Apuntes JavaScript – Curso BluuWeb

## Hoisting

Hoisting es un término para describir que las declaraciones de variables y funciones son movidas a la parte superior del scope más cercano: scope global o de función. Esto sucede solamente con las declaraciones y no con las asignaciones.

una estricta definición de hoisting sugiere que las declaraciones de variables y funciones son físicamente movidas al comienzo del código, pero esto no es lo que ocurre en realidad. Lo que sucede es que las declaraciones de variables y funciones son asignadas en memoria durante la fase de compilación, pero quedan exactamente en dónde las has escrito en el código.

Texto

Descripción generada automáticamente

El código escrito arriba está escrito de la manera que sería esperada para que funcione. Ahora, veamos qué sucede cuando llamamos a la función antes de escribirla:

Texto

Descripción generada automáticamente

Como se puede observar, aunque primero llamamos a la función en el código, antes de que sea escrita, el código aún funciona. Esto sucede por la manera en la que el contexto de ejecución trabaja en JavaScript

En este artículo, vamos a aprender acerca de dos conceptos,

* Contexto de ejecución
* Hoisting

Como principiante en JavaScript, comprender estos conceptos te ayudará a comprender las palabras clave this, scope y closure de manera mucho más cómoda. Así que disfruta y sigue leyendo.

# **Contexto de ejecución en JavaScript**

En general, un archivo fuente de JavaScript tendrá varias líneas de código. Como desarrolladores, organizamos el código en variables, funciones, estructuras de datos como objetos, matrices, y más.

Un Ámbito Léxico determina cómo y dónde escribimos nuestro código físicamente. Eche un vistazo al siguiente código:

function doSomething() {

var age= 7;

// Some more code

}

En el código anterior, la variable edad está léxicamente dentro de la función doSomething.

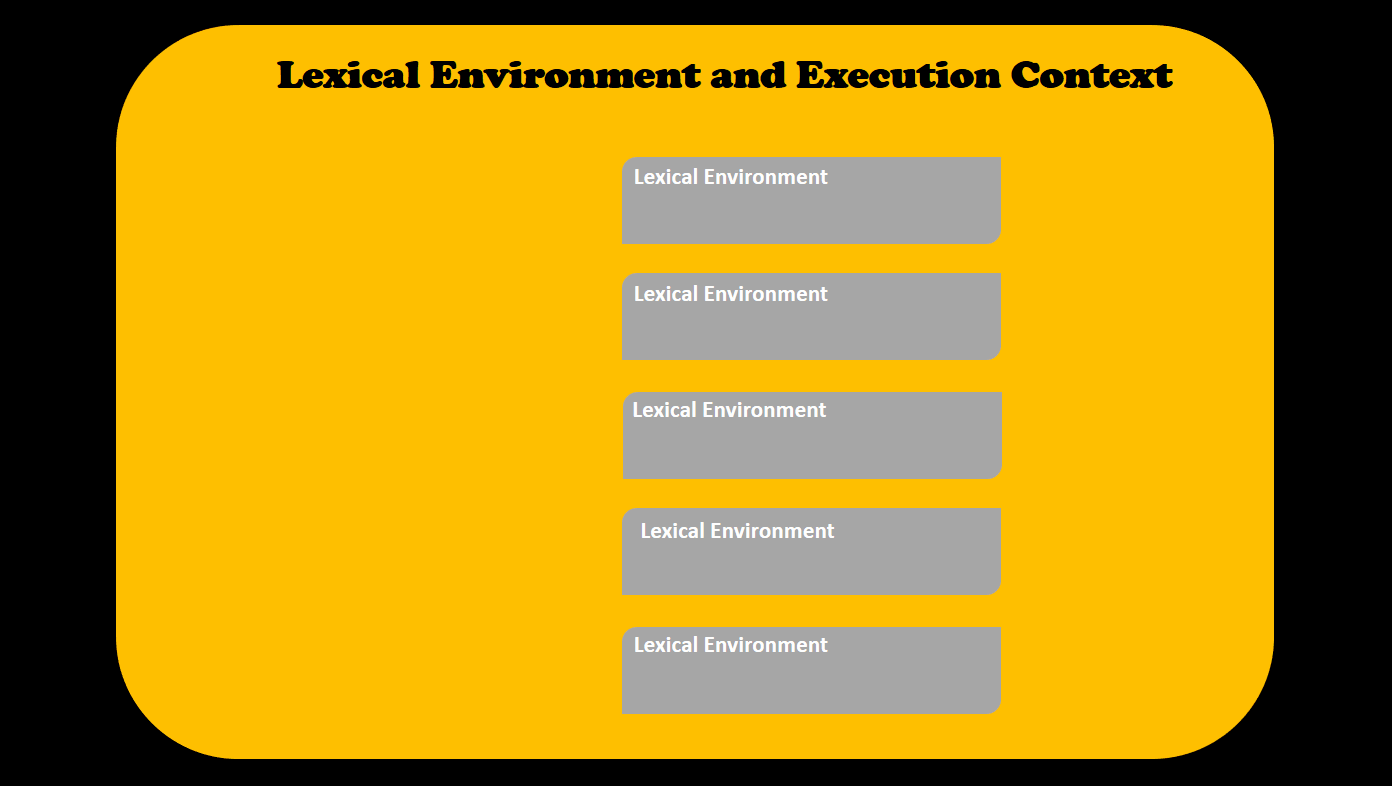
Tenga en cuenta que nuestro código no se ejecuta como está. Tiene que ser traducido por el compilador a un código de bytes comprensible por computadora. Por lo tanto, el compilador necesita mapear lo que está colocado léxicamente en un lugar significativo y válido.

Por lo general, habrá más de un Ámbito Léxico en su código. Sin embargo, no todos los ámbitos se ejecutan a la vez.

El ámbito que ayuda a que se ejecute el código se denomina Contexto de Ejecución. Es el código que se está ejecutando actualmente y todo lo que lo rodea que lo ayuda a ejecutarlo.

Puede haber muchos Ámbitos Léxicos disponibles, pero el Contexto de Ejecución administra el código que se está ejecutando actualmente.

Mira la imagen a continuación para comprender la diferencia entre un Ámbito Léxico y un Contexto de Ejecución:

Ámbito Léxico vs Contexto de Ejecución

Entonces, ¿qué sucede exactamente en el contexto de ejecución? El código se analiza línea por línea, genera código de bytes ejecutable, asigna memoria y se ejecuta.

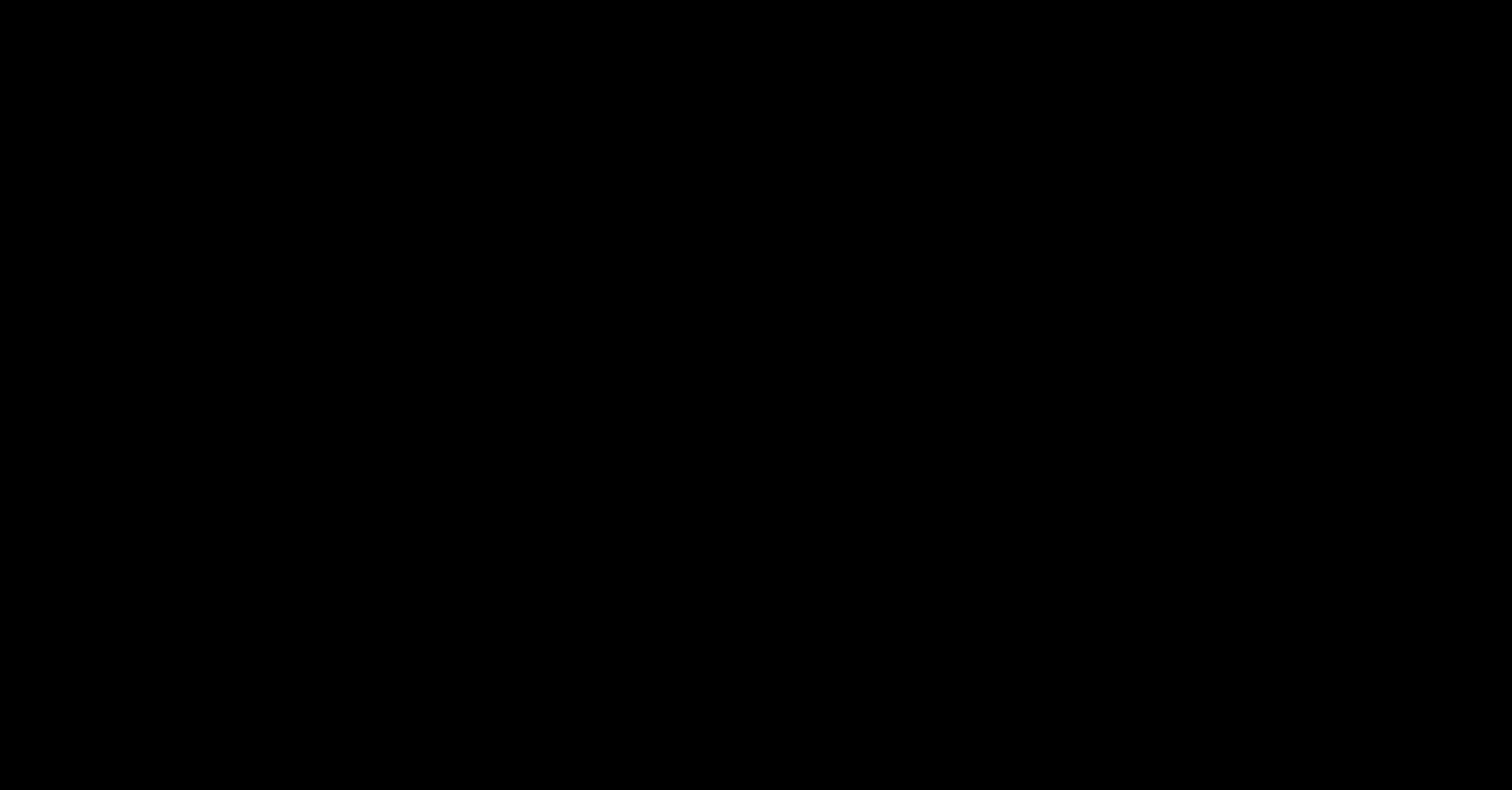
Tomemos la misma función que hemos visto anteriormente. ¿Qué crees que puede pasar cuando se ejecuta la siguiente línea?

var age = 7;

Hay muchas cosas que suceden detrás de escena. Ese fragmento de código fuente pasa por las siguientes fases antes de que finalmente se ejecute:

* **Tokenización:** en esta fase, la cadena de código fuente se divide en varios fragmentos significativos llamados Tokens. Por ejemplo, el código var age = 7; se tokeniza en var, age, =, 7 y,;.
* **Parsing:**La siguiente fase es el análisis, donde una matriz de tokens se convierte en un árbol de elementos anidados comprendidos por la gramática del idioma. Este árbol se llama AST (árbol de sintaxis abstracta).
* **Generación de código:** En esta fase, el AST creado en la fase de análisis se convierte en código de bytes ejecutable. Este código de bytes ejecutable luego se optimiza aún más mediante el compilador JIT (Just-In-Time).

La siguiente imagen animada muestra la transición del código fuente al código de bytes ejecutable.

Código fuente a código byte ejecutable

Todas estas cosas suceden en un Contexto de Ejecución. Entonces, el contexto de ejecución es el entorno donde se ejecuta una parte específica del código.

Hay dos tipos de contextos de ejecución:

* Contexto de ejecución global o Global Execution Context (GEC)
* Contexto de ejecución de funciones o Function Execution Context (FEC)

Y cada uno de los contextos de ejecución tiene dos fases:

* Fase de creación
* Fase de ejecución

Echemos un vistazo detallado a cada uno de ellos y comprendamos un poco mejor.

## **Contexto de ejecución global (GEC) en JavaScript**

Siempre que ejecutamos código JavaScript, se crea un contexto de ejecución global (también conocido como contexto de ejecución base). El contexto de ejecución global tiene dos fases.

### **Fase de creación**

En la fase de creación, se crean dos cosas únicas:

* Un objeto global llamado window (para JavaScript del lado del cliente).
* Una variable global llamada this.

Si hay alguna variable declarada en el código, se asigna un espacio en la memoria para la variable. La variable se inicializa con un valor único llamado undefined. Si hay una función en el código, se coloca directamente en la memoria. Aprenderemos más sobre esta parte en la sección Hoisting más adelante.

### **Fase de ejecución**

La ejecución del código comienza en esta fase. Aquí tiene lugar la asignación de valor de las variables globales. Ten en cuenta que aquí no se invoca ninguna función, como sucede en el contexto de ejecución de funciones. Veremos eso en un rato.

Entendamos ambas fases con un par de ejemplos.

#### **Ejemplo 1: Cargar un script vacío**

Cree un archivo JavaScript vacío con el nombre index.js. Ahora cree un archivo HTML con el siguiente contenido:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

<script src='./index.js'></script>

</head>

<body>

I'm loading an empty script

</body>

</html>

Ten en cuenta que estamos importando el archivo de script vacío en el archivo HTML usando la etiqueta <script>.

Carga el archivo HTML en el navegador y abre Chrome DevTools (generalmente usando la tecla F12) o equivalente para otros navegadores. Ve a la pestaña de la console, escribe window y presiona enter. Deberías ver el valor del objeto Window del navegador.

Objeto Window

Texto, Carta

Descripción generada automáticamente

Ahora, escribe la palabra this y presiona enter. Debería ver el mismo valor del objeto Window impreso en la consola del navegador.

Texto, Carta

Descripción generada automáticamenteValor de 'this'

### SOLO EN CHROME

Genial, ahora intenta comprobar si la window es igual a this. Sí lo es.

window es igual a 'this'

Imagen que contiene Gráfico

Descripción generada automáticamente

Muy bien, entonces, ¿qué hemos aprendido?

* El Contexto de Ejecución Global se crea cuando cargamos el archivo JavaScript, incluso cuando está vacío.
* Se crea dos cosas especiales para nosotros en su fase de creación, que es el objeto window y this.
* En el Contexto de Ejecución Global, el objeto de window y this son iguales.
* No hay nada que ejecutar, ya que el archivo de secuencia de comandos está en blanco. Entonces no pasa nada en la fase de ejecución.

#### **Ejemplo 2: Con variables y funciones**

Veamos ahora un ejemplo con algo de código en el archivo JavaScript. Agregaremos una variable (blog) con un valor asignado. También definiremos una función con el nombre logBlog.

var blog = 'freeCodeCamp';

function logBlog() {

console.log(this.blog);

}

En la fase de creación:

* Se crea el objeto global window y la variable this.
* Se asigna un espacio en la memoria  para la variable blog y la función logBlog.
* La variable blog se inicializa con un valor especial undefined. La función logBlog se coloca directamente en la memoria

En fase de ejecución:

* El valor freeCodeCamp se asigna a la variable blog.
* Como hemos definido la función, pero aún no la hemos llamado, la ejecución de la función no tiene lugar. Llamaremos a la función y veremos qué sucede cuando aprendamos sobre el Contexto de Ejecución de la Función.

## **Contexto de ejecución de funciones (FEC) en JavaScript**

Cuando invocamos una función, se crea un Contexto de Ejecución de Función. Extendamos el mismo ejemplo que usamos anteriormente, pero esta vez llamaremos a la función.

var blog = 'freeCodeCamp';

function logBlog() {

console.log(this.blog);

}

// Llamemos a la función

logBlog();

El contexto de ejecución de la función pasa por las mismas fases, creación y ejecución.

La fase de ejecución de la función tiene acceso a un valor especial llamado arguments. Son los argumentos pasados ​​a la función. En nuestro ejemplo, no se pasan argumentos.

Ten en cuenta que el objeto window y la variable this creada en el contexto de ejecución global todavía son accesibles en este contexto.

Cuando una función invoca a otra función, se crea un nuevo contexto de ejecución de función para la nueva llamada a la función. Cada uno de los contextos de ejecución de la función determina el alcance o scope de las variables utilizadas en las funciones respectivas.

# **Hoisting en JavaScript**

Espero que hayas disfrutado aprendiendo sobre el Contexto de Ejecución. Pasemos a otro concepto fundamental llamado Hoisting. Cuando escuché por primera vez sobre Hoisting, me tomó un tiempo darme cuenta de que algo andaba muy mal con el nombre de Hoisting.

En el idioma inglés, hoisting significa levantar algo usando cuerdas y poleas. El nombre puede inducirlo al error de pensar que el motor de JavaScript extrae las variables y funciones en una fase de ejecución de código específica. Bueno, esto no es lo que pasa.

Entonces, entendamos el Hoisting usando el concepto de Contexto de Ejecución.

## **Variable hoisting en JavaScript**

Eche un vistazo al siguiente ejemplo y adivina el resultado:

console.log(name);

var name;

Estoy seguro de que ya lo adivinaste. Es lo siguiente:

undefined

Sin embargo, la pregunta es ¿por qué? Supongamos que usamos un código similar en algún otro lenguaje de programación. En ese caso, es posible que obtengamos un error que indique que el nombre de la variable no está declarado y que estamos intentando acceder a él mucho antes. La respuesta está en el contexto de ejecución.

En la fase de creación,

* Se asigna un espacio en la memoria para el nombre de la variable y
* Se asigna el valor especial undefined a la variable.

En la fase de ejecución,

* Se ejecutará la instrucción console.log (name).

Este mecanismo de asignación de memoria para variables e inicialización con el valor undefined en la fase de creación del contexto de ejecución se denomina Variable Hoisting.

El valor especial undefined significa que se declara una variable, pero no se asigna ningún valor.

Si le asignamos a la variable un valor como este:

name = 'freeCodeCamp';

La fase de ejecución asignará este valor a la variable.

## **Function Hoisting en JavaScript**

Ahora hablemos de Function Hoisting. Sigue el mismo patrón que la Variable Hoisting.

La fase de creación del contexto de ejecución coloca la declaración de la función en la memoria y la fase de ejecución la ejecuta. Eche un vistazo al siguiente ejemplo:

// Invocar la función functionA

functionA();

// Declarar la función functionA

function functionA() {

console.log('Function A');

// Invocar la función FunctionB

functionB();

}

// Declarar la función FunctionB

function functionB() {

console.log('Function B');

}

El resultado es el siguiente:

Function A

Function B

* El contexto de ejecución crea la memoria para la función y coloca la declaración de función completa de functionA en ella.
* Las funciones crean su propio contexto de ejecución. Entonces sucede algo similar con la functionB.
* A continuación, las funciones se ejecutan en su contexto de ejecución respectivamente.

Poner toda la declaración de función en la memoria en la fase de creación se llama Function Hoisting.

## **Algunas reglas básicas**

Dado que ahora entendemos el concepto de Hoisting, comprendamos algunas reglas básicas:

* Siempre defina variables y funciones antes de usarlas en su código. Reduce las posibilidades de errores sorpresa y depuración de pesadilla.
* Hoisting es solo para la declaración de funciones, no para la inicialización. A continuación se muestra un ejemplo de inicialización de una función en donde la ejecución del código se romperá.

logMe();

var logMe = function() {

console.log('Logging...');

}

La ejecución del código se interrumpirá porque con la inicialización de la función, la variable logMe se elevará como una variable, no como una función. Entonces, con el levantamiento de variables, la asignación de memoria ocurrirá con la inicialización con undefined. Esa es la razón por la que obtendremos el error:

Error en hoisting de la inicialización de una función

Supongamos que intentamos acceder a una variable antes de la declaración y usamos las palabras clave let y const para declararla más tarde. En ese caso, serán elevados pero no asignados con el undefined predeterminado. Acceder a tales variables resultará en ReferenceError. Aquí hay un ejemplo:

console.log(name);

let name;

Lanzará el error:

Error con hoisting de la variable declarada con las palabras clave let y const

El mismo código se ejecutará sin problemas si usamos var en lugar de let y const. Este error es un mecanismo de protección del lenguaje JavaScript, como ya hemos comentado, ya que el hoisting accidental puede causar problemas innecesarios.