialized. In the second line it would initialize x with undefined as it will do with any other uninitialized variable and that's what it will output in the second log. And finally, the third log would give 1 since it's been initialized in the previous line. I should note that the same can't be done with const as it must be initialized while it's declared. And so there won't be the middle log.

The period between line 1 and the initialization line above is called the temporal dead zone (TDZ), owing to the fact that the let variable can't be used in that span. The TDZ is about the time and not

the order of the code lines because not their physical ordering but execution ordering that defines it. The code below depicts that.

```
function func () {console.log(x)}; //no error

let x = 1;

func();
```

Since the log line is executed after func is invoked and therefore after the let variable is initialized, there won't be any error thrown - the order of execution doesn't cause that so.

## 4. **Semicolons in JavaScript: To Use or Not to Use?**

Before answering the question, answering why are semicolons optional will help.

Not all semicolons are optional in JavaScript. The semicolons inside for-loop parentheses are an example. But other than the likes of those, writing semicolons is optional. Why? Because JavaScript has an Automatic Semicolon Insertion (ASI) mechanism whereby it interprets unwritten semicolons on places based on the Rules of ASI as put by the ECMAScript Specification. So, what are the rules and what do they say about whether we should use semicolon or not?

1. If an illegal JavaScript new line or '}' is encountered during parsing, a semicolon is interpreted before it. Note here that there must be non-grammatical continuation up to the new line or curly brace if a semicolon is to be interpreted.

So, for something like this:

```
var a

b=

3;
```

It would start parsing from var, continues a but will found a problem over at b since there could not be a legit JavaScript reading '**var a b…**'. So, since there's already a new line and a grammatical error, it will add a semicolon before b, and continue on its reading with '**b = 3;**'. A trickier example would be:

```
var x= "Hi"
var y = "Hello" + x
```

```
['h'].forEach((l) => console.log(l))
```

Here it would peacefully read up until "Hi", see that it can't continue with var, and add a semicolon after "Hi". But when it reaches x at the end of the second line, it will not put a semicolon there since it won't see any grammatical error - it can simply keep going on with '**var y = "Hello" + x['h'].forEach((l) => console.log(l))**' since for example x['h'] might be in the form of object['property name'].

2. At the end of the program, if the parser can't parse the whole of the script as complete program, and if there are no errors, it will append a semicolon in the end. This is very helpful if the code in letter gone be concatenated with another one, since things like in the previous example might happen (the end of one and the start of the other being connected somehow like if one ends with a variable and other starts with parentheses, the parser might take it as function call).

3. There are a certain set of keywords like return, break, continue and the ++ and -- assignment that if you put a new line after would be interpreted to have a semicolon in the end, whether there's any grammar rule broken or not in the continuation. So, this rule can be an exception to rule 1 in certain scenarios. For example:

```
x
++
y
```

Normally with rule one, one would assume the parser would go up to '++' and make a semicolon since '**x++**' is grammatically correct and '**x++y**' isn't. But according to rule 3, the ASI will read a semicolon after x and the code would be read like '**x; ++y;**'. The advice here that the specification is trying to make is "A postfix ++ or -- operator should appear on the same line as its operand."

Likewise, if there's a return statement, or any of the listed above, is alone in a line then a semicolon will be interpreted just after it in the same line.

So, what does all these tell us? Well, one thing is clear, we must know what we're doing when dealing with semicolons in JavaScript as the ASI is always gone be there whether we use semicolons or not. Clearly, if one is to avoid using optional semicolons, he/she better know the Rules of ASI and be mindful when coding. Otherwise, it would seem a good habit to keep those semicolons and avoid dealing with ASI if at all possible, especially since debugging would considerably be harder - since even a harmless looking new line can make your function return undefined. Additionally, if one minifies the program using a program, it will very likely mess with ASI if the optional semicolons are not used. Besides these, there is no real performance advantage or otherwise with either using not using semicolons.

## 5. **Expression vs Statement in JavaScript?**

A valid expression in JavaScript is a collection of tokens that results in a value. There are different types of expressions. Arithmetic expressions evaluate to a number. For example, "3;". String expressions are expressions that evaluate to string. Logical expressions are those that evaluate to a Boolean value. Primary expressions are those stand-alone literal values, certain keywords and variable values. Assignment expressions are those that use the = operator to assign a value to a variable.

A statement is a collection of one or more expressions and keywords that consists of a specific action. Declaration statements are one type of statements. Their action is to create variables and function. Conditional, Loop and Jump statements are types of statements that act on based on the expressions, loop through codes of line or jump to a specific location in the program, respectively.

Functional expressions are those functions assigned to a variable. Named or not they are not a statement but a value as they are being assigned to a variable. The conditional ternary operator is also an expression since it evaluates to a value, say as a function argument or variable initialization.

# References


- https://medium.com/@almog4130/javascript-is-it-compiled-or-interpreted-9779278468fc
- https://blog.usejournal.com/is-javascript-an-interpreted-language-3300afbaf6b8
- https://blog.greenroots.info/javascript-interpreted-or-compiled-the-debate-is-over-ckb092cv302mtl6s17t14hq1j
- https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/get-started/ch1.md#whats-in-an-interpretation
- https://medium.com/jspoint/how-javascript-works-in-browser-and-node-ab7d0d09ac2f
- https://medium.com/jspoint/how-javascript-works-in-browser-and-node-ab7d0d09ac2f
- https://www.telerik.com/blogs/journey-of-javascript-downloading-scripts-to-execution-part-i
- https://www.telerik.com/blogs/journey-of-javascript-downloading-scripts-to-execution-part-ii
- https://softwareengineering.stackexchange.com/questions/138521/is-javascript-interpreted-by-design
- https://www.quora.com/Is-JavaScript-a-compiled-or-interpreted-programming-language
- https://www.youtube.com/watch?v=p-iiEDtpy6I
- https://2ality.com/2013/10/typeof-null.html#:~:text=In%20JavaScript,%20typeof%20null%20is,it%20would%20break%20existing%20code.&text=The%20data%20is%20a%20reference%20to%20an%20object.
- https://2ality.com/2013/01/categorizing-values.html
- https://developer.mozilla.org/en-US/docs/Glossary/Hoisting
- https://www.telerik.com/blogs/the-journey-of-javascript-from-downloading-scripts-to-execution-part-iii
- https://dmitripavlutin.com/variables-lifecycle-and-why-let-is-not-hoisted/
- https://www.freecodecamp.org/news/var-let-and-const-whats-the-difference/#:~:text=var%20variables%20can%20be%20updated,const%20variables%20are%20not%20initialized.
- https://262.ecma-international.org/7.0/#sec-rules-of-automatic-semicolon-insertion
- https://flaviocopes.com/javascript-automatic-semicolon-insertion/
- https://dev.to/adriennemiller/semicolons-in-javascript-to-use-or-not-to-use-2nli
- https://stackoverflow.com/questions/2846283/what-are-the-rules-for-javascripts-automatic-semicolon-insertion-asi
- https://www.youtube.com/watch?v=B4Skfqr7Dbs
- https://medium.com/@danparkk/javascript-basics-lexical-grammar-expressions-operators-and-statements-d9a61c7e71a8
- https://medium.com/launch-school/javascript-expressions-and-statements-4d32ac9c0e74
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#expressions