**Reading Assignment**

1. Is Javascript Interpreted language in it entirety ? : Make Up your justification.

Programming languages are generally either complied or interpreted.Let’s see what they mean.

**In Compiled language** — the source file typically will be “compiled” to machine code (or byte code) before being executed.

**In Interpreted language** — the source code will be read and directly executed, line by line.

Well, JavaScript is hard to categorize only under one. In my opinion , I would say it is sort of in between.According to most of the internet, JavaScript is an interpreted language, but that’s not necessarily true.

For example, look at this program:

console.log('Hello World.');  
oops oops;

In theory, an interpreter would read the first line, print “Hello World” 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.

Another example is Hoisting, consider:

min(1, 2);  
// 1

function min(num1, num2){  
 return num1 < num2 ? num1 : num2;  
}

How does the JS engine know about the ‘max’ Function before it “reaches” to the deceleration? Again, the only reasonable answer to this question is that the code must first be compiled before execution. But this doesn’t mean it is compiled either like I said in - between.

1. The history of “typeof null” .

In JavaScript, typeof null is 'object', which incorrectly suggests that null is an object (it isn’t, it’s a primitive value).This is a flaw(bug) and one that can't be patched because it will break the current code, sadly.

Let's analyze this bug's history.

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.

That is, the lowest bit was either one, then the type tag was only one bit long. Or it was zero, then the type tag was three bits in length, providing two additional bits, for four types.

Two values were special:

* undefined (JSVAL\_VOID) was the integer −230 (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.

It should now be obvious why typeof thought that null was an object: it examined its type tag and the type tag said “object”.

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

All written JavaScript is interpreted within the **Execution Context** that it is written in. When you open up your text editor and create a new JavaScript file, you create what is called a **Global Execution Context**.

The JavaScript engine interprets the JavaScript written within this Global Execution Context in two separate phases; **compilation** and **execution**.

## Compilation

## During the compilation phase, JavaScript parses the written code on the lookout for all function or variable declarations. This includes:

-let  
-const  
-class  
-var  
-function

When compiling these keywords, JavaScript creates a unique space in memory for each declared variable it comes across. This process of “lifting” the variable and giving it a space in memory is called hoisting.

Typically, hoisting is described as the moving of variable and function declarations to the top of their (global or function) scope.

However, the variables **do not** move **at all**.

What actually happens is that during the compilation phase declared variables and functions are stored in memory before the rest of your code is read, thus the illusion of “moving” to the top of their scope.

## Execution

After the first phase has finished and all the declared variables have been hoisted, the second phase begins; execution. The interpreter goes back up to the first line of code and works its way down again, this time assigning variables values and processing functions.

# Are variables declared with let and const hoisted?

Yes, 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**. 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.

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

5.Expression vs  Statement in Javascript ?