ADDIS ABABA INSTITUTE OF TECHNOLOGY

Center of Information Technology and Scientific Computing

Fundamental of Web Development

Assignment -1

Name: *Rihana Abdela*

***ID NO**: ATR/5728/08*

Add

Submitted to:

Mr.Fitsum

Submitted date:

Feb, 2021

1.

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.

Interpreted Languages: - Interpreters run through a program line by line and execute each command. Here, if the author decides he wants to use a different kind of olive oil, he could scratch the old one out and add the new one. Your translator friend can then convey that change to you as it happens.

JavaScript is an interpreted language, not a compiled language. 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 bytecode 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 bytecode just as it is about to run.

2.

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 -2^{30} (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.

3.

HOISTING

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

All variable and function declarations are hoisted to the **top** of their scope. I should also add that variable *declarations* are processed before any code is executed. *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.**

Let

`let` are block scoped and not function scoped. That's significant, but it shouldn't trouble us here. Briefly, however, it just means that the variable's scope is bound to the block in which it is declared and not the function in which it is declared.

```
console.log(hoist); // Output: ReferenceError: hoist is not defined ...
let hoist = 'The variable has been hoisted.';
```

Like before, for the `var` keyword, we expect the output of the log to be `undefined`. However, since the es6 *let* doesn't take kindly on us using undeclared variables, the interpreter explicitly spits out a `Reference` error.

This ensures that we **always** declare our variables first.

However, we still have to be careful here. An implementation like the following will result in an ouput of `undefined` instead of a `Reference error`.

```
let hoist;

console.log(hoist); // Output: undefined
hoist = 'Hoisted'
```

Hence, to err on the side of caution, we should *declare* then *assign* our variables to a value before using them.

const

The `const` keyword was introduced in es6 to allow *immutable variables*. That is, variables whose value cannot be modified once assigned.

With `const`, just as with `let`, the variable is hoisted to the top of the block.

Much like the `let` keyword, instead of silently exiting with an `undefined`, the interpreter saves us by explicitly throwing a `Reference error`.

The same occurs when using `const` within functions.

```
function getCircumference(radius) {
  console.log(circumference)
  circumference = PI*radius*2;
  const PI = 22/7;
}

getCircumference(2) // ReferenceError: circumference is not defined
```

With `const` , es6 goes further. The interpreter throws an error if we use a constant before declaring and initialising it.

Our linter is also quick to inform us of this felony:

```
PI was used before it was declared, which is illegal for const variables.
```

Globally,

```
const PI;
console.log(PI); // Ouput: SyntaxError: Missing initializer in const declaration
PI=3.142;
```

Therefore, a constant variable must be both declared and initialised before use.

As a prologue to this section, it's important to note that indeed, JavaScript hoists variables declared with es6 let and const. The difference in this case is how it initialises them.
Variables declared with *let* and *const* remain **uninitialised** at the beginning of execution whilst variables declared with *var* are initialised with a value of **undefined**.

4.

Semi-colons are optional in JavaScript, but omitting them can lead to dire consequences on rare occasions. 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

An expression produces a value and can be written wherever a value is expected, for example as an argument in a function call.

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. Different categories of expressions

Arithmetic Expressions: Arithmetic expressions evaluate to a numeric value.

Example: 10; , 10+3;

String Expressions: String expressions are expressions that evaluate to a string.

Example: 'hi'

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).

Example: 45>23;

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

Example: 'hello world';

Assignment Expressions: When expressions use the = operator to assign a value to a variable, it is called an assignment expression.

Example: average = 55; var b = (a = 1);

Statement

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. Different categories of statement

Declaration Statements: Such type of statements creates variables and functions by using the var and function statements respectively.

Example: var;

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.

Example: var a= var b;

Conditional Statements: Conditional statements execute statements based on the value of an expression.

Example:

```
if (expression)
statement 1
else
statement 2
```

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.

Reference

1. https://medium.com/launch-school/javascript-expressions-and-statements-4d32ac9c0e74
2. https://flaviocopes.com/javascript-automatic-semicolon-insertion/#:~:text=JavaScript%20semicolons%20are%20optional.&text=This%20is%20all%20possible%20because,is%20called%20Automatic%20Semicolon%20Insertion.
3. https://code.likeagirl.io/why-the-heck-do-i-need-to-use-semi-colons-in-javascript-4f8712c82329
4. https://www.digitalocean.com/community/tutorials/understanding-hoisting-in-javascript
5. 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.