ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

CENTER OF INFORMATION TECHNOLOGY AND SCIENTIFIC COMPUTING

Assignment 1 JavaScript

Prepared By: - Sintayehu Sermessa

Id:- ATR/8798/11

Section:- IT

January 2020

Contents

1. Is JavaScript Interpreter Language in its entirety?

1.1 What is difference between Interpreter and compiler?

The interpreter is a computer program that directly executes instructions written in a programming without requiring them previously to have been compiled into a machine language program. It translates one Statement at a time.

A compiler is computer software that converts computer code written in one programming language (the source language, like JavaScript, … etc) into alternative programming language (the target language, like machine code (byte Code)).

“JavaScript used to be purely interpreted but that was many years ago. Nowadays, it is JIT (Just in Time)-compiled to native machine code in all major JavaScript implementations. JavaScript is *not* an interpreted language. I wish people would stop answering questions, saying that it is. In some cases, parts of a JavaScript program might be interpreted briefly, see below for explanation.”

Exactly when it’s compiled to machine code varies based on implementation. In the current V8 (used in Chrome and Node.js), it starts out using an interpreter since there is little reason to spend time compiling code that only runs once. However, if a function gets executed more than a couple of times, it’s immediately compiled into optimized native machine code.

For example: -

```
console.log ('Hoppity Hoppity');
oops oops;
```

In this example, we should understand that the interpreter reads the code directly line by line, and by running this code here it shows this “Uncaught Syntax Error: unexpected token” error message, in theory, the “Hoppity Hoppity” should’ve printed the first line of code and throw the Syntax Error but instead show the error message without ever running the code. This means that JavaScript optimized it to the native machine code.

Another example on Hoisting: -

```
max (1, 2);
// 2
function max (num1, num2) {
  return num1 > num2 ? num1 : num2;
}
```

This example also shows that JavaScript optimizes into compiling the code before run time, this shows that JavaScript knows the “max (1,2)” means in the first place, also shows that after compiling the code it interprets the code of the Function line by line so my conclusion is that All programming language are human-readable. Then translates to machine language. And the JavaScript engines have both compiler and interpreter to translate the JavaScript code on the web browser. The flow of the engine is that it interprets the JavaScript code and simultaneously uses the compiler to optimize the code being interpreted. So, it’s kind of both and Mach closer to compiler this day. First the interpreter reads the code and produces bytecode, secondly parsing takes place and turn it into AST (Abstract Syntax Tree), and then the code initially goes to an interpreter and spits out byte code and the compiler optimizes it and converts that part to machine code. It answers really depends on the implementation of that code.

2. The history of “typeof null”

In JavaScript, typeof null is 'object', which incorrectly advocates that null is an object (it isn’t, it’s a primitive value, this is a bug and one that unfortunately can’t be fixed, because it would break existing code.

This is a **bug** that exists since the beginning of JavaScript. The reason that this bug exists is simple. Each JavaScript value has a type tag. First 1-3 bits of each value are reserved for its type. The type tags for different types were as follows: -

- 000 - object
- 001 - int
- 010 - double
- 100 - string
- 110 - boolean

- To define **undefined,** they used a special number 2^{-30} (which of range for integer)
- To define **null** NULL pointer was used.

In the initial version of JavaScript, values were storied in 32-bit units. The first 3 bits represented the data type tag followed by the remaining bits that represented the value.

For all objects it was 000 as the type tag bits. null was considered to be a special value in JavaScript from its very first version. null was a representation of the null pointer. However, there were no pointers in JavaScript like C. So null simply meant nothing or void and was represented by all 0's. Hence all its 32 bits were 0's. So, whenever the JavaScript interpreter read null, it considered the first 3 bits as type "object". That is why typeof null returns "object".

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

3.1 Hoisting

- During compile phase, just microseconds before your 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.

Lexical environment – a lexical environment is place where variables and function live during the program execution.

```
For example: -
      LexicalEnvironment = {
        Identifier:  <value>,
        Identifier:  <function object>
      }

    helloWorld();  // prints 'Hello World!' to the console function helloWorld(){
      console.log('Hello World!');
    }
```

3.2 Hoisting Function

function declarations are added to the memory during the compile stage, so we are able to access it in our code before the actual function declaration.

So, the lexical environment for the above code will look something like this:

```
Example: -LexicalEnviroment ={
                helloworld:<fun>
                }
```

So when the JavaScript engine encounters a call to helloWorld(), it will look into the lexical environment, finds the function and will be able to execute it.

Only function declarations are hoisted in JavaScript, function expressions are not hoisted. For example: this code won't work.

```
helloWorld();  // this here is Undefined
var helloWorld = function(){
  console.log('Hello World!');
}
```

As JavaScript only hoist declarations, not initializations (assignments), so the helloWorld will be treated as a variable, not as a function. Because helloWorld is a var variable, so the engine will assign is the undefined value during hoisting.

during compile time, JavaScript only stores function and variable declarations in the memory, not their assignments (value).

So, the initial lexical environment for the above code will look something like this:

```
lexicalEnvironment = {
  a: undefined
}
```

3.3 Hoisting Let and Const variables

```
console.log(a);
let a = 3;
or
const a = 3;
```

The output is Uncaught Reference Error: can't access lexical declaration 'a' before initialization
Which begs the question **is let and const variable not hoisted?**

So, the answer is All declaration (function, var, let, const) are hoisted in JavaScript, while the var declaration initialized with **Undefined**, but let and const declarations remain uninitialized.

They will only get initialized when their lexical assignment is evaluated during runtime by the JavaScript engine. This means they 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.

For example: -

```
Const =a;
console.log(a)
a = 3;
```

In this case with const we con not do this because, it raises an error of Uncaught ReferenceError: a is not defined why? Before run time In lexical environment the a variable is set to uninitialized so, for const that I bit of a problem.

3.4 Const

- we set a variable const if we don't want to assign it after declaration or initialization. In this case it we can not change it so it will stack in the temporal dead zone we can't access its value because it wasn't initialized first this means that with const it must be initialized before run time.

4. Semicolons in JavaScript

It's our preference whether we use semicolon in code, but in JavaScript there ais a particular code construct requires them. 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 (ASI).** So, there are rules that powers semicolon the rules of **Automatic Semicolon Insertion are: -**

4.1 Rule of ASI

1. when the next line starts with code that breaks the current one.
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.

never start a line with parentheses, those might be concatenated with the previous line to form a function call, or array element reference

In the head of for loops ASI is not implied for loops semicolon are required can't rely on ASI to add a semicolon. ASI also not implied in statement appear in the same line
For example: -

```
for(let i = 0
        ; 1< 10
         ; i ++){
```

}
can't add a semicolon here so that a problem for as if we forgot to put it there. The ASI won't work .

For example: -
23 "Hello"

The semicolon won't be added here as well the ASI can't distinguish where to put semicolon.

4.2 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 (...);

For example: -

```
var name = "Abebe"
var website = "abebe.org";
```

On the first line, the JavaScript engine will automatically insert a semicolon, so this is not considered a syntax error. The JavaScript engine still knows how to interpret the line and knows that the line end indicates the end of the statement.

Where was an argument amount school about ASI generally. The first is that we should treat ASI as if it didn't exist and always include semicolons manually. The rationale is that it's easier to always include semicolons than to try to remember when they are or are not required, and thus decreases the possibility of introducing an error.

While in the case of rule 4 it can be tricky to people who uses semicolon.

For example: -

```
function myFunction(){
          return
         {
              name:"Abebe";

          };

};
```

This may look it's grammatically correct but the ASI assumes that the end of a line is termination so it adds a semicolon that the end of return like thus

```
function myFunction() {
            return;
            {
              name: "Abebe";
            } ;
 };
```

On the other side of the argument are those who say that since semicolons are inserted automatically, they are optional and do not need to be inserted manually. However, the ASI mechanism can also be tricky to people who don't use semicolons. For example, consider this code:

```
var globalCounter = { }

(function () {
   var n = 0
   globalCounter.increment = function () {
      return ++n
   }
})()
```

This also a tricky part, a semicolon will not be inserted after the first line, causing a run-time error (because an empty object is called as if it's a function).

4.3 My Conclusion

It's best if we know the rule of ASI so that we couldn't be confused about when to use it or not. In my opinion, knowing thus rules and not using the semicolon is advantageous because there is a problem if we use the semicolon or not use them as I explained above. As I explained above even if we don't use it some codes that need the semicolon so that its best to know the ASI rules use semicolon when it is crucial.

5. Expression vs Statement in JavaScript

5.1 Expression

Expression is a piece of code that resolves to a value.

JavaScript distinguishes *expressions* and *statements*. An expression produces a value and can be written wherever a value is expected, for example as an argument in a function call. Each of the following lines contains an expression:

For example: -

```
        Const y = 5;
        const y = myfunc();
```

These two values of y are an expression as explained above. Which means that in the case of const y = myfunc(); it is call a function and that function resolves a value then it's an expression

Seeing expression helps us understand the process of computation. Expression can be statements. expression is bit of code that produces a value usually part of a larger statement.

For example: -

```
        12 + square (7+5) + square (square (2));
```

This example consists of multiple subexpressions so what happens is that the number 12 resolves to 12 and the goes to square of (7 + 5) and it resolves 7 to 7 and 5 to 5 and added them together becomes 12 then squares it to be 144 and goes on until the line is finished and became one value so this is what happens:

```
      12 + square (7+5) + square (square (2));
      12 + square (12) + square (square (2));
      12 + 144 + square (square (2));
      12 + 144 + square (4);
      12 + 144 + 16;
      156 + 16;
      172;
```

5.2 Statement

statement is an instruction, an action. Which means that they perform action and control action, but don't become value. An executable block of code. Most statement are rapped in semicolon or {}'s.

For example: - loops and if statements all of them are statement because they just perform and control actions.

```
      var form = document.querySelector('form'); // this is a statement
```

```
form.addElememtListener(‘submit’,  displaypost(){
        var title = document.querySelector(‘#title’).value,
         content = document.querySelector(‘#content’).value;
console.log(title);
console.log(content);
event.preventDefualt();
} // this is also a statement
```

in this example the first line is that statement and it has an expression in it the document.querySelector(‘form’) is an expression because it give us a value of the form element itself.

And also when the form submit the entire function actually an example of expression that form part of a larger statement and the statement start from form.addElememtListener(…); end at the semicolon.

Roughly, a statement performs an action. Loops and if statements are examples of statements. A program is basically a sequence of statements (we’re ignoring declarations here). 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.

In programming language an expression is a combination of value and function that are combined and interpreted by the compiler to create a new value.

6. References

1.https://www.quora.com

2. https://medium.com

3. https://stackoverflow.com

4. https://dev.to

5. https://fsharpforfunandprofit.com