Variable scopes in javascript

A variable is said to be in **global**scope when it is accessible

* throughout the program,
* across functions,
* across files

All variables in JavaScript are global by default. Observe the validateTravellerPassword() and validateTravellerName() shown below.

1. function validateTravellerPassword(password) {
2. for (i = 0; i < password.length; i++) {
3. }
4. }
5. function validateTravellerName(name) {
6. console.log("The value of i is "+i);
7. for (i = 0; i < name.length; i++) {
8. }
9. }
10. validateTravellerPassword("12345");
11. validateTravellerName("Jack");

As shown above, the variable i used in the for-loop of validateTravellerPassword() is global.

At the end of this function, the value of i becomes 5 because the loop executes 5 times. The same value of i, is also accessible in validateTravellerName(), since i has global scope.

**Local scope** is when a variable is accessible only within a function. This is also called function scope.

In JavaScript, any variable declared with the **var** keyword inside a function, is considered local. However, if the variable is created with **var** outside a function, it still behaves like a global variable.

To fix the previous code, we can add the var keyword to **i**. Then, the value of **i** in validateTravellerPassword() is restricted only to itself and will not affect the value of **i** in validateTravellerName().

1. function validateTravellerPassword(password) {
2. for (var i = 0; i < password.length; i++) {
3. }
4. }
5. function validateTravellerName(name) {
6. console.log("The value of i is "+i);
7. for (var i = 0; i < name.length; i++) {
8. }
9. }
10. validateTravellerPassword("password");
11. validateTravellerName("Josh");

**Observe**that in line 6 of validateTravellerName(), we are printing the value of **i**, even before it is declared. Will this cause an error?

The previous code will not throw an error, even though the variable **i** was accessed before its declaration!

In JavaScript, before any function executes, all the local variables are **hoisted**in the function. Hoisting is a phenomenon, where no matter where the variable is declared inside the function, they are all pushed as the first statements inside the function during the function execution.

Here is the validateTravellerName()

1. function validateTravellerName(name) {
2. *//var i; ( internally the variable i gets hoisted as the first statement of the function )*
3. console.log("The value of i is "+i);
4. for (var i = 0; i < name.length; i++) {
5. }
6. console.log(i);
7. }

It is because of this variable hoisting, we are able to access the value of i even before it is declared. However, only variable name is hoisted and not its value. That is why we get undefined instead of zero in the console.

# Block scope

In the validateTravellerName() or validateTravellerPassword() functions, the value of ***i*** makes sense only within the for-loop block. However, since the variable ***i*** has been declared with var keyword, it is accessible throughout the function.

A variable with a **block scope** is accessible only within the block of statements and not throughout the function.

We can create such block scoped variables with the **let**keyword as shown below.

1. function validateTravellerName(name) {
2. for (let i = 0; i < name.length; i++) {
3. }
4. console.log(i); *// This will give an error as i is not accessible outside the for block.*
5. }
6. validateTravellerName("Josh");

**Note**: Such block scoped variables are not hoisted.

# Const

**const** is a keyword which is also used to create a block scoped variable. But the difference between const and let is that, a const variable cannot be modified. It is constant.

For example the below code will given an error as we are modifying a const value:

1. {
2. const a=10
3. a=20
4. }

The below code will given an error as we are accessing const outside a block:

1. {
2. const a=10;
3. }
4. console.log(a);

* Functions are objects in JS
* Functions can be stored in variables, passed as parameters or returned as values
* Functions without names are called anonymous functions
* Arrow functions are shorter way to write anonymous functions
* Variables can either be in global, local or block scope

# Built-in Functions

JS also has many useful in-built functions. Two commonly used ones are Number() and String(). Let us look at them in detail.

**Number()**, which is the constructor of **Number**object, converts value of an object to a number. If it is unable to convert the value, it returns **NaN**.

Syntax:

|  |
| --- |
| 1. Number(argument) |

Example:

|  |
| --- |
| 1. var x="123"; 2. console.log(Number(x)); *// 123* 3. console.log(Number("123")); *// 123* 4. console.log(Number("123.1")); *// 123.1* 5. console.log(Number(10/0)); *// Infinity* 6. console.log(Number(NaN)); *// NaN* 7. console.log(Number("123abc")); *// NaN* |

**String()** converts the value of an object to a string. It is the constructor of the **String**object.

Syntax:

|  |
| --- |
| 1. String(argument) |

Example:

|  |
| --- |
| 2. console.log(String(1)); *// "1"* 3. console.log(String("Hello")); *// "Hello"* |