ADDIS ABABA INSTITUTE OF TECHNOLOGY

CENTER OF INFORMATION TECHNOLOGY AND SCIENTIFIC COMPUTING

DEPARTMENT OF SOFTWARE ENGINEERING

Basics of JavaScript

Submitted by: - Naboni Abebe

Student Id: - ETR/0714/11

Section: 1

Submitted to: - Fitsum Alemu

January 2021

Table of Contents

1. Is JavaScript Interpreted Language in its entirety?

1.1 What is the Difference between Interpreter and Compiler?

Before answering whether JavaScript is an Interpreted or Compiled language, Lets first differentiate between Interpreter and Compiler.

Compliers and interpreters are programs that help convert the high-level language (Source Code) into machine codes to be understood by the computers. Computer programs are usually written on high level languages. A high-level language is one that can be understood by humans. To make it clear, they contain words and phrases from the languages in common use – English or other languages for example. However, computers cannot understand high level languages as we humans do. They can only understand the programs that are developed in binary systems known as a machine code. To start with, a computer program is usually written in high level language described as a source code. These source codes must be converted into machine language and here comes the role of compilers and interpreters.

1.1.1 Differences between Interpreter and Compiler

Interpreter translates just one statement of the program at a time into machine code.	Compiler scans the entire program and translates the whole of it into machine code at once.
An interpreter takes very less time to analyze the source code. However, the overall time to execute the process is much slower.	A compiler takes a lot of time to analyze the source code. However, the overall time taken to execute the process is much faster.
An interpreter does not generate an intermediary code. Hence, an interpreter is highly efficient in terms of its memory.	A compiler always generates an intermediary object code. It will need further linking. Hence more memory is needed.
Keeps translating the program continuously till the first error is confronted. If any error is spotted, it stops working and hence debugging becomes easy.	A compiler generates the error message only after it scans the complete program and hence debugging is relatively harder while working with a compiler.
Interpreters are used by programming languages like Ruby and Python for example.	Compliers are used by programming languages like C and C++ for example.

Now that we got a handle on the difference between Interpreter and Compiler, Lets try to conclude what JavaScript is categorized under.

1.2 What created the confusion?

Well, even MDN clearly says that JavaScript is an interpreted language (it also says JIT-compiled which I will address later in the article). Still there is a question that if JavaScript is really interpreted because of the following points.

- If interpreted then how does hoisting takes place?
- JIT (just-in-time compiler) makes code optimizations (also create compiled versions); interpreted languages can never do that.

1.2.1 What is Hoisting?

Hoisting was thought up as a general way of thinking about how execution contexts (specifically the creation and execution phases) work in JavaScript.

Below is the way how declarations are handled in JavaScript.

- Whenever v8 enters the execution context of a certain code (function); it starts by lexing or tokenizing the code. Which mean it will split your code into atomic tokens like foo = 10.
- After analyzing the entire current scope, it parses a translated version of into an AST (for Abstract Syntax Tree).
- Each time it encounters a declaration, it sends it to the scope to create the binding. For each declaration it allocates memory for that variable. Just allocates memory, doesn't modify the code to push the declaration up in the codebase. And as you know, in JS, allocating memory means setting the default value undefined.
- After that, each time it encounters an assignment or an evaluation, it asks the scope for the binding. If not found in the current scope, it goes up into parent scopes until it finds it.
- Then it generates the machine code that the CPU can execute.
- Finally, the code is executed.

So hoisting is nothing but the game of execution context and not code modification, unlike many websites describe it. Before executing any expression, the interpreted has to find the value of the variables from the scope which was already there since execution context was created.

1.2.2 What is JIT

JIT or just in time compilers are not specific to JavaScript. Other languages like Java also have these kinds of mechanism to compile the code just before the execution.

The modern JavaScript engines also has JIT. Yes, they have a compiler. Now let me explain you why they need JIT and how it works in JavaScript execution.

The most important differences between a compiled and an interpreted language is; the compiled one takes a longer time to prepare itself to start executing, as it has to take care of lexing the entire codebase, making awesome optimizations etc. In the other hand an interpreted language starts executing in no time but doesn't do any optimization of code. So, each expression is translated separately. Consider the code snippet below.

```
for(i=0; i<1000; i++){
   sum += i;
}
```

In case of compiled language, the sum += i part was already compiled down to machine code and when the loop will run, the machine code will be executed 1000 times. But, in case of interpreted language, it will translate the sum += i 1000 times to machine code and execute. So, there's a huge performance drop cause the same code is getting translated 1000 times. This is why the Google and Mozilla people brought JIT into the picture in case of JavaScript.

1.3 Conclusion

A program must be translated so it's understood by a computer before we can run it. JavaScript code is executed by a JavaScript engine. The engine makes sure that what you've written is understood by the machine.

- An interpreter does this during runtime and executes statement by statement.
- A compiler translates beforehand and requires more time, but this allows the compiler to optimize and give us a fast execution later on when we run the code.

The first JavaScript engines were only interpreters. As JavaScript became, more commonly used, the loss of performance caused by interpretation (amongst other things of course) gave birth to new engines.

These modern JavaScript engines use a JIT (just-in-time) compilation. A just-in-time compiler doesn't compile the same way a compiler compiles for example C++. Compilers for other languages often have lots of time to optimize during compilation, but just as the name implies, that's not the case with JIT (just-in-time) compilation. Instead, just about when the JavaScript code is supposed to run, it gets compiled to executable bytecode.

So, unlike other programming languages like Java, the compilation doesn't take place at the build time. The three phases described above are not the only things that happen to Compile JavaScript Source code. JavaScript engine needs to perform lots of optimization steps to tackle performance issues. In Conclusion, JavaScript is a Compiled Language.

2. The history of “typeof null”

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.

2.1 What is typeof operator?

The **typeof** operator in JavaScript evaluates and returns a string with the data type of an operand.

For example,

Table 1: typeof operator in JavaScript

Type	Result
Undefined	"undefined"
Null	"object" (see below)
Boolean	"boolean"
Number	"number"
BigInt (new in ECMAScript 2020)	"bigint"
String	"string"
Symbol (new in ECMAScript 2015)	"symbol"
Function object (implements [[Call]] in ECMA-262 terms)	"function"
Any other object	"object"

2.2 Why is typeof Null an object?

In the first implementation of JavaScript, JavaScript values were represented as a type tag and a value. The type tag for objects was 0. null was represented as the NULL pointer (0x00 in most platforms). Consequently, null had 0 as type tag, hence the typeof return value "object".

The “typeof null” bug is a remnant from the first version of JavaScript. In this version, values were stored in 32 bit units, which consisted of a small type tag (1–3 bits) and the actual data of the value. The type tags were stored in the lower bits of the units. There were five of them:

- 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.

The main idea is that the code assigned each item some bits for use as flags for different types, but *null* was different. Objects had a flag of 000, so an object's last 3 bits were 000. *null* was previously defined as 32 0 bits: 00000000000000000000000000000000. When the code tried to check *null*'s flag, the last three bits of *null* (000) matched the Object flag (000), so it was incorrectly determined to be an object.

This bug won't go away in the foreseeable future as it will break the existing code that relies exactly on this principle, which means that every web application out there will need to undergo a refactoring.

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

3.1 How Hoisting works

During compile phase, just microseconds before your 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 (***lexical environment* is a place where variables and functions live during the program execution.**)**. So that they can be used even before they are actually declared in the source code.

Consider the following code:

```
sayHi();
function sayHi() {
  console.log('Hi there!');
}
```

It works and prints 'Hi there!'. Even though the function is defined after it is used. Or another example:

```
john = 'John Doe';
console.log(john); //John Doe
var john;
```

Variable john is declared after it is used, yet it still works. How is this possible? When your JavaScript code is being processed, in the first iteration, before actually executing it line by line, all the variable and function declarations are detected. Then they are created in memory and space is allocated for them. Only after that, the code is executed line by line. This behaves exactly the same as if the declarations were moved on top of the scope (e.g., function body). That means your

code behaves as if the declarations of variables and functions were first and then the rest of the code. There is one caveat though.

Consider the following line:

```
var john = 'John Doe';
```

It is actually consisting of two parts:

- *var john* means that variable john is declared
- *= 'John Doe';* means that previously declared variable *john* is assigned a value of string 'John Doe'

It is basically one-liner for:

```
var john; // declaration
john = 'John Doe'; //initialization
```

The thing is that only declarations are hoisted, not initializations. That means you can access variable *john* before it is actually declared but its value will be undefined.

```
console.log(john); //undefined
var john = 'John Doe';
console.log(john); //'John Doe'
```

Another thing to watch for is that while function declarations and variable declarations using *var* keyword are hoisted, class declarations are not. You cannot use class before it is declared.

```
var john = new Person('John', 'Doe'); //ReferenceError: Person is not defined

class Person {
   constructor(name, surname) {
      this.name = name;
      this.surname = surname;
   }
}

var jane = new Person('Jane', 'Doe'); //No problems here
```

```

```

3.2 Are variables declared with let and const hoisted?

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.

So, variables declared with let and const are hoisted. Where they differ from other declarations in the hoisting process is in their initialization.

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

```
console.log(name) // undefined
var name = "Andrew";
```

In the above example, JavaScript first runs its compilation phase and looks for variable declarations. It comes across var name, hoists that variable and automatically assigns it a value of undefined.

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

```
console.log(name); // Uncaught ReferenceError: name is not defined
let name = "Andrew";
```

These variable declarations only become initialized when they are evaluated during runtime. The time between these variables being declared and being evaluated is referred to as the **temporal dead zone** (A time span between variable creation and its initialization where they can't be accessed.). If you try to access these variables within this dead zone, you will get the reference error above.

To walk through the second example, JavaScript runs its compilation phase and sees let name, hoists that variable, but does not initialize it. Next, in the execution phase, console.log() is invoked and passed the argument name.

Because the variable has not been initialized, it has not been assigned a value, and thus the reference error is returned stating that name is not defined.

3.3 Conclusion

Conceptually, hoisting suggests that variables and function declarations are physically moved to the top of your code. Technically, what happens is that the variable and function declarations are put into memory during the compilation phase but stay exactly where you typed them in your code.

The primary importance of hoisting is that it allows you to use functions before you declare them in your code.

The key things to take out of the definition for hoisting are

- What gets moved around is variable and function declarations. Variable assignments or initialization are never moved around.
- The declarations are not exactly moved to the top of your code; instead, they are put into memory.

In JavaScript, all variables defined with the var keyword have an initial value of undefined. This is due to hoisting which puts the variable declarations in memory and initializes them with the value of undefined. This behavior can be shown with the following example

```
console.log(x); // prints undefined
console.log(y); // throws ReferenceError: y is not defined
var x = 1;
```

However, variables defined with let or const keywords when hoisted are not initialized with a value of undefined. Rather, they are in a state called the Temporal Dead Zone and are not initialized until their definitions are evaluated.

```
console.log(x); //throws TDZ ReferenceError: x is not defined
let x = 1;
```

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

JavaScript does not strictly require semicolons. When there is a place where a semicolon was needed, it adds it behind the scenes. The process that does this is called Automatic Semicolon Insertion.

4.1 The rules of JavaScript Automatic Semicolon Insertion (ASI)

The JavaScript parser will automatically add a semicolon when, during the parsing of the source code, it finds these particular situations:

1. when the next line starts with code that breaks the current one (code can spawn on multiple lines)
2. when the next line starts with a }, closing the current block
3. when the end of the source code file is reached
4. when there is a return statement on its own line
5. when there is a break statement on its own line
6. when there is a throw statement on its own line

7. when there is a continue statement on its own line

Examples of code that does not do what you think. Based on those rules, here are some examples.

```
const hey = 'hey'
const you = 'hey'
const heyYou = hey + ' ' + you

['h', 'e', 'y'].forEach((letter) => console.log(letter))
```

You'll get the error Uncaught TypeError: Cannot read property 'forEach' of undefined because based on rule 1 JavaScript tries to interpret the code as

```
const hey = 'hey';
const you = 'hey';
const heyYou = hey + ' ' + you['h', 'e', 'y'].forEach((letter) => console.log(letter))
```

Example:

```
(1 + 2).toString()
```

prints "3".

```
const a = 1
const b = 2
const c = a + b
(a + b).toString()
```

instead raises a TypeError: b is not a function exception, because JavaScript tries to interpret it as

```
const a = 1
const b = 2
const c = a + b(a + b).toString()
```

Another example based on rule 4:

```
(() => {
  return
  {
    color: 'white'
  }
})()
```

You'd expect the return value of this immediately-invoked function to be an object that contains the color property, but it's not. Instead, it's undefined, because JavaScript inserts a semicolon after return.

Instead, you should put the opening bracket right after return:

```
(() => {
  return {
    color: 'white'
  }
})()
```

Example:

```
1 + 1
-1 + 1 === 0 ? alert(0) : alert(2)
```

You'd think this code shows '0' in an alert but it shows 2 instead, because JavaScript per rule 1 interprets it as:

```
1 + 1 -1 + 1 === 0 ? alert(0) : alert(2)
```

4.2 When do you need Semicolon?

The semicolon in JavaScript is used to separate statements, but it can be omitted if the statement is followed by a line break (or there's only one statement in a {block}).

var i = 0; i++	semicolon obligatory (but optional before newline)
var i = 0	semicolon optional
i++	semicolon optional

A statement is a piece of code that tells the computer to do something. Here are the most common types of statements:

var i;	variable declaration
i = 5;	value assignment
i = i + 1;	value assignment
i++;	same as above
var x = 9;	declaration & assignment
var fun = function() {...};	var decl., assignmt, and func. defin.
alert("hi");	function call

All of these statements can end with a ; but none of them must. Some consider it a good habit to terminate each statement with a ; – that makes your code a little easier to parse, and to compress: if you remove line breaks you needn't worry about several statements ending up unseparated on the same line.

5. Expression vs Statement in JavaScript?

5.1 What is Expression?

Any unit of code that can be evaluated to a value is an expression. Since expressions produce values, they can appear anywhere in a program where JavaScript expects a value such as the arguments of a function invocation.

Wherever JavaScript expects a statement, you can also write an expression. Such a statement is called an expression statement. The reverse does not hold: you cannot write a statement where JavaScript expects an expression. For example, an if statement cannot become the argument of a function.

Expression	
Arithmetic Expression	Numeric value E.g. 10; 10+13;
String Expression	String value E.g. ‘hello’; ‘hello’ + ‘world’;
Logical Expression	Boolean value E.g. 10 > 9; true; a===20 && b===30;
Primary Expression	Literal values, certain keywords and variable values E.g. true; sum; this;

Left-hand-side Expression	That can appear on the left side of an assignment expression
	E.g. i = 10; // variable i
	var obj = {} // properties of objects
	array[0] = 20; // elements of arrays
	++(a+1); // invalid left hand-side errors

Assignment Expression	When = is is used to assign a value to a variable E.g. average = 55;
Expression with side effects	Setting or modifying the value of a variable through the assignment operator = , function call, incrementing or decrementing

	E.g. sum = 20;
	sum++; // increments value by 1
	
	function modify() {a*= 10;}
	var a = 10;
	modify(); // function call modifies a to 100

5.2 What is Statement?

When we write programs - we describe the sequences of actions that should be performed to get a desired result. In programming languages those actions are called statements. So, every JavaScript program basically consists of statements. In JavaScript statements are separated by semicolons.

A statement is an instruction to perform a specific action. Such actions include creating a variable or a function, looping through an array of elements, evaluating code based on a specific condition etc. JavaScript programs are actually a sequence of statements.

Statements	

Declaration Statement	Create variables and functions by using var and function statements
	E.g. var total = 0; // var total is statement and
	// total = 0 is an assignment expression
	function greet(message){ console.log(message); }
	// function declaration statement

Expression Statement	Wherever JavaScript expects a statement, you can also write an expression
	E.g. var a = var b; // Error, because we used a statement in place of an expression
	var a = (b = 1); // Correct, since (b = 1) is an assignment expression
	console.log(var a); // Error, we can only pass an expression as function argument

Conditional Statements	if…else and switch statements. E.g. if (expression) Statement 1 else Statement 2
Loops and Jumps	While, do/while, for and for/in are examples of Loop Statements Break, continue, return and throw are examples of Jump Statements

Reference

1. Business Insider, https://www.businessinsider.in/difference-between-compiler-and-interpreter/articleshow/69523408.cms#:~:text=Interpreter%20translates%20just%20one%20statement,the%20process%20is%20much%20slower, January 22, 2021
2. The Green Roots Blog, https://blog.greenroots.info/javascript-interpreted-or-compiled-the-debate-is-over-ckb092cv302mtl6s17t14hq1j, January 22, 2021
3. Void Canvas, https://www.voidcanvas.com/is-javascript-really-interpreted-or-compiled-language/, January 22, 2021
4. 2ality – JavaScript and more, https://2ality.com, January 22, 2021
5. Alex Ellis, https://alexanderell.is/posts/typeof-null/, January 22, 2021
6. JavaScript in plain English, https://medium.com/javascript-in-plain-english/how-hoisting-works-with-let-and-const-in-javascript-., January 22, 2021
7. Bits and Pieces, https://blog.bitsrc.io/hoisting-in-modern-javascript-let-const-and-var-b290405adfda, January 22, 2021
8. Flaviocopes, https://flaviocopes.com/javascript-automatic-semicolon-insertion/, January 23, 2021
9. Dev, https://dev.to/adriennemiller/semicolons-in-javascript-to-use-or-not-to-use-2nli, January 23, 2021
10. Medium, https://medium.com/launch-school/javascript-expressions-and-statements-4d32ac9c0e74, January 23, 2021