UNIT 1

JAVASCRIPT

Part 1 - Annex
JavaScript Basics

Client-side Web Development
2nd course – DAW
IES San Vicente 2020/2021
Author: Arturo Bernal Mayordomo

Index

Unit 1 – part 1 - Annex

Variable “hoisting”(ES5 and before using var)

The word “hoisting” defines a particular JavaScript behavior with variable declared with the **var** keyword. Before executing the code, JavaScript analyzes it and does 2 main things:

- Load function declarations in memory (so they can be accessible wherever).
- Move variable declarations (with **var**) to the top of the funcions (or if not in a function, to the top of the program main block).

Let’s see an example:

```
function printHello() {
   console.log(hello);
   var hello = "Hello World";
}

printHello(); // This will print undefined
```

It should print an error because you’re trying to print hello variable before it exists. But instead prints “undefined”. That’s because in the first pass it has transformed the code internally to this:

```
function printHello() {
  var hello = undefined;
   console.log(hello);
   hello = "Hello World";
}

printHello();
```

With the new JavaScript standard (ES2015), you should declare a variable using **let** instead of **var** to avoid this behavior. The interpreter will show an error saying you’re trying to use a variable which is not yet declared.

var vs let

Using **let** is the recommended way now to declare variables. The main difference with var is that it avoids ***hoisting***. If a variable is accessed before its declaration, it won’t return undefined (variable declaration is moved to the top internally by the interpreter), instead, it will throw an error (logical behavior).

```
'use strict';
console.log(number); // Prints undefined (hoisting)
var number = 14;

'use strict';
console.log(number);  → Uncaught ReferenceError: number is not defined
```

```
let number = 14;
```

Other advantage of **let** is that a variable becomes also local to any block. For example, a variable declared with **let** inside an if or while block, only exists inside that block (not in all the function block).

```
'use strict';
for(var i = 0; i < 10; i++) {
    console.log(i);
}
console.log(i); // Prints 10 (i exists outside the loop)
```

```
'use strict';
for(let i = 0; i < 10; i++) {
    console.log(i);
}
console.log(i); → Uncaught ReferenceError: i is not defined
```

We can also create a simple block just to change the scope of a variable.

```
'use strict';
let number = 10;
{
    let number = 200; // Here number is local to this block
}
console.log(number); // Prints 10 (number value outside the previous block)
```

Let's see another example which usually is problematic when creating functions inside a loop for example, and using the counter inside those functions:

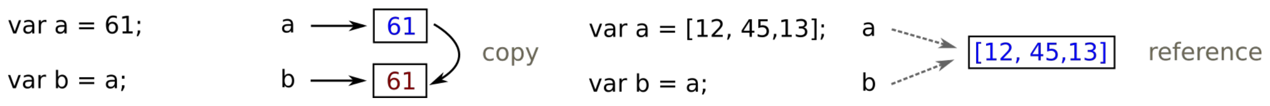
```
'use strict';
let functions = [];
for(var i = 0; i < 10; i++) {
    functions.push(function() {
        console.log(i);
    });
}
functions[0](); // Prints 10, i current value (not local to each iteration)
functions[1](); // Still prints 10
```

Using let, it works as expected:

```
'use strict';
let functions = [];
for(let i = 0; i < 10; i++) {
    functions.push(function() {
        console.log(i);
    });
}
functions[0](); // Prints 0 (now, every iteration, a local value is created)
functions[1](); // Prints 1
```

Reference vs Copy

In contrast with primitive types like **boolean**, **number** or **string**, arrays are treated as every other object in JavaScript, that means that variables hold a pointer or a reference to a memory zone containing data, and when we copy that variable or pass its value to a function as an argument, we are copying the reference only, not the data, so, both (or more) variables will point to the same memory zone. That means if we make changes to the object through any variable, it will affect the data accessed by the other variables, because it's the **same object**.



Lets see some examples to support this explanation:

```
let a = 61;
let b = a; // b = 61
b = 12;
console.log(a); // Prints 61
console.log(b); // Prints 12

let a = [12, 45, 13];
let b = a; // b now references the same array as a
b[0] = 0;
console.log(a); // Prints [0, 45, 13]
```

Another example, passing the array to a function:

```
function changeNumber(num) {
   var localNum = num; // Local variable. Gets a copy of the value
   localNum = 100; // Changes only localNum. Value was copied.
}

let num = 17;
changeNumber(num);
console.log(num); // Prints 17. Hasn't changed its original value

function changeArray(array) {
   var localA = array; // Local variable. Doesn't matter, still gets a reference...
   localA.splice(1,1,101, 102); // Remove 1 item at position 1 and insert 101 and 102 there
}

let a = [12, 45, 13];
changeArray(a);
console.log(a); // Prints [12, 101, 102, 13]. Has been changed inside the function!
```