

Curso de Closures y Scope en JavaScript

Las variables **globales** se instancian en el objeto window.

Si no declaras una variable con var, let, const y sólo la asignas, por defecto va a pasar a ser una variable global.

Ejemplo: Una variable sin declarar dentro de una función pasa a ser una variable global.

```
function myCountries() {  
  country = 'Mexico';  
  console.log(country);  
}  
myCountries();  
console.log(country);
```

El **scope local de función** (function scope) es el entorno donde las variables locales solo se pueden acceder desde una función del programa. Es decir; las variables declaradas dentro de una función se vuelven locales a la función así como los parametros.

En este scope no importa que las variables sean declaradas con var, let o const. Sin embargo, ten presente que se puede redeclarar una variable con var, pero no con let y const.

Según la cadena de scope, si existe una función dentro de otra función, la función hijo podrá acceder a las variables de la función padre, pero NO en viceversa. Recuerda esto en el tema de los **Closures**.

El **scope local de bloque** es el entorno donde las variables locales únicamente pueden ser accedidas desde un bloque de código del programa. Un bloque de código es todo aquello que está dentro de los caracteres de llaves {}.

Hoisting es un término para describir que la declaración de variables y funciones son desplazadas a la parte superior del scope más cercano.

var no tiene scope de bloque y se debe tener cuidado porque puede provocar errores en el código. Con excepción dentro de una función, ahí **var** sí se vuelve **local** la función.

Importante ejemplo de un bloque for con **var** con asincronismo.

La respuesta es **diez veces 10**, y sucede por el **hoisting**. La declaración de i se eleva hasta arriba de la función en el scope de función, por lo que cuando termine el ciclo este tendrá un valor de 10.

```
function example() {  
  for (var i = 0; i < 10; i++){  
    setTimeout(function(){  
      console.log(i);  
    }, 1000);  
  }  
}  
example();
```

El **modo estricto** es una funcionalidad que le permite al motor de JavaScript cambiar la manera en que ejecuta el código. En este modo, se reduce las cosas que podemos hacer, esto es bueno porque permite manejar errores que son poco perceptibles o que el motor de JavaScript sobreentiende y ayuda a su compilación para corregirlos.

Se usa en el código colocando en la primera línea **"use strict"** para todo el archivo. También puede utilizarse en la primera línea de una función, pero no para un bloque en específico.

Ejemplo usando el modo estricto: En modo estricto, no te permitirá realizar esto y provocará un error.

```
"use strict";

nombre = "Andres"
console.log(nombre) // ReferenceError: nombre is not defined
```

Ejemplo usando el modo estricto en una función: En modo estricto, no te permitirá realizar esto y provocará un error.

```
"use strict";

function myFunction(){
  return pi = 3.14
}

console.log(myFunction()) // ReferenceError: pi is not defined
```

Un **closure** es la combinación entre una función y el ámbito léxico en el que esta fue declarada. Con esto, la función recuerda el ámbito con el cual se creó. Puedes entender los closures como: función interna + scope. Mira estos pasos:

1. Genera una función que retorna una función interna.
2. Esta función interna tiene un scope, el cual puede ser accedido únicamente por esta función, es decir, las variables, funciones, etc. definidas en el scope solo pueden ser accedidas por la función interna.
3. Como resultado, esta función interna retornada con su scope será nuestro closure.

IMAGEN DE EJEMPLO ABAJO

CLOSURE = FUNCTION + SCOPE

```
const moneyBox = () => {
  var saveCoins = 0;
  let another1 = 'x';
  const another2 = () => {};
  const countCoins = (coins) => {
    saveCoins += coins;
    console.log(`MoneyBox: ${saveCoins}`);
  }
  countCoins.var1 = 'v1';
  countCoins.f1 = () => {};
  return countCoins;
}

//creando el closure 'myMoneyBox'
let myMoneyBox = moneyBox();
myMoneyBox(4); //4
myMoneyBox(6); //10
myMoneyBox(10); //20
```

Scope

Function

attributes of Function

return Function

* myMoneyBox is the Closure with its respective scope.

* myMoneyBox is the function countCoins(coins).

Benefit: **variables in scope are private.**

El **ámbito léxico** se refiere al alcance de una variable siguiendo la cadena de scopes. Una variable se puede abordar desde un nivel inferior hasta uno superior, pero no al contrario.

```
1  const myGlobal = 0;
2
3  function myFunction() {
4    const myNumber = 1;
5    console.log(myGlobal);
6
7    function parent() {
8      const inner = 2;
9      console.log(myNumber, myGlobal);
10
11     function child() {
12       console.log(inner, myNumber, myGlobal);
13     }
14
15     return child();
16   }
17
18   return parent();
19 }
20
21 myFunction();
```

Los **closures** son básicamente cuando aprovechamos la habilidad de JavaScript de emplear las variables que están en el scope padre de nuestro bloque de código, por eso el global scope es un closure grande.

Si tú declaras la variable **saveCoins** en el global scope, estarías usando el mismo principio que si emplearas la segunda función de los ejemplos anteriores porque estás usando las variables que están en el scope padre.

```
var saveCoins = 0;

const moneyBox = (coins) => {
  saveCoins += coins;
  console.log(saveCoins);
}

moneyBox(5); //5
moneyBox(10); //15
```

Sin embargo, está mal visto modificar variables globales, por eso es que quieres crear variables dentro de un scope cerrado y que interactúen entre ellas. Entonces, declaras las variables que vas a usar dentro del **scope padre** del bloque que las va a modificar para que siempre pueda acceder a ellas.

Para eso originas un nuevo “global scope” ficticio que va a conservar todas las variables que tú quieras monitorear. Ahora mira las similitudes entre el código de arriba y el que está justo abajo de aquí:

```
const moneyBox = () => {
  var saveCoins = 0;
  const countCoins = (coins) => {
    saveCoins += coins;
    console.log(saveCoins);
  }
  return countCoins;
}

let myMoneyBox = moneyBox()
myMoneyBox(4)
myMoneyBox(10)
myMoneyBox(6)
```

Lo que estás haciendo es simplemente bajar un nivel tu scope. Quieres que la función moneyBox regrese una función que estuvo declarada dentro de sí misma porque esa función tiene acceso a ese scope que ya no va a existir para que alguien más lo utilice, solamente lo podrá emplear la función countCoins.

Al guardar el resultado de moneyBox (countCoins) en otra variable estás generando el ámbito léxico que menciona el profesor, necesario para no dejar morir ese scope.

Hoisting es un término para describir que las declaraciones de variables y funciones son desplazadas a la parte superior del scope más cercano, scope global o de función. En el caso de una función esto sucede solamente con las declaraciones (el definir una función) y no con las asignaciones (asignarla a una variable).

El código permanece igual, solo es una interpretación del motor de JavaScript. En el caso de las variables solamente sucede cuando son declaradas con **var**.

Hoisting de variables dentro de una función: El hoisting desplaza las declaraciones a la parte superior del scope más cercano, en el caso de una función seguiría el siguiente comportamiento. Si tenemos esto (imagen izquierda), JavaScript lo interpretaría como lo siguiente (imagen derecha):

```
function scope() {  
  console.log(nombre) // undefined  
  console.log(edad) // undefined  
  console.log(i) // undefined  
  
  var nombre = "Andres"  
  var edad = 20  
  for (var i = 0; i < 6; i++) {  
    //...  
  }  
}
```

```
function scope() {  
  var nombre = undefined  
  var edad = undefined  
  var i = undefined  
  
  console.log(nombre) // undefined  
  console.log(edad) // undefined  
  console.log(i) // undefined  
  
  nombre = "Andres"  
  edad = 20  
  for ( i = 0; i < 6; i++) {  
  
    //  
  }  
}
```

Hoisting en funciones asignadas a variables: Mira el siguiente código y piensa cuál sería el resultado del console.log.

IMAGEN ABAJO

```

console.log( saludar() )

var saludar = function saludar() {
  return "hola"
}

```

La respuesta es un **error de tipo** porque si asignas una función a una variable declarada con **var**, y la invocas antes de declararla, la variable será de tipo **undefined** y no de función por el hoisting.

Por el **hoisting** la imagen de arriba sería lo equivalente a esto:

```

var saludar = undefined

console.log( saludar() ) // TypeError: saludar is not a function

saludar = function saludar() {
  return "hola"
}

```

Hoisting con let y const: Aunque te haya dicho que el hoisting solo ocurre con declaraciones con var, no es totalmente cierto. El hoisting hará que el intérprete de JavaScript eleve las declaraciones con let y const a la **Temporal Dead Zone**. La Temporal Dead Zone es una región del código donde la variable está declarada, pero no es posible acceder a esta, provocando un error de tipo **ReferenceError**.

La Zona Muerta Temporal (Temporal Dead Zone), explicada: Esto es lo que es la TDZ: El término describe el estado donde las variables son inaccesibles. Se encuentran en el scope, pero no han sido declaradas todavía. Las variables **let** y **const** existen en la TDZ desde el inicio de su ámbito de aplicación, su scope, hasta que son declaradas.

```

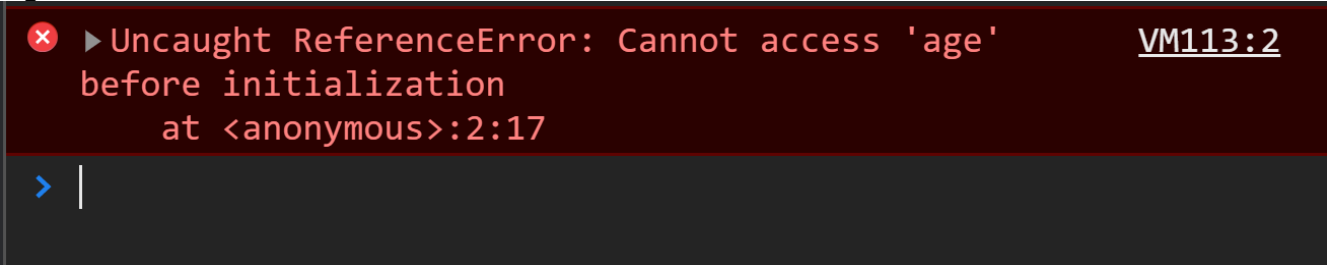
{
  // ¡Esta es la Zona muerta temporal para la variable age!
  // ¡Esta es la Zona muerta temporal para la variable age!
  // ¡Esta es la Zona muerta temporal para la variable age!
  // ¡Esta es la Zona muerta temporal para la variable age!
  let age = 25; // Cuando llegamos aquí, se acabó TDZ.
  console.log(age);
}

```

Puedes ver arriba que, si se quiere acceder a la variable age, antes de su declaración, nos arrojará un ReferenceError. Por la TDZ.

¿Por qué se crea la TDZ cuando se crea?

Si volvemos al ejemplo (imagen), Si añadimos un console.log dentro de la TDZ veremos el siguiente error:



```
✖ ▶ Uncaught ReferenceError: Cannot access 'age' before initialization
    at <anonymous>:2:17
VM113:2
```

¿Por qué existe la TDZ entre en inicio del scope y la declaración de la variable?

¿Cuál es la razón específica para ello?

Es por el hoisting (elevación):

El motor de JS que está analizando y ejecutando tu código tiene 2 pasos que realizar:

1. Analizar el código en el Arbol de Sintaxis Abstracto/código de bytes ejecutable, y
2. Ejecución en tiempo real.

En el paso uno es cuando sucede el hoisting, realizado por el motor de JS.

La única diferencia con **const** y **let** es que, cuando son elevadas, sus valores **no** toman el valor por defecto **undefined**.

Conclusión: Cuando las variables son elevadas, **var** obtiene el valor de inicialización undefined por defecto en el proceso de hoisting o elevación. Por su parte **let** y **const** también son elevadas, pero no son puestas con el valor undefined en el proceso.

Y este es el único motivo por el que tenemos TDZ. Por eso ocurre con let y const pero no con var.

Siempre asegúrate de definir tus lets y consts en la parte superior del scope.

Debugging es el término para solucionar bugs. Los bugs (“bichos” en inglés) son errores en la aplicación.

Todo navegador dispone de Dev tools o herramientas de desarrollador, que es un conjunto de características del código de la página web, una de estas es el debugging.

La palabra reservada **debugger** sirve para detener la ejecución del programa, pero solo funciona si el panel de las herramientas de desarrollo está abierto. Este panel te mostrará información sobre el código hasta la línea del debugger.

Los **breakpoints** son puntos donde la ejecución del programa se parará. Para activarlos se debe dar clic en la línea de código que se desea parar.

Si el código tiene **closures**, aparecerán en el panel “Scope”.