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, aka 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. Here, if the author decides he wants to use a different kind of olive oil, he could scratch the old one out and add the new one.
So which one is JavaScript?

Java implementations typically use a two-step compilation process. Java source code is compiled down to *bytecode* by the Java compiler. The bytecode is executed by a Java Virtual Machine (JVM). Modern JVMs use a technique called Just-in-Time (JIT) compilation to compile the bytecode to native instructions understood by hardware CPU on the fly at runtime.

Some implementations of JVM may choose to interpret the bytecode instead of JIT compiling it to machine code, and running it directly. While this is still considered an "interpreter," It's quite different from interpreters that read and execute the high level source code (i.e. in this case, Java source code is not interpreted directly, the bytecode, output of Java compiler, is.)
It is technically possible to compile Java down to native code ahead-of-time and run the resulting binary. It is also possible to interpret the Java code directly.
To summarize, depending on the execution environment, bytecode can be:

- compiled ahead of time and executed as native code (similar to most C++ compilers)
- compiled just-in-time and executed
- interpreted
- directly executed by a supported processor (bytecode is the native instruction set of some CPUs)

Java is the first substantial language which is neither truly interpreted nor compiled; instead, a combination of the two forms is used. This method has advantages which were not present in earlier languages.
Platform-Independence

To understand the primary advantage of Java, you'll have to learn about platforms. In most programming languages, a compiler (or interpreter) generates code that can execute on a specific target machine. For example, if you compile a C++ program on a Windows machine, the executable file can be copied to any other machine but it will only run on other Windows machines but never another machine (e.g., a Mac or a Linux machine). A platform is determined by the target machine (along with its operating system). For earlier languages, language designers needed to create a specialized version of the compiler (or interpreter) for every platform. If you wrote a program that you wanted to make available on multiple platforms, you, as the programmer, would have to do quite a bit of additional work. You would have to create multiple versions of your source code for each platform.

Java succeeded in eliminating the platform issue for high-level programmers (such as you) because it has reorganized the compile-link-execute sequence at an underlying level of the compiler. Details are complicated but, essentially, the designers of the Java language isolated those programming issues which are dependent on the platform and developed low-level means to abstractly refer to these issues. Consequently, the Java compiler doesn't create an object file, but instead it creates a bytecode file which is, essentially, an object file for a virtual machine. In fact, the Java compiler is often called the JVM compiler (for Java Virtual Machine).

Consequently, you can write a Java program (on any platform) and use the JVM compiler (called javac) to generate a bytecode file (bytecode files use the extension .class). This bytecode file can be used on any platform (that has installed Java). However, bytecode is not an executable file. To execute a bytecode file, you actually need to invoke a Java interpreter (called java). Every platform has its own Java interpreter which will automatically address the platform-specific issues that can no longer be put off. When platform-specific operations are required by the bytecode, the Java interpreter links in appropriate code specific to the platform.

To summarize how Java works (to achieve platform independence), think about the compile-link-execute cycle. In earlier programming languages, the cycle is more closely defined as "compile-link then execute". In Java, the cycle is closer to "compile then link-execute".
As with interpreted languages, it is possible to get Java programs to run faster by compiling the bytecode into an executable; the disadvantage is that such executables will only work on the platform in which it is created.

So it can be summarized as JavaScript being both because a java program is first compiled into bytecode which JRE can understand. Bytecode is then interpreted by the JVM making it as interpreted language.

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 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.

> 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.

JavaScript has 2 kinds of types: primitives (strings, Booleans, numbers, symbols) and objects. Objects are complex data structures. The simplest object in JavaScript is the plain object — 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*. If you see null (either assigned to a variable or returned by a function), then at that place should have been an object, but for some reason, an object wasn’t created.

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 because missingObject variable contains a null value.

If the variable contains a non-null value, like an object, the expression existingObject === null evaluates to false.

It all goes back to the first version of JavaScript when the typeof 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 typeof operator is how it is today.

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**. So that they can be used even before they are actually declared in the source code.

A *lexical environment* is a data structure that holds **identifier-variable mapping**. (here **identifier** refers to the name of variables/functions, and **the variable** is the reference to actual object [including function object] or primitive value).
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 you can't access the variable before the engine evaluates its value at the place it was declared in the source code. This is what we call "**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 was needed, it adds it behind the scenes.

The process that does this is called **Automatic Semicolon Insertion**.

The reason semicolons are sometimes optional in JavaScript is because of automatic semicolon insertion, or ASI. ASI doesn't mean that actual semicolons are inserted into your 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. I found a helpful lecture from Fullstack Academy on the topic, which you can check out here. I also found a blog post from Bradley Braithwaite on the topic. Below I highlight the main takeaways from these resources.

3 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.

The rules of JavaScript Automatic Semicolon Insertion

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

when the next line starts with code that breaks the current one (code can spawn on multiple lines)

- when the next line starts with a }, closing the current block
- when the end of the source code file is reached
- when there is a return statement on its own line
- 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 I Not Use Semicolons?

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

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

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

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.

References

- https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/
- https://medium.com/@almog4130/javascript-is-it-compiled-or-interpreted-9779278468fc
- https://stackoverflow.com/questions/1326071/is-java-a-compiled-or-an-interpreted-programming-language
- https://www.tutorialspoint.com/Why-java-is-both-compiled-and-interpreted-language
- https://bitsofco.de/javascript-typeof/
- https://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%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-in-javascript-725616df7085
- https://dev.to/adriennemiller/semicolons-in-javascript-to-use-or-not-to-use-2nli
- https://flaviocopes.com/javascript-automatic-semicolon-insertion/
-