Jan,2021

ADDIS ABABA INSTITUTE OF TECHNOLOGY (AAiT)

Prepared By, Simele Geleta,Atr/9018/12

[READING ASSIGNMENT ON BASED ON LECTURE 04]

The assignment is based on lecture 04(Java script). It contains 5 questions with good explanations of the questions.

CONTENTS

1.IS JAVA SCRIPT INTERPRETED LANGUAGE IN IT ENTIRETY

Before deciding whether java script is interpreted entirely or not first let's see some points about java script.

JavaScript is primarily a client-side language. JavaScript started at Netscape, a web browser developed in the 1990s. A webpage can contain embedded JavaScript, which executes when a user visits the page. The language was created to allow web developers to embed executable code on their web pages, so that they could make their web pages interactive, or perform simple tasks. Today, browser scripting remains the main use-case of JavaScript.

JavaScript's syntax is heavily inspired by C++ and Java. If one has experience in C++ or Java, JavaScript's syntax will seem familiar. However, the inner workings of Java Script is closer to a dynamically-typed, interpreted language such as Python or Ruby.

Now let's see the difference between interpreted and compiled programming languages

Compiled Languages

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. They also give the developer more control over hardware aspects, like memory management and CPU usage.

Compiled languages need a “build” step – they need to be manually compiled first. You need to “rebuild” the program every time you need to make a change. Examples of pure compiled languages are C, C++, Erlang, Haskell, Rust, and Go.

Interpreted Languages

Interpreted languages were once significantly slower than compiled languages. But, with the development of just-in-time compilation, that gap is shrinking.
Examples of common interpreted languages are PHP, Ruby, Python, and JavaScript There is an ambiguity about whether it is compiled or interpreted entirely but in most of its behaviours we can say that JavaScript is an interpreted language, not a compiled language. It is Client side scripting language. The **client-side** environment used to run **scripts** is usually a browser. The processing takes place on the end users computer. The source code is transferred from the web server to the users computer over the internet and run directly in the browser. The **scripting** language needs to be enabled on the **client** computer. A program such as C++ or Java needs to be compiled before it is run. The source code is passed through a program called a compiler, which translates it into byte code that the machine understands and can execute. In contrast, JavaScript has no compilation step. Instead, an interpreter in the browser reads over the JavaScript code, interprets each line, and runs it. More modern browsers use a technology known as Just-In-Time (JIT) compilation, which compiles JavaScript to executable byte code just as it is about to run.

2.THE HISTORY OF “TYPE OF 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.

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.

Null and undefined in JavaScript are actually values and types created to simulate errors and keywords common in other programming languages.

When a variable is `undefined`, or uninitialized, in most programming languages it means that a space in memory has been assigned to a variable name, but the programmer has not yet done anything with that space in memory. This usually results in a compile time error.

When a variable is `null` in other programming languages, null is typically a keyword to indicate the space in memory is a pointer (reference), and that pointer is pointing to an invalid memory address (usually 0x0). This is usually used when a programmer is done using the value of a variable and wants to purposefully clear it by literally pointing it to nothing.

In JavaScript, `null` and `undefined` are values and types. Just like numbers and characters, `null` has a specific configuration of 1’s and 0’s that indicates it’s type is `null` and that it’s value is `null`. Same with `undefined`. These are used in JavaScript to act as placeholders to let the programmer know when a variable has no value.

3. EXPLAIN IN DETAIL WHY HOISTING IS DIFFERENT WITH LET AND CONST

The term hoisting is confusing. One of the first and foremost reasons people struggle to understand *hoisting* is because the term itself is somewhat misleading. The Merriam-Webster definition of the word *hoist* is “an act of raising or lifting”. This might lead one to assume that hoisting involves written code being physically rearranged somehow. This is not true. Instead, the term *hoisting* is used as a kind of simile to describe a process that occurs while the JavaScript engine interprets written JavaScript code.

HOW IS JAVASCRIPT CODE INTERPRETED?

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). 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.

WHERE CAN I REFERENCE `LET` AND `CONST`?

Again, variables declared with `let` and `const` are only initialized when their assignment (also known as lexical binding) is evaluated during runtime by the JavaScript engine.

It’s not an error to reference `let` and `const` variables in code above their declaration as long as that code is not executed before their declaration.

4. SEMICOLONS IN JAVA SCRIPT: TO USE OR NOT TO USE

Semicolons in JavaScript divide the community. Some prefer to use them always, no matter what. Others like to avoid them.

Some find it natural to avoid semicolons, the code looks better and it's cleaner to read.

This is all possible because 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**.

It's important to know the rules that power semicolons, to avoid writing code that will generate bugs because does not behave like you expect.

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:

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

5. EXPRESSION VS STATEMENT IN JAVA SCRIPT

Statements and expressions are two very important terms in JavaScript. Given how frequently these two terms are used to describe JavaScript code, it is important to understand what they mean and the distinction between the two.

Expressions

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. As per the MDN documentation, JavaScript has the following expression categories.

- Arithmetic Expressions**:** Arithmetic expressions evaluate to a numeric value.
- String Expressions: String expressions are expressions that evaluate to a string.
- Logical Expressions: Expressions that evaluate to the boolean value true or false are considered to be logical expressions. This set of expressions often involve the usage of logical operators && (AND), ||(OR) and !(NOT).
- Primary Expressions: Primary expressions refer to stand alone expressions such as literal values, certain keywords and variable values.
- Primary Expressions: Primary expressions refer to stand alone expressions such as literal values, certain keywords and variable values.
- Left-hand-side Expressions: Also known as lvalues, left-hand-side expressions are those that can appear on the left side of an assignment expression.
- Assignment Expressions: When expressions use the = operator to assign a value to a variable, it is called an assignment expression.

Statements

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 in JavaScript can be classified into the following categories

- Declaration Statements: Such type of statements create variables and functions by using the var and function statements respectively.
- Expression Statements: Wherever JavaScript expects a statement, you can also write an expression. Such statements are referred to as expression statements. But the reverse does not hold. You cannot use a statement in the place of an expression. Stand alone primary expressions such as variable values can also pass off as statements depending on the context.
- Conditional Statements: Conditional statements execute statements based on the value of an expression. Examples of conditional statements includes the if..else and switch statements.
- Loops and Jumps: Looping statements includes the following statements: while, do/while, for and for/in. Jump statements are used to make the JavaScript interpreter jump to a specific location within the program. Examples of jump statements includes break, continue, return and throw.

REFERENCES

- https://web.stanford.edu/class/cs98si/slides/overview.html#:~:text=JavaScript%20is%20an%20interpreted%20language,compiled%20before%20it%20is%20run.
- https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/#:~:text=Interpreted%20vs%20Compiled%20Programming%20Languages%3A%20What's%20the%20Difference%3F,-Every%20program%20is&text=In%20a%20compiled%20language%2C%20the,reads%20and%20executes%20the%20code.
- https://medium.com/@almog4130/javascript-is-it-compiled-or-interpreted-9779278468fc
- https://medium.com/@stephenthecurt/a-brief-history-of-null-and-undefined-in-javascript-c283caab662e
- https://bitsofco.de/javascript-typeof/
- https://medium.com/javascript-in-plain-english/how-hoisting-works-with-let-and-const-in-javascript-725616df7085
- https://flaviocopes.com/javascript-automatic-semicolon-insertion/
- https://medium.com/launch-school/javascript-expressions-and-statements-4d32ac9c0e74