ADDIS ABABA INSTITUTE OF TECHNOLOGY

CENTER OF INFORMATION TECHNOLOGY AND SCIENTIFIC COMPUTING

DEPARTMENT OF SOFTWARE ENGINEERING

JavaScript in depth

Submitted by: - Yonatan Merkebu

Student Id: - ATR/4308/11

Section: 2

Submitted to: - Fitsum Alemu

January 2021

Contents

1. Is JavaScript Interpreted language in its entirety?

1.1 First of all, what is the difference between an Interpreted and compiled languages?

In an interpreted language, the source code will be read and directly executed, line by line by using interpreter program. An interpreter is a program that directly executed instructions written in a programming language, without requiring them previously to have been compiled into a machine language program. It translates on Statement at a time.

Whereas,

In a compiled language the code will typically be compiled to machine code (byte code) by the compiler before being executed. A compiler is a software that transforms computer code written in one language into a language that the machine understands.

Languages regarded as "compiled" usually produce a portable (binary) representation of the program that is distributed for later execution. With JS we distribute the source code, not the binary form so many claims that disqualifies JS from the category. In reality, the distribution model for a program's "executable" form has become drastically more varied and also less relevant over the last few decades; to the question at hand, it doesn't really matter so much anymore what form of a program gets passed around.

The real reason it matters to have a clear picture on whether JS is interpreted or compiled relates to the nature of how errors are handled and Hoisting.

Historically, interpreted languages were executed in generally a top-down and line-by-line fashion; there's typically not an initial pass through the program to process it before execution begins.

In an interpreted language, an error on line 5 of a program won't be discovered until lines 1 through 4 have already executed. But for modern JavaScript's runtime environments, this is not the case, immediately after running the program, it crashes.

An invalid command (such as broken syntax) on line 5 would be caught during the parsing phase, before any execution has begun, and none of the program would run. For catching syntax (or otherwise "static") errors, generally it's preferred to know about them ahead of any partial execution.

So, what do "parsed" languages have in common with "compiled" languages? First, all compiled languages are parsed. So, a parsed language is quite a way down the road toward being compiled already. In classic compilation theory, the last remaining step after parsing is code generation: producing an executable form.

Once any source program has been fully parsed, it's very common that its subsequent execution will, in some form or fashion, include a translation from the parsed form of the program—usually called an Abstract Syntax Tree (AST)—to that executable form.

In other words, parsed languages usually also perform code generation before execution, so it's not that much of a stretch to say that they're compiled languages.

JS source code is parsed before it is executed. The specification requires as much, because it calls for "early errors"—statically determined errors in code, such as a duplicate parameter name—to be reported before the code starts executing. Those errors cannot be recognized without the code having been parsed.

Another example is Hoisting, for example if you call a function before you define it, the JS engine will know about the function before it reaches to its declaration.

Conclusion:

So, is JS compiled language?

The answer is closer to yes than no. The parsed JS is converted to an optimized (binary) form, and that "code" is subsequently executed; the engine does not commonly switch back into line-by-line execution mode after it has finished all the hard work of parsing—most languages/engines wouldn't, because that would be highly inefficient.

2. The history of “typeof null”

The null value is technically a primitive. This would typically mean that the type of null should also be "null". However, this is not the case because of a peculiarity with the way JavaScript was first defined.

In the first implementation of JavaScript, values were represented in two parts - a type tag and the actual value. There were 5 type tags that could be used, and the tag for referencing an object was 0. The null value, however, was represented as the NULL pointer, which was 0x00 for most platforms. As a result of this similarity, null has the 0-type tag, which corresponds to an object. It examined its type tag and the type tag said “object”.

It is considered a bug in JavaScript that typeof null is an object. It should be null.

A fix was proposed for ECMAScript, but was rejected. It would have resulted in typeof null === 'null'.

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

All written JavaScript code is interpreted within the Execution Context that it is written in. When you open up your text editor and create a new JavaScript file, you create what is called a Global Execution Context.

The JavaScript engine interprets the JavaScript written within this Global Execution Context in two separate phases; compilation and execution.

3.1 Compilation

During the compilation phase, JavaScript parses the written code on the lookout for all function or variable declarations.

When compiling these keywords, JavaScript creates a unique space in memory for each declared variable it comes across. This process of giving a variable a space in memory is called hoisting.

Typically, hoisting is described as the moving of variable and function declarations to the top of their (global or function) scope.

However, the variables do not move at all.

What actually happens is that during the compilation phase declared variables and functions are stored in memory before the rest of your code is read.

3.2 Execution

After all the declared variables have been hoisted, the interpreter starts assigning variables values and processing functions.

When variables are in the hoisting process, they differ in their initialization.

During the compilation phase, JavaScript variables declared with var and function are hoisted and automatically initialized to undefined.

Contrastingly, variables declared with let, const, and class are hoisted but remain uninitialized:

These variable declarations are only initialized when their assignment (also known as lexical binding) is evaluated during runtime. If you try to access these variables before they are initialized, you will get a reference error because the variable has not been initialized, it has not been assigned a value, and thus the reference error is returned stating that it is not defined.

It’s not an error to reference let and const variables in code above their declaration as long as that code is not executed before their declaration.

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

The reason semicolons are sometimes optional in JavaScript is because of automatic semicolon insertion, or ASI.

The principle of the feature is to provide a little mercy when evaluating the syntax of a JavaScript program by conceptually inserting missing semicolons.

A JavaScript program that parses correctly is made up of smaller statements that must match its grammar rules. Each statement that makes up a program is separated by semicolons, so that when

a sequence of those statements are read from left to right, it’s easier to determine the end of a statement and the start of the next by the semicolons.

There are three basic rules for semicolon insertion:

Rule #1

A semicolon will be inserted when it comes across a line terminator or a '}' that is not grammatically correct.

Here’s an example:

```
var a
  b =
  3;
```

A semicolon is inserted at the end of the first line, since the grammar rule didn’t match (i.e. it didn’t expect to encounter b immediately after a).

Rule #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.

Rule #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.

Expressions in a return statement should begin on same line!

If you put a line break where there shouldn't be one, ASI may jump in and assume a semicolon even if there shouldn't be one. For this reason and also it is hard to debug without semi-colons it's probably good to use semi-colons or at least know what ASI is doing.

Here are a few cases where you don't need semicolons:

```
if (...) {...} else {...}

for (...) {...}

while (...) {...}
```

Note: You do need one after: do{...} while (...);

5. Expression vs Statement in JavaScript?

An expression produces a value and can be written wherever a value is expected.

A statement performs an action.

Expressions have an analog, the conditional operator. The above statements are equivalent to the following statement.

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

The code between the equal sign and the semicolon is an expression.

5.1 Expressions

JavaScript has the following expression categories.

Arithmetic Expressions:

Arithmetic expressions evaluate to a numeric value.

String Expressions:

String expressions are expressions that evaluate to a string.

Logical Expressions:

Expressions that evaluate to the Boolean value true or false.

Primary Expressions:

Primary expressions refer to stand alone expressions such as literal values, certain keywords and variable values.

Assignment Expressions:

When expressions use the = operator to assign a value to a variable, it is called an assignment expression.

5.2 Statements

Statements in JavaScript can be classified into the following categories:

Declaration Statements:

Such type of statements creates variables and functions by using the var and function statements respectively.

Expression Statements:

Wherever JavaScript expects a statement, you can also write an expression. Such statements are referred to as expression statements. But the reverse does not hold. You cannot use a statement in the place of an expression.

Conditional Statements:

Conditional statements execute statements based on the value of an expression.

Function expression versus function declaration

Function declaration (function statement) defines a function with the specified parameters.

Function declarations are statements as they perform the action of creating a variable whose value is that of the function.

Function declarations are hoisted to the top of the code unlike function expressions. Function declarations must always be named and cannot be anonymous. Furthermore, only a function expression can be immediately invoked, but not a function declaration.

Function declarations in JavaScript are hoisted to the top of the enclosing function or global scope. This doesn’t apply for function expressions.

Reference

1. Medium, https://medium.com/@almog4130/javascript-is-it-compiled-or-interpreted-9779278468fc Jan 22, 2021
2. Green Roots, https://blog.greenroots.info/javascript-interpreted-or-compiled-the-debate-is-over-ckb092cv302mtl6s17t14hq1j Jan 22, 2021
3. https://2ality.com/2013/10/typeof-null.html Jan 23, 2021
4. MDN, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof#typeof_null Jan 23, 2021
5. Medium, https://medium.com/javascript-in-plain-english/how-hoisting-works-with-let-and-const-in-javascript. Jan 23, 2021
6. Dev.to, https://dev.to/adriennemiller/semicolons-in-javascript-to-use-or-not-to-use-2nli Jan 23, 2021
7. http://www.bradoncode.com/blog/2015/08/26/javascript-semi-colon-insertion/ Jan 23, 2021
8. Liferay Community, https://2ality.com/2012/09/expressions-vs-statements.html Jan 24, 2021
9. MDN, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function Jan 24, 2021
10. Medium, https://medium.com/launch-school/javascript-expressions-and-statements-4d32ac9c0e74 Jan 24, 2021