**CONOCIMIENTOS INTERMEDIOS**

**Entendiendo el scope**

**¿Qué es el scope?**

El scope es el conjunto de variables y funciones que podemos llamar en cada parte del código. Si definimos una variable en el scope global, podemos acceder a ella dentro de cualquier lugar del código; en cambio, si definimos una variable (let/const) dentro de una función solo la podremos utilizar dentro de esa función o bloque.

Al **scope** se le conoce también como *ámbito*, *contexto* o ***alcance***.

Ahora bien, si estamos en un bloque o función y declaramos la variable con var, no la mantendremos en el scope donde estamos situados sino que **la declararemos como una variable global en el scope global**. Esto generalmente hay que evitarlo y se considera una mala práctica porque **estaremos contaminando el scope global** y por ende, podremos generar un problema de seguridad de datos. Hay que intentar usar variables de tipo **let o const para respetar mejor el scope** y que nuestros datos estén seguros.

*Ejemplo 1:*

**var nombre = “Dani”;**

**function saludar() {**

**var nombre = “Alex”;**

**console.log(`Hola ${nombre}`);** // imprime “Hola Alex”

**}**

**saludar();**

**console.log(`La variable nombre tie-ne el valor ${nombre}`);** // Aquí imprime “Dani” porque la variable nombre ya existe, pero si no estuviera declarada fuera de la función utilizaría la de dentro de la función saludar() (“Alex”).

Si en vez de escribir var nombre = “Alex” dentro de saludar(), lo pusieramos sin el var delante, estaríamos sobreescribiendo la variable nombre fuera de la función y por lo tanto el console.log de fuera imprimiría “Alex” en vez de “Dani”.

*Ejemplo 2:*

**function saludar() {**

**let nombre = “Alex”;**

**console.log(`Hola ${nombre}`);** // imprime “Hola Alex”

**}**

**saludar();**

**console.log(`La variable nombre tiene el valor ${nombre}`);** // Aquí nos salta un error al no poder encontrar la variable nombre en el scope global. Eso se debe porqué usamos let en vez de var, y como hemos dicho antes let/const respetan el scope en el que se encuentran.

**Hoisting, ¿que es?**

Las variables en Javascript se pueden utilizar antes de que se declaren más tarde. Esto hace referencia al concepto de hoisting. La variable declarada más tarde se eleva a la parte superior de la función en la que se encuentra. Lo mismo sucede con las funciones, en Javascript podemos ejecutarlas antes de que las declaremos.

Gracias a ECMAScript 2015, declarando las variables con **let** podremos evitar el hoisting y que las variables que se declaren en un bloque dentro de una función se queden dentro del scope del bloque y no se vaya arriba de la función.

**function saludar10(nombre){**

**for (var i = 0; i < 10; i++) {**

**console.log(`Hola ${nombre}`);**

**}**

**console.log(`El valor de i es: ${i}`);** //Este valor i se va a ir imprimiendo del for porqué como dijimos, declarando las variables con var se elevarán arriba del clousure de la función. Si la declaramos con let respetará el scope y en ese console.log nos aparecerá un error como que “i” no está definido (como debería ser).

**}**

**saludar10(“Dani”);**

**Operaciones con arrays**

Los arrays son un tipo de objeto que nos permite almacenar datos en forma de lista. Es muy habitual el uso de este tipo de objeto y es importante saber operar y utilizar los arrays correctamente.

Los arrays tienen una serie de métodos que nos permiten evitar tener que hacer ciclos para recorrer sus elementos (**.reduce()**, **.map()**, **.filter()**). Aprendiendo a utilizarlos podremos ahorrarnos el uso de esos ciclos y potenciar el uso de los arrays.

En las funciones podemos escribir un parámetro para que almacene todos los atributos que le pasemos a la función y los almacene en un array:

**function lista(...params) {**

**console.log(params);** //imprimirá un array con todos lo que introducimos a la hora de llamar la función.

**}**

**lista(1, 3, 20, “string”, “xd”);**

Esto se conoce como **operador de propagación** (spread operator) y se escribe con tres puntos delante y el nombre del parámetro. Contando con esta opciones vamos a poder crear funciones que operen con los datos de esos arrays.

**Array.reduce()**

Todos los arrays tienen un método reduce(). Este método aplica una función a un acumulador mientras va recorriendo el array y, todos los valores del array los convertimos en un solo valor (variable acumulador).

A este método le tenemos que pasar una función con cuatro posibles argumentos: el acumulador, el valor actual, el índice actual (opcional) y el array entero (opcional). Tal que así:

**array.reduce(function(acumulador, valorActual, indice, array){**

**return acumulador + valorActual**

**});**

Ej: Suma todos los números del array.

**function suma(...nums){**

**return nums.reduce((acum, num\_actual) => {**

**acum += num\_actual;**

**return acum;**

**});**

**}**

**suma(3, 5, 1);** //Devuelve 9

**let array = [10, 15, 5];**

**suma(array);** //Devuelve 30

**Array.map()**

Otro de los métodos es .map(), que nos permitirá modificar cada uno de los elementos dentro de un array y al final nos devolverá el nuevo array con los elementos modificados.

La sintaxis es muy parecida a .reduce(), donde le pasamos una función con tres posibles argumentos: el valor actual a modificar, el índice del elemento actual en el array (opcional) y el array que le pasamos (opcional).

**array.map(function (valor\_actual, indice, array){**

**return valor\_actual + 1;**

**});**

Ej: Multiplicar por dos todos los elementos del array.

**function dobles(...nums){**

**return nums.map( (num\_actual) => num\_actual \* 2);**

**}**

**dobles(10, 2, 3, 8);** //Devuelve un array con los valores [20, 4, 6, 16]

**Array.filter()**

Si queremos filtrar los elementos de un array tenemos disponible .filter(). Lo que hace es crear un nuevo array con los elementos que cumplan la condición que queramos.

La sintaxis es calcada a la de .map(), donde le pasamos una función callback al método map() y en esa función tenemos tres argumentos disponibles para usar: el elemento actual, el índice y el array que le hemos pasado. Ahora bien, dentro de la función callback no modificamos los elementos sino que escribimos una condición que validará si el elemento actual es apto para introducirse al nuevo array.

**array.filter((valor\_actual, indice, array){**

**return condición;** //Debe devolver un booleano para validar la condición

**});**

Ej: Filtramos los números pares dentro de nuestro array.

**function pares(...nums){**

**nums.filter(valor\_actual => valor\_actual % 2 == 0);**

**}**

**pares(2, 3, 9, 20);** // Devuelve un array con [2, 20]

Un apunte, si queremos **añadir elementos a nuestro array** es tan sencillo como usar el método **.push()** sobre el array al que queremos introducir esos elementos: **array.push(4, “elemento”, 23)** .

**Entendiendo los clousures**

Los **clousure** son funciones que recuerdan el scope y todas las variables que tenían en la función en que se encuentran. Es decir, que si tenemos una función dentro de otra función, la función interna recordará el scope de la función externa en la que están dentro.

**function funcionPadre(parametro1, parametro2) {**

**return function funcionHijo(parametro) {**

**console.log(parametro1, parametro2, parametro);** // Fíjate que podemos acceder a parámetros y variables que han sido declaradas dentro de la función en la que hemos declarado nuestra nueva función (función padre).

**}**

**}**

Ej: Clousure donde recuerda el apellido de una familia para saludar a un miembro con nombre y apellido.

**function saludarFamilia(apellido){**

**return function (nombre) {**

**console.log(`Hola ${nombre} ${apellido}`);**

**}**

**}**

// Versión con arrow function

**const saludarFamilia = apellido => nombre => console.log(`Hola ${nombre} ${apellido}`);**

// Ahora guardamos el primer parámetro de la función saludarFamilia en una constante

**const saludarCueto = saludarFamilia(‘Cueto’);**

// La constante recuerda el parámetro que le hemos dado (“Cueto”) y ahora podemos introducir los parámetros de la función clousure.

**const saludarDaniel = saludarCueto(‘Daniel’);**

// Esto nos devuelve: “Hola Dani Cueto”.

RETO: Genera un clousure para escribir palabras con prefijos

**function makePrefixer(prefix){**

**return function(word){**

**return prefix+word;**

**}**

**}**

// Versión con arrow function

**const makePrefixer = prefix => word => prefix + word;**

// Creamos el prefijo e imprimimos palabras.

**const prefixRe = makePrefixer(‘re’);**

**prefixRe(‘hola’);**

**const reload = prefixRe(‘load’);**

**reload**

**Variables de débil tipado**

Javascript es un lenguaje con tipado débil. ¿Qué significa eso? Pues que las variables no se inicializan con tipo específico sino que podemos modificar el tipo de la variables cuando queramos (menos los const porqué son constantes y no se pueden modificar). Además en un método podemos pasar como parámetro un atributo que no sea el tipo de dato necesario para ejecutar esa función y seguramente genere un error.

Existen otros lenguajes que están diseñados con tipado fuerte, por lo tanto los tipos de cada variable no son modificables y existe un control preciso de lo que pasamos como parámetros a las funciones y a las clases.

El tipado débil tiene sus pros y sus contras al igual que el tipado fuerte. Por lo tanto en Javascript podemos hacer ciertas cosas que en un lenguaje de tipado fuerte no, y lo mismo a la inversa.

**IIFE: Inmediatly Invoked Function Express**

En Javascript existen funciones que se pueden autoejecutar justo al declararse. Estas son las denominadas IIFE (Inmediatly Invoked Function Express). Es uno de los patrones de programación en Javascript más potentes e interesantes y, por lo tanto, es algo extenso de explicar. Lo más interesante de las IIFE no es solo el hecho de que se invoquen justo en el momento de declararlas, sino que además desaparecen justo al invocarse. Esto da pie a poder crear variables y datos que no puedan ser modificados, y por ende, crear un entorno más seguro y limpio de variables en el scope global.

Cualquier cosa que creemos en esa función (objetos, arrays, strings, etc...) nos resultará imposible acceder a ella a no ser que lo devolvamos en el return. Esto da a pie a poder utilizar también clousures como retorno de la función iife.

La sintaxis de una IIFE puede ser de distintas formas, aunque la más común de ver es esta:

**(function iife(){**

**//código a ejecutar**

**})();**

Evidentemente si no entiendes el contexto de por qué se hace cada cosa no es posible entender esta sintaxis.

Para entender las iife en profundidad recomiendo este artículo de aquí: <https://medium.com/@vvkchandra/essential-javascript-mastering-immediately-invoked-function-expressions-67791338ddc6>

Lo más básico para entender las múltiples formas de escribir una iife es entender que una función iife es una function expression, no una function declaration.

Una function expression es una función asignada a una variable que luego la podemos ejecutar escribiendo el nombre de la variables seguido de dos paréntesis.

**const funcionHola = function hola(){ console.log(“hola”); }**

**funcionHola();** //Imprime “hola” en consola.

Para convertir esa función en una expresión precisamos de un carácter especial como !, +, - o ~ delante del function tal que así: !function hola() {return “hola”}(); .

Rodear los paréntesis también indica que no es una función declarada sino una expresión de función y por ende podemos llamar esa función de forma seguida con los dos paréntesis.

Si queremos poder reutilizar esa función iife para que nos devuelva algún valor, objeto o una clousure, podemos asignarla a una variable para luego poder llamarla.

Ej: Vamos a crear una iife que devuelva un objeto con algunos métodos que permitan modificar la variable current dentro del iife (recordamos que desde fuera es imposible modificar esa variable, solo la podemos modificar desde dentro de la iife) y luego poder volver a llamar esa iife para utilizar los métodos del objeto que devolvemos.

**let Sequencia = (function secuenciaIIFE(){**

// Variable privada para almacenar el valor actual del contador

**var current = 0;**

// Objeto que devolvemos con los métodos a poder utilizar fuera del IIFE.

**return {**

**getCurrentValue: () => current,**

**getNextValue: () => current += 1**

**};**

**})();**

**console.log(Secuencia.getNextValue());** // Imprime 1

**console.log(Secuencia.getCurrentValue());** // Imprime 1

**console.log(Secuencia.getNextValue());** // Imprime 2

La variable current sólo se puede modificar con estos métodos y además no podemos reescribir esos métodos desde fuera. Con esto conseguimos un entorno mucho más seguro, protegido y avanzado.