Curso de Closures y Scope en JavaScript

# Qué es el Scope y cómo funciona el Global Scope

Scope: Es el alcance que va a tener una variable dentro del código. En otras palabras, el Scope se encargará de decidir a qué bloques de código va a acceder una variable.

**Global Scope** : No están dentro de funciones o bloques, por lo tanto se puede acceder a ellas de manera global.

* Con var podemos re-asignar una variable pero es una mala práctica.
* Con let y const no podemos, aparecerá un error.
* Es una mala práctica crear una variable sin las palabras reservadas: var, let y const.  
  Si se asigna una variable dentro de una función sin las palabras reservadas será una variable global.
* La doble asignación de una variable también es una mala práctica.

# Local Scope

Lexical Scope / Ámbito Léxico: El intérprete de JavaScript funciona desde el ámbito de ejecución actual y funciona hasta encontrar la variable en cuestión. Si la variable no se encuentra en ningún ámbito, se genera una excepción.

Este tipo de búsqueda se llama ámbito léxico. El alcance de una variable se define por su ubicación dentro del código fuente, y las funciones anidadas tienen acceso a las variables declaradas en su alcance externo. No importa de dónde se llame una función, o incluso cómo se llama, su alcance léxico depende solo de dónde se declaró la función.

# Function Scope

Un pequeño resumen sobre cómo actúan las diferentes variables:

* Las variables escritas con la palabra clave **var** pueden ser redeclaradas, pero esto a futuro puede darnos problemas, así que es mejor no usarla.
* Las variables escritas con la palabra clave **let** no pueden ser redeclaradas, si lo haces mostrara un ***“error: esta variable ya ha sido declarada”*** , pero su valor puede ser reasignado:

**let** fruit = "apple";

fruit = "banana";

console.log(fruit); // banana

* Las variables escritas con la palabra clave **const** no pueden ser redeclaradas o reasignadas, ya que const quiere decir que su valor será constante, es decir que no va a cambiar.

# block Scope

A diferencia del scope local este scope está limitado al bloque de código donde fue definida la variable. Desde ECMAScript 6 contamos con los keyword let y const los cuales nos permiten tener un scope de bloque, esto quiere decir que las variables solo van a vivir dentro del bloque de código correspondiente.

EXTRA:

Con var, tiene un scope de función y solo un enlace compartido para todas sus iteraciones de bucle, es decir, i en cada callback setTimeout significa la misma variable que finalmente es igual a 6 después de que finaliza la iteración del bucle.

Con let tener un scope de bloque y cuando se utiliza en el ciclo for obtiene un enlace nuevo para cada iteración, es decir, el i en cada callback setTimeout significa una variable diferente, cada una de las cuales tiene un valor diferente: la primera es 0, la el siguiente es 1 etc.

Ahora, ¿Pero por qué me devuelve 10 veces 10? ¿No debería devolverme 10 veces 9?  
veamos la declaración del ciclo for:

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

...

}

El ciclo for termina cuando la condición (i < 10) sea falsa, osea que mientras sea verdadera el recorrerá el ciclo. La variable i aumentará su valor en 1 (i++) por cada iteración, osea que tomará estos valores: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10). Toma el 10 porque es ese el valor donde la condición (i < 10) es falsa puesto que 10 no es menor que 10, si no que es igual… y el ciclo termina.  
Espero haberles ayudado y que hayan aclarado sus dudas!

# ¿Qué es un closure?

Bueno quisiera empezar con comentar que me costó entender esto, tuve que ver el video 2 veces para poderlo entender, me gustaría empezar con cambiar un poco los nombres de funciones y variables por algo menos confuso, y quizás mas divertido…

**const** papa = (nuevoEngendro) =>{

**var** cantidadDeHijos = 0;

**const** crearEngendro = (nuevoEngendro) =>{

cantidadDeHijos += nuevoEngendro;

console.log(`cantidad de engendritos = ${cantidadDeHijos}`)

}

**return** crearEngendro;

}

**let** traigamosHijosAlMundo = papa();

traigamosHijosAlMundo(1) // cantidad de engendritos = 1

traigamosHijosAlMundo(1) // cantidad de engendritos = 2

traigamosHijosAlMundo(1) // cantidad de engendritos = 3

Si estudiamos la función **papa** esta tiene la declaración de la variable **cantidaDeHijos** y la declaración de la función **crearEngendro** y al final retorna la función **crearEngendro**,

Ahora después de la declaración de la función papa viene la línea

**let**traigamosHijosAlMundo = papa();

Notese el **PARENTESIS** () de la función papa

**let** traigamosHijosAlMundo = papa(); // <-- este paréntesis indica que **la** función **se** está ejecutando

Este paréntesis indica que la función se está ejecutando, es decir cada línea de esa función se ejecuta una a una, y si recorremos línea a línea a la función tenemos  
la declaración de **cantidadDeHijos** que vale 0  
la declaración de la función **crearEngendro** y el retorno de esta función,

así que finalmente **traigamosHijosAlMundo** vale lo que haya retornado la función **papa** que es **crearEngendro** (nótese que no se ejecuta la función crearEngendro sino que simplemente se retorna la referencia a ella)

por ende cada vez que ejecutamos

**traigamosHijosAlMundo**(1)

Realmente estamos es llamando a la función **crearEngendro** con todo su ámbito que es la variable cantidadDeHijos con valor de 0, y a este le suma lo que se le pase por parámetro,

En la segunda ejecución de

**traigamosHijosAlMundo**(1)

no se está volviendo a correr todas las líneas de la función papa, esto ya se hizo en la asignación ( let traigamosHijosAlMundo = papa(); ), sino que realmente se está volviendo a llamar a **crearEngendro()** la cual había modificado su variable cantidadDeHijos en la primera llamada

asi que pienso que la clave es entender que en esta asignación

**let**traigamosHijosAlMundo = papa();

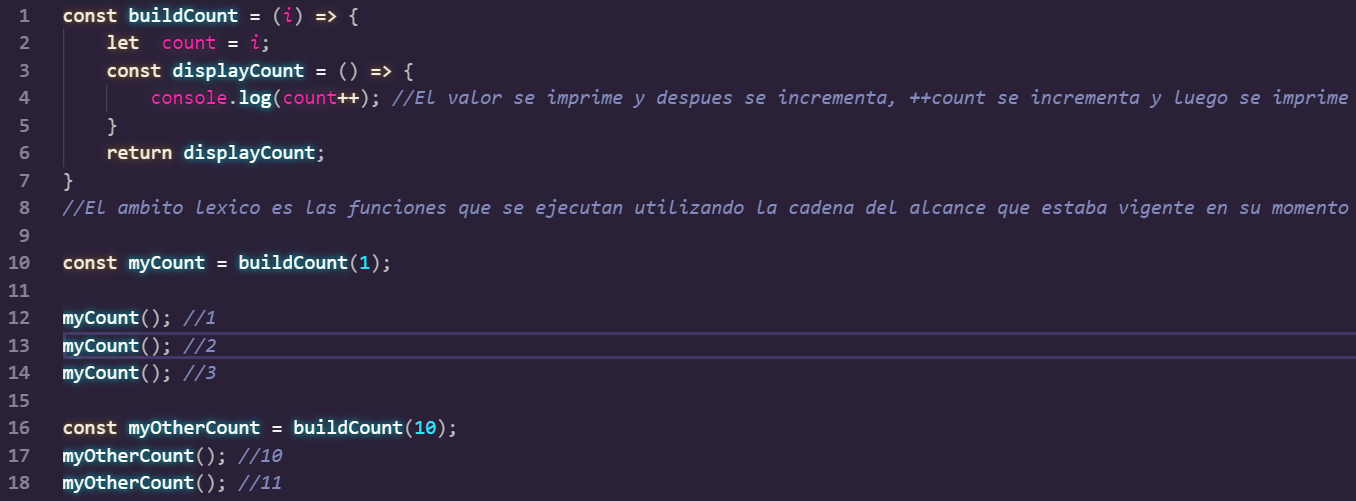
se ejecutó la función papá donde se declararó la variable de dicha función (cantidadDeHijos) y la función que retorna (crearEngendro) UNA SOLA VEZ! lo que se ejecuta multiples veces es el la función crearEngendro

# Ámbito léxico en closures

El ámbito léxico es cuando las funciones se ejecutan utilizando la cadena del alcance donde estaban vigente en su momento.

Esto significa que podemos acceder al valor “count” dentro de la función porque es el alcance donde está asignado.

Podemos tener varias formas de manejar la constante “buildCount”, significa que la podemos asignar a myCount y myOtherCount. Trabajaremos con el scope(alcance) que tiene esta variable, poder ejecutarla y empezar a contar desde donde necesitemos.



# Cómo crear variables privadas con closures

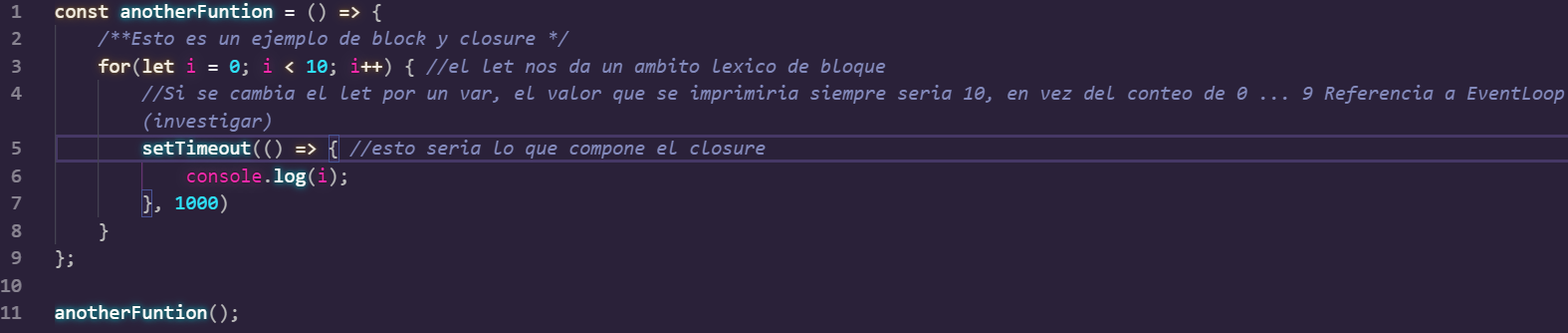
Variables privadas con Closures: JS por su naturaleza no fomenta el uso de datos privados pero por medio de los Closures podemos crear valores que solo puedan ser accedidos por medio de métodos, que no van a estar disponibles fuera de esta función.



# loops

Variables privadas con Closures: JS por su naturaleza no fomenta el uso de datos privados pero por medio de los Closures podemos crear valores que solo puedan ser accedidos por medio de métodos, que no van a estar disponibles fuera de esta función.

Para este ejemplo lo importarte a notar es que cuando usamos let en un loop este valor no puede ser reasignado como si usáramos var, resultando para el proceso asíncrono una variable con valor independiente a mostrar ya que genera un nuevo scope local de forma interna para cada ciclo.



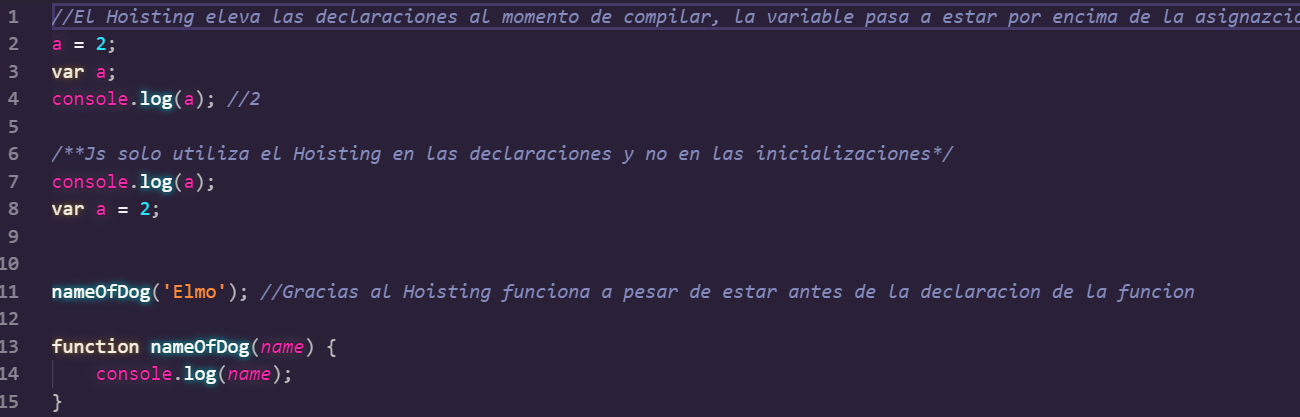
# ¿Qué es el hoisting?

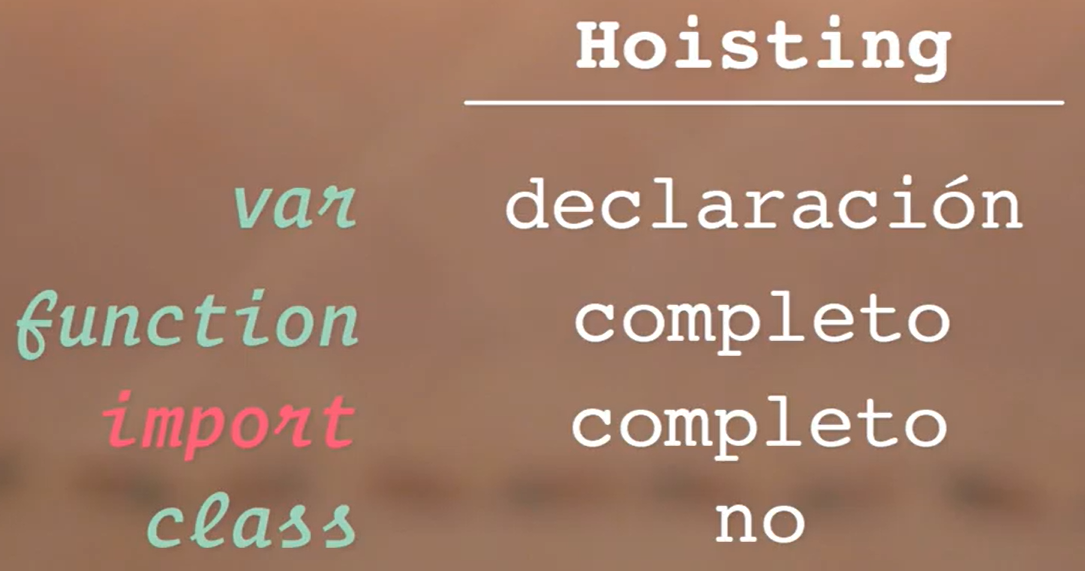
**El hosting:** Es el trabajo que hace el interpreter del engine de javaScript (motor de javaScript) que esta por defecto en nuestros navegadores, cuyo propósito es levantar todas nuestras funciones y variables inicializadas al principio de nuestro código, todo con el fin de poderlo optimizar.

**Nota:** para que el hoisting no te de una mala pasada, por buenas costumbres manda a llamar primero las variables y funciones en tu editor de código.

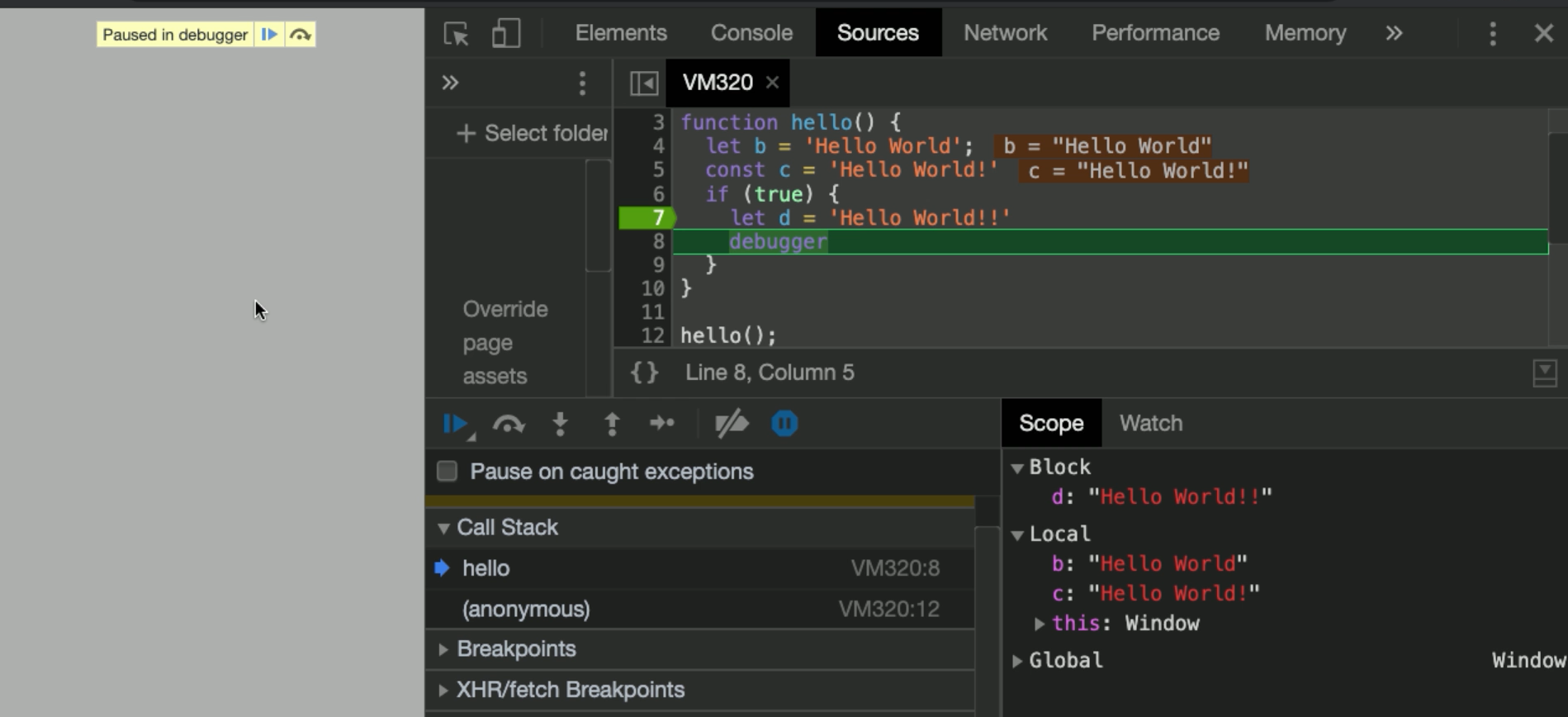
Lo que se hace en realidad es tomar ‘registros’ en memoria de donde está cada declaración (todo esto previo a que se ejecute el código en sí) y depende si es var, let, const o una función, JS va a asignarle referencias a cada una.  
Si es:

* var : asigna la referencia undefined (si de acá viene el famoso undefined)
* let/const: asigna la referencia uninitialized(declarado pero no inicializado)
* función: guarda un registro con la función entera (por eso la podemos llamar antes de que este creada).





# Debugging



moneyBox()

