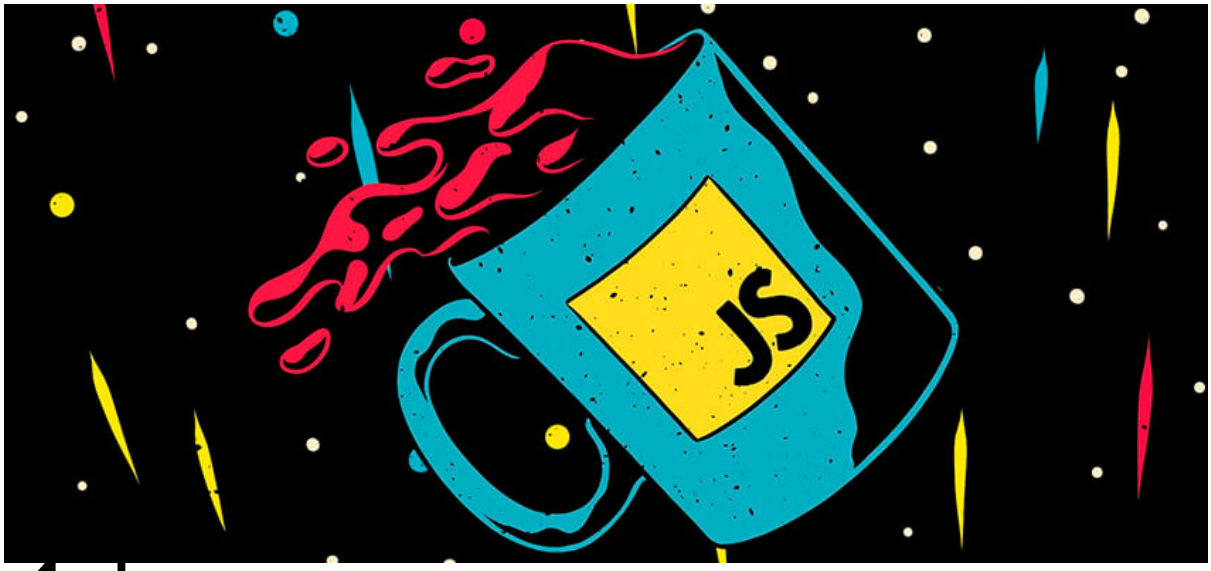

Javascript S2 | Scope - Clousures - Context - Hoisting

Después de esta lección podrás:

1. Entender y trabajar con los conceptos básicos de Scope, Hoisting y Clousures.
2. Diferenciar el ámbito de las variables y funciones.
3. Entender cómo organiza Js nuestro código antes de ejecutarlo.
4. Trabajar con let y const → Dejando var en el pasado.
5. Trabajar con Arrow functios → Bienvenidos a ES6.

Qué es el Scope?

El scope es el **alcance** de una **variable**, puede ser de dos tipos, **global y local**. Una variable cuyo scope es global se puede acceder desde cualquier parte del código, una local solo desde la función que la contiene. Ejemplo:

```
// Declaramos variable
var avenger = 'Wolverine';
// Función
function global() {
// Accedemos a la variable
 console.log(avenger);
}
// Ejecutamos función
global();
// Imprimimos valor de variable
console.log(avenger);
```

En ese caso a es una **variable global** ya que podemos **acceder** tanto **fuera** como **dentro** de una **función** debido a haberla definido fuera de cualquier función.

En el siguiente caso, la variable será local ya que la definiremos dentro de la función **local()**, esto quiere decir que solo podemos acceder a ella dentro dicha función, cuando ejecutamos **local()** te muestra correctamente '***Wolverine***', mientras que si haces **console.log(avenger)** te va a dar error porque a no esta definida, para el scope global esa variable no existe.

```
// Declaramos una función
function local() {
 // Declarmaos variable
 var avenger = 'Wolverine';
 // Imprimimos por consola
 console.log(avenger);
}
// Ejecutamos función -> wolverine
local();
// Ahhhg! Error
console.log(avenger);
```

Qué es un clousure?

Un closure es una **función sin variables propias**, pero tiene **acceso a las variables de la función padre**. Un ejemplo:

```
function lordVader() {
  var jedi = 'May the Force be with you';
  function lukeSkywalker() {
    console.log(jedi);
  }
  lukeSkywalker();
}
lordVader();
```

La función ***lordVader() o padre*** inicializa una variable local e invoca ***lukeSkywalker()***. **Esta función interna es un closure** y solo esta **disponible dentro** de ***lordVader()***. A diferencia de ***lordVader()*** esta función, ***lukeSkywalker()***, no tiene variables locales y usa las declaradas dentro de ***lordVader***().

Qué es el contexto - this?

El contexto determina cómo se invoca una función. Cuando una función es invocada en un objeto, el “this” será este objeto. Vamos a nuestra consola de chrome y ejecutemos:

```
console.log(this);
//this apunta al objeto ventana por defecto
> Window { postmensage:f, blur:f, focus:f, close:f, window,... }
```

Nos devuelve el objeto window porque en este caso el contexto de ejecución es Global, lo que significa que no está bloqueado por ninguna ámbito de objeto. Pero y si por ejemplo ejecutamos:

```
var avengers = {
   value: 'this se encuentra dentro de avengers',
   avengerFunction: function() {
```

```
        //this en el scope de un objeto
        console.log(this);
    }
}
avengers.avengerFunction();

{value: "this se encuentra dentro de avengers", avengerFunction: ƒ}
  avengerFunction: ƒ ()
  value: "this se encuentra dentro de avengers"
  __proto__: Object
```

El valor de this apunta a ***avengers***. Esto se debe a que el valor de this depende de los objetosdesde los que se invoca.

Recordáis en la sesión anterior cuando hablabamos sobre las clases en Javascript y teníamos un ejemplo en el que existía un comentario diciendo "***Uso del this. en la sesión 2 → CONFIANZA"*** . Os dejamos el ejemplo del que hablamos y es el momento de explicarlo:

```
class Animal {
   constructor(nombre, sonido) {
      //Uso del this. en la sesión 2 -> CONFIANZA
      this._nombre = nombre;
      this._sonido = sonido;
    }
    emitirSonido() {
      console.log("El " + this._nombre + " hace " + this._sonido);
    }
}

var miPerro = new Animal("Perro", "Guau!");
var miGato = new Animal("Gato", "Miau!");
miPerro.emitirSonido(); // El Perro hace Guau!
miGato.emitirSonido(); // El Gato hace Miau!
```

Vale os preguntaréis **¿Por qué ponerle "this" al principio en lugar de solamente ponerle "nombre = nombre" o "sonido = sonido"?**

En JavaScript **se requiere** usar this en clases para denotar que dicha variable **es propiedad** de la clase. Si no utilizas `this`, dicha variable

se **almacena en el scope global**. Y entonces tendría acceso a esta variable desde cualquier punto de la aplicación.

Trabajando con el contexto

Imaginemos el siguiente ejemplo:

```
var fantasticFour = {
  nombre: "Reed Richards",
  miName: function () {
    console.log(this.nombre);
} };
var miFantastic = fantasticFour.miName;
// Devuelve - undefined
```

Si llamamos a **miFuncion** directamente lo estaríamos llamando **sin contexto** por lo que la variable ***this*** tendría el **objeto global** dentro de miFuncion, ¿como podemos hacer que ejecute ***miFantastic*** pero pasándole ***fantasticFour*** como this? Para ésto tenemos las funciones **.call() y .apply()**, empecemos por la función .call().

Trabajando con el contexto → call()

La función ***.call()*** recibe los mismos argumentos que la función más uno, el valor que tendrá this que se pasa antes que los demás argumentos. Es decir, nuestra función ***miFantastic*** no recibe ningún argumento así que si llamamos a su método ***.call()*** y le pasamos lo que queremos que sea ***this*** es decir: ***fantasticFour***, así conseguiremos que el método funcione igual que si lo hubiésemos llamado con ***fantasticFour.miName***

```
miFantastic.call(fantasticFour);
// Devuelve - Reed Richards
```

Vamos a probar lo mismo con una función que reciba argumentos:

```
var fantasticFour = {
 nombre: "Reed Richards",
 saludar: function (amigo1, amigo2) {
console.log("Hola " + amigo1 + " y " + amigo2 + ", yo soy " + this.nombre); }
};

var miFantastic = fantasticFour.saludar;

miFantastic.call(fantasticFour, "Sue Storm", "Johnny Storm");
// Devuelve - Hola Sue Storm y Jhonny Storm, yo soy Reed Richards
```

Trabajando con el contexto → apply()

El método ***.apply()*** actúa de forma bastante **similar** a ***.call()***, pero con una variación, **solo recibe dos argumentos**, el **primero** es el **contexto** de la función, el valor de this y el **segundo** será **un array** que contendrá los argumentos que se le pasarán a la función, veamos su uso en el ejemplo anterior:

```
miFantastic.apply(fantasticFour, [ "Ben Grimm", "Sue Storm" ]);
```

Trabajando con el contexto → Bind()

Recibimos un argumento, el **contexto** que se le podrá a la función sobre la que se aplica el ***.bind()*** y devolverá una función que cuando sea llamada ejecutará la función original con el contexto que se le pasó a **.bind()**. Lo veremos mejor con un ejemplo:

```
var xMen = {
     nombre: "Jubilee"
};

function myXmen() {
  console.log(this.nombre);
}
myXmen(); // undefined

var myXmenBind = myXmen.bind(xMen);

myXmenBind();
// Jubilee
```

Ves la diferencia, bien, pero entonces ¿en que casos puedo utilizar esto? Cuando queremos **compartir la función** de un **objeto** para que **otro la utilice pero con sus propios argumentos**, veamos el ejemplo.

```
var dragonBall = {
  name: 'Son',
  lastname: 'Goku',
  fullname: function() {
    return this.name + ' ' + this.lastname;
  }
}

var mySon = {
  name: 'Son',
  lastname: 'Gohan'
}

var myDragonBall = dragonBall.fullname.apply(mySon);

console.log(myDragonBall);
// Devolvemos Son Gohan
```

Otro caso donde podemos usar algunos de estos métodos es “Function currying”, ***.bind()***, veamos a que se refiere este termino.

```
function sumando(numA, numB) {
  return numA + numB;
```

```
}

const sumandoCopy = sum.bind(this, 2);

sumandoCopy(3); //5
```

Como vemos la función ***sumando()*** recibe dos valores por parámetro y devuelve la suma de ambos. Después hemos declarado otra función, ***sumandoCopy()***, en la que le **pasamos el contexto y un valor por parámetro.** El contexto que le pasamos es el mismo que tiene por eso ***this*** y como segundo parámetro asignamos **un default al parámetro *numA***, en este caso, es decir, al colocarlo, la copia de la función en memoria podría verse así:

```
function sumandoCopy(a, b) {
  a = 2;
  return a + b;
}
```

Primero le hemos pasado el contexto y el primer valor por defecto y después hemos llamado a la función pasándole el parámetro restante.

Combinando clousure y contexto

Vamos a ver el funcionamiento de this con los clousures con un ejemplo:

```
function nuevoPokemon() {
  // Nueva variable
  var pokemon = 'Charmander';
  // Función clousure
  function closurePokemon() {
    // Accedemos a la variable del padre
    console.log(pokemon);
  }
  // Devolvemos la función del clousure
  return closurePokemon;
}
```

```
// Nueva variable con el valor de la función nuevoPokemon()
var miPokemon = nuevoPokemon();
// Ejecutamos la función
miPokemon();
```

En este caso ***miPokemon()*** se convirtió en un closure que incorpora la función **closurePokemon()** y el valor ***Charmander*** almacenado en a cuando la función fue creada.

Un poco lío verdad? pero y si os añadimos un ejemplo más práctico para que veáis la utilidad? Vamos a ello:

```
function changePokemon(pokemon) {
  return function() {
    console.log('Mi pokemon preferido es: ' + pokemon)
  }
}
var pokemonAgua = changePokemon('Blastoise');
var pokemonPlanta = changePokemon('Bulbasur');
var pokemonElectrico = changePokemon('Pikachu');
```

Esta tres funciones ***pokemonAgua***, ***pokemonPlanta*** y ***pokemonElectrico*** permiten cambiar el nombre del pokemon en el momento de crear el closure.

Clousures → Patrón Module

La forma más popular de closures es el patrón module. Esto te permite emular variables privadas o públicas:

```
var module = (function () {
  // Esta variable es privada
  var privateProperty = 'Soy una variable privada!!!';
  return {
    publicProperty: 'Soy una variable pública',
    publicMethod: function (args) {
```

```
      // do something
console.log("Método público " + privateProperty); },
}; })();
console.log(module.privateProperty); // undefined
console.log(module.publicProperty); // Soy una variable pública
module.publicMethod(); // Método público Soy una variable privada!!!
```

Qué es el Hoisting?

Una de las particularidades de JavaScript es lo que se conoce comúnmente como hoisting. Dicha característica consiste en que con independencia de donde esté la declaración de una variable, ésta es **movida al inicio del ámbito al que pertenece**. Es decir, aunque nuestro código sea como el siguiente:

```
function foo() {
  console.log(x); // undefined
  var x = 10;
}
```

Realmente se tratará a todos los efectos como si hubiésemos escrito:

```
function foo() {
  var x;
  console.log(x); // undefined
  x = 10;
}
```

El hoisting muchas veces pasa inadvertido, pero debemos tener cuidado con él. Por ejemplo, supongamos el siguiente código:

```
var x = 'global value';
function foo() {
  console.log(x);
```

```
  // undefined
  var x = 'local value';
  console.log(x);
  // local value
}
foo();
```

Uno podría esperar que se imprimiese primero “global value” y luego “local value”, ya que parece que cuando se ejecuta el primer console.log(x) la variable x local todavía no existe, por lo que se imprimiría el valor de la variable x global.Pero no ocurre esto. En su lugar dicho **código muestra “undefined” y luego “local value".**

Es importante además recalcar que, a diferencia de otros lenguajes, el **código dentro** de las llaves de un **if** o de un **for no** abre un **ámbito nuevo** (al menos no cuando usamos var).

Hoisting: let vs. var → To ES6

Sin embargo desde la aparición de l**et (ES6) podemos definir las variables dentro de un ámbito más restrictivo**.

Veíamos que var restringe su ámbito a la función en la que se encuentra. Si no tiene función será global. Sin embargo let restringe su ámbito a las llaves en las que está definido. Ya sea un if, un for, un while, una función...

Se puede decir que let es más restrictivo, más seguro y al liberar su memoria antes, también es más eficiente.

```
var variableVar = "Soy un VAR";
let variableLet = "Soy un LET";
if(true) {
   var variableVar = "Nuevo valor de VAR";
   let variableLet = "Nuevo valor de LET";
}
```

```
console.log(variableVar);
// Nuevo valor de VAR
console.log(variableLet);
// Soy un LET
```

Hoisting: Const → To ES6

Const es igual que *let*, con una pequeña gran diferencia: no puedes re-asignar su valor.

```
function explainConst(){
 const x = 10;
 console.log(x); // output 10
}
```

¿Qué pasa si tratamos de re asignar la variable *const*?

```
function explainConst(){
 const x = 10;
 console.log(x); // output 10
 x = 20; //throws type error
 console.log(x);
}
```

La consola mostrara un error cuando tratemos de re-asignar el valor de una variable *const*.

Después de esto toca explicar la nueva manera de declarar funciones, las **arrow functions** es una manera de clarar **funciones** más **compacta** y además nos ayudará a **ententer** mejor los problemas del **scope**.

Funciones flecha → Es6

Lo primero ver la sintaxis :

```
const sendMessage = () => 'hi upgraders';

// Devuelve - Esto es una función que devuelve una cadena.
```

> Cuando se componen de una **única sentencia no** necesitamos emplear la palabra clave `return` El **valor** que **devuelven** será aquel empleado **como cuerpo de la función**.

Por tanto, cuando invoquemos nuestra función sendMessage , lo que obtendremos será el texto `hi upgraders` :

```
const sendMessage = () => 'hi upgraders';

const message = sendMessage();

console.log(message);

// hi upgraders
```

Un aspecto interesante de estas funciones compuestas por una única línea es que si queremos **devolver un objeto**, deberemos emplear los **paréntesis** para rodear el **cuerpo de la función**:

```
❌ const student = () => { name: 'Peter Parker', age: '16' };

✅ const student = () => ({ name: 'Peter Parker', age: '16' });
```

Por supuesto, las *arrow functions* pueden componerse de más de una línea, lo cual ya nos obligará a emplear los **paréntesis** y a emplear `return` para, ahora sí, especificar el valor de retorno:

```
const sendMessage = () => {
  const message = 'hi upgraders';
  return message;
}
```

Argumentos en las arrow functions

Por otra parte, cuando queramos pasar **parámetros** a este tipo de funciones podremos hacerlo de dos formas. En el caso de que nuestra función tenga **un único argumento no** será necesario especificar los **paréntesis**. Por ejemplo:

```
const double = x => x*2;

const result = double(3);

//Devuelve - 9
```

En el **resto** de **casos** sí que necesitaremos especificar los **paréntesis**:

```
const multi = (a, b) => a * b;

const multiplication = multi(1, 2);

// Devuelve - 3
```

Por lo demás, esta clase de funciones también nos permite especificar **valores por defecto**:

```
//b simepre será 2

const pow = (a, b = 2) => Math.pow(a, b);

const operation = pow(2);
```

High Order Functions

Las ***arrow functions*** también nos permiten simplificar la forma en que trabajamos con *high order functions*. Es una **función** que **devuelve** una **nueva función**. Por ejemplo:

```
const generateAMultiplier = function(a) {
  return function(b) {
    return b * a;
  }
};

const multiplier = generateAMultiplier(5);

const foo = multiplier(10);

// Devuelve - 50
```

Este tipo de declaraciones que empleando la sintaxis “antigua” quedan muy largas, con las arrow functions se simplifican enormemente:

```
const generateAMultiplier = a => b => a * b;

const multiplier = generateAMultiplier(5);

const foo = multiplier(10);
```

```
// Devuelve - 50
```

> En general, el valor de ***this*** está determinado por cómo se invoca a la función. No puede ser establecida mediante una asignación en tiempo de ejecución, y puede ser diferente cada vez que la función es invocada. Las arrow functions no proporcionan su propio "binding" de ***this*** (se mantiene el valor de this del contexto léxico que envuelve a la función).

Limitaciones de las arrow functions

Sin embargo, las *arrow functions* tienen algunas limitaciones. Entre ellas me gustaría destacar las siguientes.

No podemos emplearlas para construir objetos. Por tanto, si intentamos algo de este estilo:

```
const MyClass = () => {};
const object = new MyClass();
```

Obtendremos un precioso `TypeError` .

No pueden ser usadas como funciones generadoras. Las *arrow functions* no admiten la palabra `yield` dentro de su cuerpo, por lo que si queremos crear una función generadora deberemos seguir recurriendo a la forma habitual: `function*` .

El método call y apply es ignorado. Los métodos `call` o `apply` nos permiten modificar el valor de `this` dentro de una función

Ejercicio

Sign in

GitLab.com

https://gitlab.com/upgrade-hub/ejercicios-master/javascript/javascript-sesion-2

Continuamos con la sesión 3: