ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

Frequently Asked JavaScript Questions

Barok Dagim

IT

Jan 24, 2021

Contents

Is JavaScript Compiled or Interpreted?

Before we discuss whether JavaScript is compiled or interpreted, let's first define what each term means via an analogy. Let's say that we are stranded on an alien planet. Our spaceship needs fixing and the only way that can happen is if we hire a local mechanic. But therein, lies a problem. The instruction manual that we have is written in a language that the mechanic doesn't understand. Logically, our next step is to hire a translator and we have a choice of either hiring an interpreter or a compiler. An interpreter would take the list of instructions from us, read the first instruction, translate it to the mechanic, wait for the mechanic to execute the step, and then keep repeating this process till the end of the document. Conversely, if we hire the compiler, it would take our list of instructions, immediately translate the entire page and give it back to us. We can then give the translated list to the mechanic who will quickly start working on the spaceship.

There are drawbacks to both. The interpreter, since it translates line by line, takes too much time between steps but it gives us a chance to quickly correct our instructions if we notice any mistakes. In the case of the compiler, the execution step will be a lot faster, as the mechanic doesn't have to wait to receive instructions; however, if there's a mistake in our instruction, we can't fix it on the fly. This is the case in programming, where the stranded pilot is the user, the interpreter is an engine (a program that executes source code) and the mechanic is the computer.

Now that we've got the definitions out of the way, let's see which category JavaScript fits in. There is a general consensus that JavaScript like most scripting languages is an interpreted language. Although that was the case when JavaScript was first implemented, it has changed over time. Nowadays, it is JIT-compiled (linked article describes it in detail) to native machine code in all major JavaScript implementations. Exactly when it's compiled to machine code varies based on implementation. In the current V8 (used in Chrome and Node.js), it starts out using an interpreter since there is little reason to spend time compiling code that only runs once. However, if a function gets executed more than a couple of times, it's immediately compiled into optimized native machine code. There is no way a JavaScript engine could ever hope to compete with other JavaScript implementations without compiling to machine code as modern JavaScript is really quite fast.

In conclusion, JavaScript, although it was initially implemented as an interpreter, is currently closer to a compiler than an interpreter.

The History of typeof null

The typeof operator in javascript takes the operand that is passed to it, and lets the user know what type the operand is from the 9 Data and Structure types that the latest ECMAScript standard defines. For example, if we run the command 'typeof(3)' it will return "number", if we change the operand to 'Hello' it will return "string" and so on and so forth. But when the statement typeof(null) is run on JavaScript, we get 'object'. This is a bug in the system, as null is a primitive value and not an object.

This bug is a remnant from the first version of JavaScript, where values were stored in 32-bit units, which consisted of a small type tag and the actual data of the value. Initially, there were five types of type tags:

- 000: object. The data is a reference to an object.
- 1: int. 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.

It was easy to assign types to most values bar for two -:

- undefined (JSVAL_VOID) was the integer -2^{30} (a number outside the integer range).
- null (JSVAL_NULL) was the machine code NULL pointer. Or: an object type tag plus a reference that is zero.

Since null was represented as the NULL pointer (0x00 in most platforms), it has 0 as type tag, and therefore, typeof(null) returns 'object'. There have been attempts to fix this bug. One proposal was implemented on Chrome's V8 engine, but it was found that the change broke a lot of existing code and had to be rejected.

In the grand scheme of things, this bug doesn't have a big enough influence other than semantics and it's likely that it will remain unfixed for a long time.

How hoisting is different with let and const

To begin, let's start by describing the term 'hoisting'. Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope (to the top of the current script or the current function). This allows us to use a variable before it is declared. This applies to all variables defined with the keyword 'var'. For variables which have been defined with either the let or const keywords, hoisting doesn't work. 'let' and 'const' were introduced in ECMAScript2015 to provide block scope variables. This just means that variables defined with either of those keywords are accessible only in the code block that they are declared in, with the difference being that 'let' variables can be reassigned, but 'const' variables can't.

Variables defined with 'let' are hoisted to the top of the block, but not initialized. This means that although the block of code is aware of the variable, it can't use it until it has been declared. This will result in a ReferenceError.

Variables defined with 'const' are similarly hoisted to the top of the block, and not initialized. But when we try to run a 'const' variable before it is declared, a syntax error occurs and the code will simply not run.

Should we use semicolons in JavaScript?

Semicolons are sometimes optional in JavaScript because of a feature called Automatic Semicolon Insertion (ASI). ASI isn't a system that literally inserts semicolons into the source code, but is more of a set of rules used by JavaScript that will determine whether or not a semicolon will be interpreted in certain spots.

Why do programmers want to omit optional semicolons?

From the sources I've read, so far, I haven't come across a good enough reason to omit semicolons. Some say omitting semicolons allows them to type less characters, others state that since there are some situations (very rare), where the rules to insert the semicolons are confusing and some omit them just for the sake of aesthetics. There are some cases, however, where a company requires it's employees to omit semicolons for specific reasons.

If you want to omit semicolons from your code, make sure to know, by heart, the rules of ASI. This will enable you to write code that is less likely to cause errors.

You should be warned, however, that there are some disadvantages to omitting semicolons:

- Minification, compression, etc. on otherwise valid JS code could cause unforeseen errors if your code doesn't use semicolons
- ASI has the potential to make your invalid JS code valid gibberish, making it harder to debug since it (sometimes) doesn't have an outright syntax error.

It is also worth mentioning, that omitting semicolons doesn't result in any improvements in either performance or file size.

In conclusion, there is no real reason to omit semicolons, but if you want to type in a few less characters or if aesthetics is important to you, make sure to understand what ASI is actually doing.

JavaScript Expressions vs. Statements

Expressions are any unit of code that can be evaluated to a value. According to the MDN documentation, JavaScript has the following expression categories:

Arithmetic Expressions: expressions that evaluate to a numeric value.

String Expressions: expressions that evaluate to a string.

Logical Expressions: expressions that evaluate to the Boolean value true or false. These expressions often involve logical operators (&&, ||, and !).

Primary Expressions: these refer to stand alone expressions such as literal values, certain keywords and variable values. Examples include:

```
'hello world';

23;

true;

sum;

this;
```

Assignment Expressions: when expressions use the '=' operator to assign a value to a variable, it is called an assignment expression. Examples include:

```
Average = 55;
```

Left-hand-side Expressions: also known as lvalues, these expressions are those that can appear on the left side of an assignment expressions. Examples of left-hand-side expressions include:

```
i = 10;

total = 0;
```

In the above two examples, the variables 'I' and 'total' are the lvalues.

Expressions with side effects: are expressions that result in a change of value of a variable through the assignment operator, function call, incrementing or decrementing the value of a variable.

```
sum = 20;
```

```
sum++;
```
Increments the value

```
function modify(){
    a *= 10;
}

var a = 10;
modify();
```
Modifies the value of through a function

Statements are instructions to perform a specific action. Such as creating a variable, looping through an array, evaluating code based on conditions, etc. JavaScript programs are a sequence of statements.

Statements in JavaScript can be classified into the following categories:

Declaration Statements: the types of statements create variables and functions by using the var and function statements respectively.

```
var sum;
var average;

var total = 0;
function greet(message) {
    console.log(message);
}
```

Expression Statements: Wherever JavaScript expects a statement, we can write an expression. Such statements are referred to as expression statements. But we can’t use a statement in the place of an expression.

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

statement used in place of expression, results in an error `var a = (b = 1);` an assignment expression, not a statement

`console.log(var a);` results in error, because we tried to pass a statement instead of an expression

Conditional Statements: execute statements based on the value of an expression. Includes if…else and switch statements.

Loops and Jumps:

Looping statements include: while, do/while, for, for/in.

Jumping statements include: break, continue, return and throw.

Function Expressions vs. Function Declarations:

A **Function Expression** is part of a variable assignment expression and may or may not contain a name. Since this type of function appears after the assignment operator '=', it is evaluated as an expression. Function expressions are evaluated only when the interpreter reaches the line of code where function expressions are located.

```
var num = function message(x) {
                return x + x;
          }
num(7);
```

In the snippet above we assign a function to the variable num and use it to call the function. The expression 'num(7)' returns 14. Only function expressions can be immediately invoked. Such types of function expressions are referred to as Immediately Invoked Function Expression (IIFE).

Function Declarations are statements as they perform the action of creating a variable whose value is that of the function. Function declaration falls under the category of declaration statements. Function declarations, unlike function expressions are hoisted to the top of the code. In addition, function declarations must always be named and can’t be anonymous.

```
function greet(message) {
    return “Hi ” + message;
}
```

References

- Interpreters and Compilers (Bits and Bytes, Episode 6)
- JavaScript — is it Compiled or Interpreted?
- Is JavaScript a compiled or interpreted programming language?
- JavaScript engine
- The history of “typeof null”
- typeof
- var vs let vs const in JavaScript
- JavaScript Hoisting
- JavaScript Expressions and Statements
- ASI Guide – Are Semicolons Necessary in JavaScript?
- Semicolons in JavaScript: To Use or Not to Use?