ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

CENTER OF INFORMATION TECHNOLOGY AND SCIENTIFIC COMPUTING

Fundamental of Web Development

Assignment on lecture 04

Prepared By: - Yonas Assefa

Instructor: - Mr. Fitsum A.

January 2021

Table of Contents

1. Is JavaScript Interpreted Language in its entirety?

1.1. What is an Interpreted and compiled languages mean?

-Every program is a set of instructions, whether it’s very easy two line code or complex networking codes. Compilers and interpreters take human-readable code and convert it to computer-readable machine code.

-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, also known as the interpreter, reads and executes the code.

I. 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. One needs to rebuild the program every time he/she make a change.

Examples of pure compiled languages are C, C++, Erlang, Haskell, Rust, and Go.

II. Interpreted Languages

Interpreters run through a program line by line and execute each command. Interpreted languages were once significantly slower than compiled languages. But, with the development of just-in-time-compilation that gap is shrinking.

Just-in-time compilation (JIT) also known as Dynamic compilation is a method for improving the performance of interpreted programs. During execution the program may be compiled into native code to improve its performance.

Dynamic compilation has some advantages over static compilation. When running Java or C# applications, the runtime environment can profile the application while it is being run. This allows for more optimized code to be generated. If the behavior of the application changes while it is running, the runtime environment can recompile the code.

Some of the disadvantages include startup delays and the overhead of compilation during runtime. To limit the overhead, many JIT compilers only compile the code paths that are frequently used.

Traditionally there are two methods for converting source code into a form that can be run on a platform. Static compilation converts the code into a language for a specific platform. An interpreter directly executes the source code.

JIT compilation attempts to use the benefits of both. While the interpreted program is being run, the JIT compiler determines the most frequently used code and compiles it to machine code. Depending on the compiler, this can be done on a method or smaller section of code.

Examples of common interpreted languages are PHP, Ruby, Python, and JavaScript.

-The difference between compiled and interpreted languages can be seen using a food recipe analogy scenario.

- Imagine we have a cake recipe that we want to make, but it's written in ancient Ge’ez. There are two ways we, a non-ancient-Ge’ez speaker, could follow its directions.

  The first is if someone had already translated it into English or any other language we could speak. Then, we could read the English version of the recipe and make cake. Think of this translated recipe as the compiled version.

The second way is if we have someone who knows ancient Ge’ez. When we’re ready to make the cake, someone who knows both the ancient Ge’ez and English translates us each steps of the recipe line by line. In this case, that person is the interpreter for the interpreted version of the recipe.

1.2. Advantages and disadvantages

I. Advantages of compiled languages

- Programs that are compiled into native machine code tend to be faster than interpreted code. This is because the process of translating code at run time adds to the overhead, and can cause the program to be slower overall.
- They are self-contained units that are ready to be executed. Because they are already compiled into machine language binaries, there is no second application or package that the user has to keep up-to-date. If a program is compiled for Windows on an x86 architecture, the end user needs only a Windows operating system running on an x86 architecture.
- Increase program performance. Users can send specific options to compilers regarding the details of the hardware the program will be running on. This allows the compiler to create machine language code that makes the most efficient use of the specified hardware, as opposed to more generic code. This also allows advanced users to optimize a program's performance on their computers

II. Disadvantages of compiled languages

-Additional time needed to complete the entire compilation step before testing

-Platform dependence of the generated binary code

- Because a compiler translates source code into a specific machine language, programs have to be specifically compiled for OS X, Windows or Linux, as well as specifically for 32-bit or 64-bit architectures.

For a programmer or software company trying to get a product out to the widest possible audience, this means maintaining multiple versions of the source code for the same application.

I. Advantages of interpreted languages

-Interpreted languages tend to be more flexible, and often offer features like dynamic typing and smaller program size. Also, because interpreters execute the source program code themselves, the code itself is platform independent.
-They offer dynamic typing as well as dynamic scoping
- Provides an ease of debugging
- They use the evaluator reflectively like in a first order evaluation function
- They provide you with an automatic memory management.

II. Disadvantages of interpreted languages

-The most notable disadvantage is typical execution speed compared to compiled languages.

- Another downside to the interpreted programs is the fact that the executable can only be run by an interpreter.

Most programming languages can have both compiled and interpreted implementations – the language itself is not necessarily compiled or interpreted. However, for simplicity’s sake, they’re typically referred to as such.

Python, for example, can be executed as either a compiled program or as an interpreted language in interactive mode. On the other hand, most command line tools, CLIs, and shells can theoretically be classified as interpreted languages.

1.3 Is JavaScript a Compiled or an Interpreted language?

The first thing to understand, Computer doesn't understand the programming languages directly. Every programming language got its own syntax, grammar, and structure. No matter what programming languages (JavaScript, Python, Java, etc.) we are writing the code with, it has to be translated into something that the machine (Computer) understands.

The most important fact here is, how does the JavaScript source code go through the journey of becoming a machine-understandable language? JavaScript Engine performs many of the steps (in fact, more cleaner and sophisticated ways) that a typical Compiler would perform in compiling source code.

- In JavaScript, the source code typically goes through the following phases it is executed.

  I. **Tokenizing**: Breaking up a source code string into meaningful chunks called **Tokens.** For example, the source code var age 7=5; can be tokenize as var, age, =,and 7

  II. **Parsing**: Parsing is a methodology to take the array of Tokens as input and, turn it into a tree of nested elements that are understood by the grammar of the programming language. This tree is called Abstract Syntax Tree (AST).

  What is an Abstract syntax tree (AST)?

  Abstract syntax trees are data structure widely use in compilers, due to their property of representing the structure of program code. An AST is usually the result of the syntax analysis phase of a compiler. AST is not only used strictly with JavaScript environments such nodejs or the browser**,** everything started on Java world long time ago, with Netscape.

III. Code Generation: In this phase, the AST used as input and an executable byte-code is generated that is understood by the environment (or platform) where the executable code will be running. The executable bite code then refined/converted even further by the optimizing JIT (Just in Time) compiler.

1.4 Conclusion

To conclude, JavaScript code indeed gets compiled. It is closer to be compiled than Interpreted. It is compiled every time. After the compilation process produces a binary byte code, by which the JS Virtual Machine executes. But, unlike other compiled programming languages, JavaScript the compilation doesn’t take place at the build time.

- So, seeing all the above properties and explanations, we can conclude that even though JavaScript is not strictly complied or interpreted language, it exhibits the properties of both types. But it takes much of the properties of the Compiled language. So, JavaScript is rather a **Compiled language** if it is not in between.

2. The history of “typeof null”

2.1 What is typeof operator?

-Before seeing about the history of “typeof null”, it is important to discover about the JavaScript **typeof** operator.

-The typeof operator in JavaScript evaluates and returns a string with the data type of the operand. For example, to find the type of 123, we would write as

-typeof 123

This will return a string with a type of 123, which, in this case will be “number”. In addition to “number”, the type of operator, can return one of the 6 potential results, which namely are “number”, ”string”, ”boolean”, “object”, “function”, “undefined” and “symbol”.

-Apart from the above types that the typeof operator could return, there are some unexpected results. Some of them are:-

- What’s the type of [1,2,3]
  -the type of the above array is actually an object. In JavaScript arrays are technically objects, just with special abilities and behaviors.
- What’s the type of null
  -Here we come to our point. **typeof null** would actually return an **object** value.
  -the null value is technically a primitive, the way object or number are primitives. This would typically means that the type of **null** should also be **null**. However, this is not the case because of the 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 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.

2.2 Nature of “typeof null” object

-In JavaScript, typeof null is ‘object’, which incorrectly suggests that null is an object (it isn’t). This is a bug and one that unfortunately can’t be fixed, because it would break existing code.

- The reason to call it a bug is that the bug came about because in the initial implementation of JavaScript all values were stored in 32 bits units, where:

- The first 1–3 bits contain the type tag
- The remaining 29–31 bits contained the actual data

-The type tag for objects was **0**. While **null** was represented with a **NULL** pointer. Note that a **NULL** pointer does not point to an address — it simply points to nothing (hence why it is called null in the first place). But in the C standard, **NULL** is considered equal to **0.** Consequently, **null** had **0** as its type tag, and since **0** is for objects, then **typeof null** fraudulently gives us “object”.

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

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

3.1 JavaScript variable declaration

- ES2015 introduced two important new JavaScript keywords: let and const.

-These two keywords provide **Block Scope** variables (and constants) in JavaScript.

-Before ES2015, JavaScript had only two types of scope: **Global Scope** and **Function Scope**. It was after the introduction of ES6 that an additional scope called **Block scope** along with const and let were introduced.

-before ES2015, it was the position or the place where the **var** is declared that determine the scope of the variable.

- **Scope** refers to the visibility of **variables**. In other words, which parts of a program can see or use it. Normally, every **variable** has a global **scope**. Once **defined**, every part of the program can access a **variable**.

- The scope is global when a var variable is declared outside a function. This means that any variable that is declared with var outside a function block is available for use in the whole window. For example

```
var myName = "yonas";

// code here can use myName

function myFunction() {
 // code here can also use myName
}
```

- A variable is function scoped when it is declared within a function. This means that it is available and can be accessed only within that function. For example

```
// code here can NOT use myName

function myFunction() {
 var myName = "yonas";
 // code here CAN use myName
}

// code here can NOT use myName
```

-But there is a significant weakness with var, and it is this weakness that led the ECMAScript developers to add **let** and **const on** the updated version, which is EC6. Lets look at this weakness by the following demo.

```
Var myName = "yonas";

Var age = 21;

If(age>20){

        myName = "yonatan"

    }
```

-At the above code embed,we can clearly observe that **myName** is redefined to "yonatan". While this is not a problem if we knowingly want **myNamr** to be redefined, it becomes a problem when you do not realize that a variable **myName** has already been defined before. So to solve this short come, it must be introduced a variable that declarer that differ on the global and block scope.

I.Let

- It preferred for variable declaration in javaScript as it comes as an improvement to var declarations. It also solves the problem with var that we just covered.

-let is block scoped. A block is a chunk of code bounded by {}. A block lives in curly braces. Anything within curly braces is a block. So a variable declared in a block with let is only available for use within that block.

-Lets demonstrate this using the above code embeds:-

```
let myName = “yonas”;

let age = 21;

If(age>20){

        let anotherName = “yonatan”

}

//at this place anotherName is not redefined
```

- Just like var a variable declared with let can be updated within its scope. Unlike var, a let variable cannot be re-declared within its scope. However, if the same variable is defined in different scopes, there will be no error

- So from the above demo, we can infer that **let** variables can be updated but can’t be redefined in the same block

II. Const

- Variables declared with the const maintain constant values. const declarations share some similarities with let declarations.

-const cannot be updated or re-declared**.** This means that the value of a variable declared with const remains the same within its scope. It cannot be updated or re-declared. So during const none of the code given below compiles.

```
const myName = “yonas”;
const myName = “Yonatan”;
//can’t be re-declared

Const myName = “yonas”;
myName = “yonatan”;
//can’t be updated
```

- Every const declaration, therefore, must be initialized at the time of declaration.
- This behavior is somehow different when it comes to objects declared with const. While a const object cannot be updated, the properties of these objects can be updated.

3.2. What does hoisting mean?

- Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.
- Inevitably, this means that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local. Of note however, is the fact that the hoisting mechanism only moves the declaration. The assignments are left in place. This is why could call a function in JavaScript before we wrote them in our code.

- In JavaScript, an undeclared variable is assigned the value ‘undefined’ at execution and is also of type undefined. But if we try to access previously undeclared variables, ReferenceError will be thrown.

- As we mentioned before, all variable and function declarations are hoisted to the **top** of their scope. Note that also the variable *declarations* are processed before any code is executed.

-However, in contrast, *undeclared* variables do not exist until code assigning them is executed. Therefore, assigning a value to an undeclared variable implicitly creates it as a global variable when the assignment is executed. This means that, **all undeclared variables are global variables.**

3.3 Why is hoisting different in const and let

- 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**.

I. 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 a code is read.

II. 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.

- 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

➢ Hoisting of let
-Just like var, let declarations are hoisted to the top. Unlike var which is initialized as undefined, the let keyword is not initialized. So if we try to use a let variable before declaration, we'll get a Reference Error.

➢ Hoisting of const
- Just like var and let declarations are hoisted to the top but contrary to two, they are not initialized.

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

4.1 Automatic Semicolon Insertion (ASI)

- The principle of ASI is to provide a little leniency when evaluating the syntax of a JavaScript program by conceptually inserting missing semicolons. There could just be a case that a program parses successfully based on this rule, as opposed to actually changing the code and adding the semicolons.

- The reason semicolons are sometimes optional in JavaScript is because of this automatic semicolon insertion, or ASI. Rather than an actual addition of a semicolon on in the 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.

4.2 Automatic Semicolon Insertion Rules:

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

I. 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.

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

-For example

```
Var a
B=
3;
//here since the parser didn't expect b after a, the ASI
will be triggered and it becomes
Var a;
B=3;
```

III. When the end of the source code file is reached

IV. When the parser reaches to a line break, it terminates the statement unconditionally and trigger ASI. Some of these line break statements are return, break, throw, continue…etc.

V. When the line starts with a }, closing the current block

-we generally don't need to use a semicolon for if-else, for loop, while loop and need only one for do…while loop at do{….}   while(…);

-In my opinion, apart from parsing and error risks, not using significantly decrease the code readability. Moreover, one should fully lay confidence on the ASI, so I highly recommend and endorse using semicolons.

5. Expression vs. Statement in JavaScript?

-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.

5.1 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. JavaScript has the following expression categories.

I. Arithmetic Expressions:

Arithmetic expressions evaluate to a numeric value. Examples include the following:-

-10;

-14 + 15; …etc are arithmetic expressions.

II. String Expressions:

String expressions are expressions that evaluate to a string. Examples include the following:-

- “hii”;

-“hello” + “world”….are string expressions

III. 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). Examples include:-

-30<14;//yields false

-10>11 && 12<13;//yields true…etc.

IV. Primary Expressions:

Primary expressions refer to stand alone expressions such as literal values, certain keywords and variable values. Examples include:-

-“hello earth”

-this

-myName//yields the value of the variable myName…etc.

V. 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.

-Examples of left-hand-side expressions include the following:-

-i=10;//i is an expression

-age = 40;..etc.

5.2 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

I. Declaration Statements:

Such type of statements creates variables and functions by using the var and function statements respectively. For example

-var =age;

-function calcAge(){…} …etc.

II. Expression Statements:

Wherever JavaScript expects a statement, we can also write an expression. Such statements are referred to as expression statements. But the reverse does not hold. We cannot use a statement in the place of an expression. For example

-var a =(b=1);

-Standalone primary expressions such as variable values can also pass off as statements depending on the context.

III. Conditional Statements:

Conditional statements execute statements based on the value of an expression. Examples of conditional statements includes the if..else and switch statements.

IV. 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 include break, continue, return and throw.

Resources

-The following resources were used during the preparation of this documentation.

-https://medium.com/@almog4130/javascript-is-it-compiled-or-interpreted-9779278468fc--medium.com

- https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/ --free code camp.org

- https://www.freecodecamp.org/news/just-in-time-compilation-explained/ --free code camp.org

- https://www.techwalla.com/articles/disadvantages-advantages-of-compilers --techwalla.com

- https://itinterviewguide.com/interpreted-language/ --it interview gide.com

- https://medium.com/jspoint/how-javascript-works-in-browser-and-node-ab7d0d09ac2f -- medium.com

- https://blog.greenroots.info/javascript-interpreted-or-compiled-the-debate-is-over-ckb092cv302mtl6s17t14hq1j --blog.greenroots.com

- https://medium.com/@jotadeveloper/abstract-syntax-trees-on-javascript-534e33361fc7 --medium.com

- https://javascriptrefined.io/null-and-typeof-9330e475d272 --javascriptrefind.io

- https://bitsofco.de/javascript-typeof/ -- bitsofco.de

- https://stackoverflow.com/questions/18808226/why-is-typeof-null-object --stackoverflow.com

-
https://2ality.com/2013/10/typeofnull.html#:~:text=In%20JavaScript%2C%20typeof%20null%20is,it%20would%20break%20existing%20code.&text=The%20data%20is%20a%20reference%20to%20an%20object. –2ality.com

- https://medium.com/javascript-in-plain-english/how-hoisting-works-with-let-and-const-in-javascript-725616df7085 --medium.com

https://www.digitalocean.com/community/tutorials/understanding-hoisting-in-javascript -- digitalocean.com

- https://www.freecodecamp.org/news/var-let-and-const-whats-the-difference/ -- freecodecamp.org

- https://www.w3schools.com/js/js_let.asp --w3schools.com

-https://flaviocopes.com/javascript-automatic-semicolon-insertion/ -- flaviocopes.com

- https://code.likeagirl.io/why-the-heck-do-i-need-to-use-semi-colons-in-javascript-4f8712c82329 --code.likeagirl.io

- http://www.bradoncode.com/blog/2015/08/26/javascript-semi-colon-insertion/ -- bradoncode.com

- https://dev.to/adriennemiller/semicolons-in-javascript-to-use-or-not-to-use-2nli -- dev.to

- https://www.youtube.com/watch?v=B4Skfqr7Dbs –Fullstack Acadamy

-https://medium.com/launch-school/javascript-expressions-and-statements-4d32ac9c0e74 – medium.com

-https://2ality.com/2012/09/expressions-vs-statements.html –2ality.com