ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

CENTER OF INFORMATION TECHNOLOGY AND SCIENTIFIC COMPUTING

JAVA SCRIPT BASICS

Prepared by: - Mercy Habte

Submitted to: - Mr. Fitsum Alemu

January 2021

Table of Contents

Is JavaScript Interpreted Language in it entirely?

In a compiled language the target machine directly translates the program. In an interpreted language the source code is not directly translated by the target machine. Instead a different program (the interpreter) reads and executes the code.

Compiled languages are converted directly into machine code that the processor can execute. As a result they tend to be faster and more efficient to execute than interpreted languages. While Interpreters run through a program line by line and execute each command.

So which one is JavaScript?

According to most of the internet, JavaScript is an interpreted language, but that's not necessarily true. For example in the code below

```
console.log('Happy                                                    Sunday');
whoo hoo;
```

In theory, an interpreter would read the first line print "Happy Sunday" and only then throw a Syntax Error. But for modern JavaScript's runtime environments this is not the case, immediately after running the program, before executing the log function, it crashes.
Considering another example:

```
max(10, 15);
function max(num1, num2){
  return num1 > num2 ? num1 : num2;
}
```

If java script was fully an interpreted language it wouldn't know about the 'max' Function before it "reaches" to the deceleration. So it could be said that Java script is a little bit of both compiled and interpreted as well.

The History of "type of null"

In the first implementation of JavaScript values were represented in two parts - a type tag and the actual value. There were five 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.

The null value is technically a primitive, the way "object" or "number" are primitives. 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 JavaScript type of null is 'object', which incorrectly suggests that null is an object. This is a bug and one that unfortunately can't be fixed, because it would break existing code. It all goes back to the first version of JavaScript when the "type of" operator was born. Back then the bits were stored in a different manner than in today's version of JS and because of that the "type of" operator is how it is today.

JavaScript has two kinds of types: primitives (strings, Booleans, numbers, symbols) and objects. Objects are complex data structures. The simplest object in JavaScript is the plain object which is a collection of keys and associated values. But there are situations when an object cannot be created. For such cases JavaScript provides a special value null — which indicates a missing object.

Null is a primitive value that represents the intentional absence of any object value. The good way to check for null is by using the strict equality operator (===).

```
const missingObject = null;

const existingObject = { message: 'Hello!' };

missingObject  === null;

// true existingObject === null;

// false missingObject === null
```

Evaluates to true if missing Object variable contains a null value and if the variable contains a non-null value, like an object, the expression existingObject === null evaluates to false.

Why hoisting is different with let and const?

During compile phase just microseconds before the code is executed, it is scanned for function and variable declarations. All these functions and variable declarations are added to the memory inside a JavaScript data structure called Lexical Environment (a data structure that holds identifier-variable mapping where identifier here refers to the name of variables/functions, and the variable is the reference to actual object [including function object] or primitive value).

So that they can be used even before they are actually declared in the source code. A lexical environment is a place where variables and functions live during the program execution.

```
console.log(a);
let a = 3;
```

the above example's output is going to be
ReferenceError: a is not defined

All declarations (function, var, let, const and class) are hoisted in JavaScript, 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 the programmer can't access the variable before the engine evaluates its value at the place it was declared in the source code. This is called "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).

Semicolons in JavaScript: To Use or Not to Use?

Semi colons are optional in java Script. JavaScript does not strictly require semicolons. When there is a place where a semicolon is needed, it adds it behind the scenes. The process that does this is called **Automatic Semicolon Insertion (ASA)**.

ASI doesn't mean that actual semicolons are inserted into the code, it's more of a set of rules used by JavaScript that will determine whether or not a semicolon will be interpreted in certain spots

Three Automatic Semicolon Insertion Rules:

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.

And Additionally:

- when there is a break statement on its own line
- when there is a throw statement on its own line
- when there is a continue statement on its own line

When Should Semicolons not be used?

Here are a few cases where semicolons are not required:

- if else statements
- for loops
- while loops

But: Semi colons are required in do while loops.

Expression vs Statement in JavaScript

Any unit of code that can be evaluated to a value is an expression while a statement is an instruction to perform a specific action. Wherever JavaScript expects a statement, an expression can be written. Such a statement is called an **expression statement**. The reverse does not hold a statement can't be written where JavaScript expects an expression.

Whether something is an expression or a statement cannot (in the general case) be determined by looking at a textual piece of code out of context; rather it is a property of a node in a syntax tree and can be decided only after the code is (mentally or actually) parsed.

For example, an if statement cannot become the argument of a function. In programming language terminology, an "**expression**" is a combination of values and functions that are combined and interpreted by the compiler to create a new value, as opposed to a "**statement**" which is just a standalone unit of execution and doesn't return anything.

The easiest way to figure out statements in JavaScript is to look for the semicolon sign. Here are some comparisons of syntaxes of statements and expressions:

If statement versus conditional operator

- The following is an example of an if statement:

```
var x;
   if (y >= 0) {
      x = y;
   } else {
      x = -y;
   }
```

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

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

Semicolon versus comma operator

- In JavaScript, one uses the semicolon to chain statements:
  foo(); bar()

- For expressions, there is the lesser-known comma operator:
  foo(), bar()

  That operator evaluates both expressions and returns the result of the second one.

Function expression versus function declaration

The code below is a function expression:

function () { }

A function expression can also be given a name and be turned into a named function expression:

function foo() { }

A named function expression is indistinguishable from a function declaration (which is, roughly, a statement). But their effects are different: A function expression produces a value (the function). A function declaration leads to an action – the creation of a variable whose value is the function. Furthermore, only a function expression can be immediately invoked, but not a function declaration.

Using object literals and function expressions as statements

Some expressions are indistinguishable from statements. That means that the same code works differently depending on whether it appears in an expression context or a statement context. Normally the two contexts are clearly separated. However, with expression statements, there is an overlap. There expressions appear in a statement context. In order to prevent ambiguity the JavaScript grammar forbids expression statements to start with a curly brace or with the keyword function:

Expression Statement:

[lookahead ∉ {"{", "function"}] Expression ;

In order to write an expression statement that starts with either of those two tokens one can put it in parentheses, which does not change its result, but ensures that it appears in an expression-only context.

For example :

eval parses its argument in statement context. For eval to return an object, parentheses has to be put around an object literal.

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

References

- https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/
- https://medium.com/@almog4130/javascript-is-it-compiled-or-interpreted-9779278468fc
- https://www.tutorialspoint.com/Why-java-is-both-compiled-and-interpreted-language
- https://bitsofco.de/javascript-typeof/
- https://2ality.com/2013/10/typeofhttps://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%20objectnull.html#:~:text=In%20JavaScript%2C%20typeof%20null%20is,it%20would%20break%20existing%20code.&text=The%20data%20is%20a%20reference%20to%20an%20object.
- https://dmitripavlutin.com/javascript-null/#2-how-to-check-for-null
- https://blog.bitsrc.io/hoisting-in-modern-javascript-let-const-and-var-b290405adfda
- https://medium.com/javascript-in-plain-english/how-hoisting-works-with-let-and-const-inhttps://medium.com/javascript-in-plain-english/how-hoisting-works-with-let-and-const-in-javascript-725616df7085javascript-725616df7085
- https://dev.to/adriennemiller/semicolons-in-javascript-to-use-or-not-to-use-2nli https://flaviocopes.com/javascript-automatic-semicolon-insertion/
- https://2ality.com/2012/09/expressions-vs-statements.html
- https://stackoverflow.com/questions/12703214/javascript-difference-between-a-statement-and-an-expression