ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

CENTER OF INFORMATION TECHNOLOGY AND SCIENTIFIC COMPUTING

Assignment I

Prepared By: Bonson Demssie Shito

Submitted To: Mr. Fitsum A.

January 25, 2021

JavaScript — is it Compiled or Interpreted?

JavaScript is an interpreted language according to many on the internet, but that's not always valid. Ok, here's why.

We all know that almost any programmer has to ask a popular question about the programming language he is studying, whether it is compiled or interpreted.

Let's describe what each of these means first.

Compiled language: The source file would usually be "compiled" into machine code (or byte code) before execution of the compiled language.

Interpreted language: the source code, line by line, will be interpreted and executed directly.

And I honestly believe, like anything in life, that you can understand it better with a recipe.

Let's presume that you'd like to make a cake you go to Google and search for the ingredients list.

There are 2 ways to make the cake, the Compiler or the Interpreter way.

The compiler would first arrange all the ingredients in front of him before doing any mixing, the exact quantities of each single ingredient, only then will he combine all the cake's ready components.

And the Interpreter would take his measuring tools and start, line by line, by reading the ingredients. He's going to go to his fridge to get eggs, crack and mix them together, etc.

The compiler's construct (preparation) period will be longer than that of the interpreters. The running (mixing) time, however, will be much shorter.

Well, like many of you, I was told that JavaScript, like most scripting languages, is an interpreted language, and made peace with this assumption when I started studying JavaScript.

JavaScript is an interpreted language, according to much of the internet, but that's not inherently true. For example, the V8 engine, the engine that runs Google Chrome and NodeJS, internally compiles to native code: V8 improves performance by compiling JavaScript to native machine code before it is executed, versus bytecode execution or interpretation to Mozilla's JavaScript® engine (known as "SpiderMonkey").

Look at this program for instance:

```
Console.log ('World of Hello ');
oops oops;
```

In principle, the first line will be read by an interpreter, printing Hello World and only then throwing a Syntax Error.

But this is not the case for modern JavaScript runtime environments; it crashes immediately after running the program, before running the log function.

So, JavaScript is a compiled language, right?

All right, it's complicated. In the past, each programming language was fairly easy to categorize as one or the other, but a kind of "in-between" field was generated by the modern approach of running the source code and created a new method to run the source code.

So, Is it Compiled then?

The answer is a matter of perspective and implementation.

The history of “typeof null”

Form null is 'object' in JavaScript, which incorrectly means that null is an object it's not, it's a primitive attribute. This is a flaw and one that can not be patched, sadly, because it will break the current code. Let's explore this bug's past.

The bug "typeof null" is a remnant of the first JavaScript version. Values were stored in 32 bit units in this version.

At that time there was very little time to finish the first version of JavaScript.

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

What are Hoisting,let and cont ?

JavaScript has the var, let and const keywords for variable declarations and with each comes a different use case. And Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope (to the top of the current script or the current function)

Example:

```
var x; // Declare x
x = 5; // Assign 5 to x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x;                     // Display x in the element.
```

The use of let and const allows us to define variables that live within the block scope.

Hoisting does not apply to let. Unlike var a let-type variable won’t be initialized to undefined before its definition gets evaluated by the parser. If you try to use a let variable before the part of the code where it’s initialized, you’ll get a ReferenceError thrown Because it was Called before Parsed.

And once const is used to declare a variable, that value cannot be changed or redeclared within that scope by reassignment.

Hoisting is an unknown or overlooked behavior of JavaScript (to many developers). Programs may contain bugs if a developer does not understand hoisting (errors). Always declare all variables at the beginning of each scope, to avoid bugs. Since this is how the code is interpreted in JavaScript, it is always a good rule.

Semicolons in JavaScript: To Use or Not to Use?

In JavaScript, semicolons are optional... except when they are not.
JavaScript has a law (Automatic Semicolon Insertion/ ASI) that will determine whether or not a semicolon will be interpreted in certain spots.

In conclusion it is Better to use semicolons because it helps avoid creating any bad habits early on and errors in debugging and If you're going to write your JavaScript without optional semicolons, it's probably good to at least know what ASI is doing.

Expression vs Statement in JavaScript?

In JavaScript, statements and expressions are two very important terms. An expression generates a value and can be written anywhere a value is required, as an argument in a function call.

Example new Date ()

while A statement is an order for carrying out a particular operation. Such actions include creating a variable or a function, evaluating code based on a specific condition etc.

Example x=5;

The one of main in both is that 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.