Programación con JavaScript II

Scope y this

Introducción

En este tema aprenderás cuál es el alcance de una variable dependiendo del lugar en donde se declara. En términos generales, si una variable o constante se declara dentro de un conjunto de llaves, esas llaves delimitan la región del código en la que es admitida esa variable o constante (aunque, por supuesto, no es válido hacer referencia a una variable o constante desde líneas de código que se ejecutan antes de la instrucción let o const que declara la variable). Actualmente, en las versiones de JavaScript está cayendo en abandono el uso de la palabra reservada ***var*** y algunas otras definiciones que habías aprendido hasta el momento.

Por otra parte, aprenderás las principales reglas del uso de la palabra reservada ***this***, la cual, si bien es cierto que es similar a lo que se podría encontrar en otros lenguajes, en JavaScript puede vincularse a diferentes objetos según la parte del programa en la cual se llame a la función.

Explicación

Scope

El alcance de una variable es la región del código fuente de su programa en la que se define. Flanagan (2020) indica que las variables y constantes declaradas con let y const tienen un alcance de bloque, esto significa que solo se definen dentro del bloque de código en el que aparece la instrucción let o cons, dicho de otra manera, es **alcance local**. Las definiciones de funciones y clases de JavaScript son bloques, al igual que los cuerpos de las sentencias *if/else*, bucles (*while, for*, etc.) y los conjuntos de llaves, solo posterior a su declaración podrán mandarse llamar en el mismo bloque de la función, no antes.

Observa el siguiente ejemplo declarando **var** dentro del **for**:

```
var numeros = [1, 2, 3, 4, 5];
var dobles = [];

for(var i = 0; i < numeros.length; i++) {
dobles.push(numeros[i] * 2);
}

console.log(numeros); // [1, 2, 3, 4, 5]
console.log(dobles); // [2, 4, 6, 8, 10]

console.log(i); // 5
```

Y ahora declarando **let** dentro del **for**:

```
var numeros = [1, 2, 3, 4, 5];
var dobles = [];

for(let i = 0; i < numeros.length; i++) {
dobles.push(numeros[i] * 2);
}

console.log(numeros); // [1, 2, 3, 4, 5]
console.log(dobles); // [2, 4, 6, 8, 10]

console.log(i); // Undefined
```

Cuando aparece una declaración en el nivel superior, fuera de cualquier bloque de código, se dice que es una variable o constante global y que tiene **alcance global**. En los módulos de JavaScript del lado del cliente, el alcance de una variable global es el archivo en la que está definida. Sin embargo, en el JavaScript del lado del cliente tradicional, el alcance de una variable global es el documento HTML en que se define. Es decir: si un <script> declara una variable o constante global, esa variable o constante se define en todos los elementos <script> de ese documento (o al menos en todos los scripts que se ejecutan después de que se ejecuta la instrucción let o const).

Explicación

En las versiones de JavaScript anteriores a ES6, la única forma de declarar una variable es con la palabra clave var y no hay forma de declarar constantes. La sintaxis de **var** es como la sintaxis de **let.**

```
var x;
var data = [], count = data.length;
for(var i = 0; i < count; i++) console.log(data[i]);
```

Aunque var y let tienen la misma sintaxis, hay diferencias importantes en la forma en que funcionan:

- Las variables declaradas con var no tienen alcance de bloque. En su lugar, están en el ámbito del cuerpo de la función que las contiene, sin importar qué tan profundamente anidadas estén dentro de esa función.

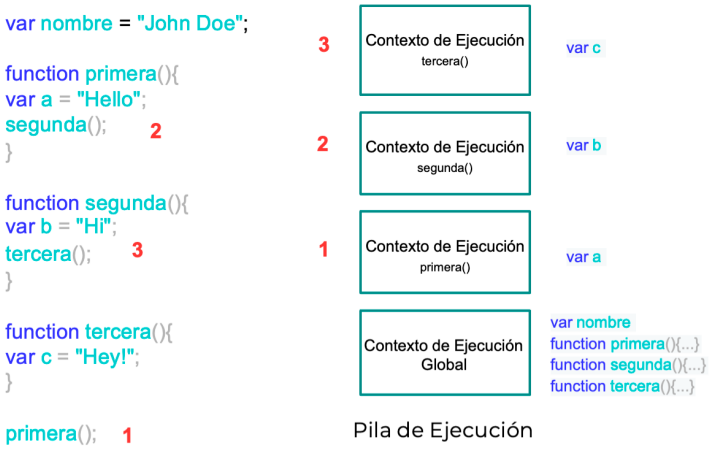
- Si usas var fuera del cuerpo de una función, declaras una variable global. No obstante, las variables globales declaradas con var difieren de las globales declaradas con let en un aspecto importante. Las globales declaradas con var se implementan como propiedades del objeto global. Se puede hacer referencia al objeto global como globalThis, así que si escribes var x = 2; fuera de una función, es como si escribieras globalThis.x = 2;. Ten en cuenta, sin embargo, que la analogía no es perfecta, las propiedades creadas con declaraciones var globales no se pueden eliminar con el operador de eliminación. Las variables y constantes globales declaradas con let y const no son propiedades del objeto global.

- A diferencia de las variables declaradas con let, es permitido declarar la misma variable varias veces con var. Además, puesto que las variables var tienen alcance de función en lugar de alcance de bloque, en realidad es común hacer este tipo de redeclaración. La variable i se usa con frecuencia para valores enteros y especialmente como variable de índice de bucles for. En una función con múltiples ciclos for, es típico que cada uno comience por (var **i = 0**; ...). Debido a que var no incluye estas variables en el cuerpo del ciclo, cada uno de estos ciclos vuelve a declarar (inofensivamente) y reinicializar la misma variable.

- Una de las características más inusuales de las declaraciones de var se conoce como elevación. Cuando una variable se declara con var, la declaración se “eleva” a la parte superior de la función que la encierra. La inicialización de la variable permanece donde se escribió, pero la definición de la variable se mueve a la parte superior de la función, entonces, las variables declaradas con var se pueden usar, sin error, en cualquier lugar de la función adjunta. Si el código de inicialización aún no se ha ejecutado, es posible que el valor de la variable no esté definido, pero no obtendrá un error si usa la variable antes de que se inicialice. Esto puede ser una fuente de errores y es una de las fallas importantes que corrige let: si declaras una variable con let, pero intentas usarla antes de que se ejecute la declaración let, obtendrás un error real en lugar de solo ver un valor indefinido.

Explicación

Cuando se ejecuta una función, se hace en un ambiente conocido como contexto de ejecución, por tanto, en cada nueva llamada o ejecución se crea un nuevo contexto y estos se van apilando en una pila de ejecución.



Todas las declaraciones de variables y funciones forman parte del Contexto de Ejecución Global. Considera el ejemplo de la ilustración. En el momento que se ejecuta la función primera(), se crea un contexto de ejecución por encima del contexto global. En este contexto se crea la variable a y posteriormente se ejecuta la función segunda() y el proceso se repite sucesivamente. Como JavaScript utiliza el modelo de pila en el cual el último en entrar es el primero en salir, el contexto que está hasta arriba en la pila es el que se está ejecutando en ese momento y será eliminado de la pila una vez que finalice y se procede con la ejecución del siguiente hasta terminar.

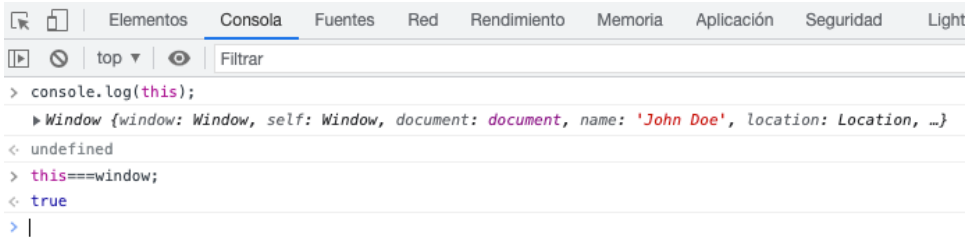
Mientras que el alcance o scope se refiere a la visibilidad de las variables y consonantes, el contexto, por otro lado, se refiere al valor de la palabra clave this.

This

Como menciona Kantor (s.f.), en JavaScript la palabra reservada this tiene un comportamiento diferente que en la mayoría de otros lenguajes de programación. El valor de this depende de cómo se invoque. Existen algunas reglas para poder entender cómo funciona, las cuales se mencionan a continuación:

- ✔ This solo

This por defecto siempre estará apuntando al objeto global, esto debido a que this está ejecutándose en el alcance global. En el navegador este objeto es *window*.



Explicación

- ✔ Enlace implícito

Cuando se llama a una función con un objeto de contexto, Ubah (2021) indica que la referencia this se vincula a dicho objeto. Según la regla de vinculación de JavaScript, una función puede utilizar un objeto como contexto si es que este objeto está vinculado a ella desde la llamada. Por ejemplo:

```
function alerta() {
console.log(this.edad + ' años');
}
const miObjeto = {
edad: 22,
alerta: alerta
}
miObjeto.alerta() // 22 años
```

Como puedes observar, la función es llamada desde **miObjeto** utilizando el punto, por tanto, this esta implícitamente vinculada al objeto. Para saber a cuál objeto está implícitamente vinculado this, es necesario observar el objeto que está a la izquierda del punto. Por ejemplo:

```
function alerta() {
console.log(this.edad + ' años');
}
const miObjeto = {
edad: 22,
alerta: alerta,
anidacionObj: {
edad: 26,
alerta: alerta
}
}
miObjeto.alerta(); // `this` vinculada a `miObjeto` -- 22 años
miObjeto.anidacionObj.alerta(); // `this` vinculada a `anidacionObj` -- 26 años
```

- ✔ Enlace explícito

Existen ocasiones en que se quiere forzar a una función a usar un objeto como contexto sin tener una referencia de función de propiedad en el objeto. Para estos casos se utilizan los métodos **call(), apply() y bind()**.

Explicación

Call, apply y bind

El método call() permite escribir un método que se puede utilizar en diferentes objetos. Por ejemplo:

```
const persona = {
nombreCompleto: function () {
return this.nombre + " " + this.apellido;
}
}

const persona1={
nombre:"John",
apellido: "Doe"
}
const persona2={
nombre:"Mary",
apellido: "Doe"
}
persona.nombreCompleto.call(persona1);  // Esto regresará "John Doe"
```

El método apply es similar al método call, la diferencia es que call toma argumentos separados y apply toma los argumentos de un arreglo.

```
const persona = {
nombreCompleto: function (city, country) {
return this.nombre + " " + this.apellido + ", " + city +", " + country;
}
}

const persona1={
nombre:"John",
apellido: "Doe"
}
persona.nombreCompleto.apply(persona1,["Oslo", "Noruega"]); // Esto regresará
"John Doe, Oslo, Noruega"
```

Explicación

El método *bind* toma prestado un método de otro objeto. El siguiente ejemplo creará dos objetos (persona y miembro):

```
const persona = {
nombre:"John",
apellido: "Doe",
nombreCompleto: function () {
return this.nombre + " " + this.apellido;
}
}

const miembro={
nombre:"Helen",
apellido: "Nilsen"
}
let nombreCompleto = persona.nombreCompleto.bind(miembro); // Esto regresará "Helen Nilsen"
```

Para tener un enlace explícito, solamente es necesario invocar a call() en esa función y pasar el objeto de contexto como parámetro. Puedes verlo de manera más específica en el siguiente ejemplo:

```
function alerta() {
console.log(this.edad + ' años');
}
const miObjeto = {
edad: 22
}
alerta.call(miObjeto); // 22 años
```

Explicación

Si se tuviera que pasar una función varias veces a nuevas variables, todas las invocaciones usarían el mismo contexto. A esto se le conoce como vinculación dura al objeto.

```
function alerta() {
console.log(this.edad + ' años');
}
const miObjeto = {
edad: 22
}


const bar = function(){
alerta.call(miObjeto);
}
bar(); //22 años
setTimeout(bar,1000); //Sigue siendo 22 años
alerta.call(miObjeto); // Aún 22 años
bar.call(window); //Continúan los 22 años
```

- ✔ Enlace constructor

Cuando se llama a una función utilizando la palabra reservada ***new***, se crea un objeto nuevo y la referencia this se vincula al mismo. A estas funciones se les conoce como función constructora. Observa el siguiente ejemplo:

```
function alerta(edad) {
this.edad = edad;
}
const bar = new alerta(22);
console.log(bar.edad + ' años'); // 22 años
```

Un aspecto interesante es que JavaScript determina el valor de this en tiempo de ejecución considerando el contexto actual, es por ello que en algunas ocasiones podría no estar referenciando el valor esperado.

Explicación

Considera el siguiente ejemplo, la clase **Perro** que tiene un método hacer **sonido**:

```
const Perro = function(nombre, sonido) {
this.nombre = nombre;
this.sonido = sonido;
this.hacerSonido = function() {
console.log( this.nombre + ' dice: ' + this.sonido );
};
}
const perrito = new Perro('Firulais', 'Guau');
perrito.hacerSonido(); // Firulais dice: Guau
```

Si se le agrega al **perro** un método **ladrar** para que pueda repetir el sonido 100 veces con un periodo de repetición de medio segundo, la implementación quedaría de la siguiente manera:

```
const Perro = function(nombre, sonido) {
this.nombre = nombre;
this.sonido = sonido;
this.hacerSonido = function() {
console.log( this.nombre + ' dice: ' + this.sonido );
};
this.molestar = function() {
let contar = 0, max = 100;
const t = setInterval(function() {
this.hacerSonido(); // <-- esta línea falla con `this.hacerSonido no es una función`
contar++;
if (contar === max) {
clearTimeout(t);
}
}, 500);
};
}
const perrito = new Perro('Firulais', 'Guau');
perrito.molestar();
```

Lo anterior no se ejecuta correctamente debido a que dentro del setInterval se ha creado un nuevo contexto con alcance global, por tanto, this ya no apunta a la instancia perrito, por lo que el navegador estará apuntando al objeto window, el cual no tiene definido un método hacerSonido().

Explicación

Esto se puede arreglar de dos maneras:

- La primera de ella es que antes de crear el contexto se asigne this a una variable local y posteriormente referenciarla dentro de la función.

```
const Perro = function(nombre, sonido) {
this.nombre = nombre;
this.sonido = sonido;
this.hacerSonido = function() {
console.log( this.nombre + ' dice: ' + this.sonido );
};
this.molestar = function() {
let contar = 0, max = 100;
const self =this;
const t = setInterval(function() {
self.hacerSonido();
contar++;
if (contar === max) {
clearTimeout(t);
}
}, 500);
};
}
const perrito = new Perro('Firulais', 'Guau');
perrito.molestar();
```

Explicación

- Otra manera de solucionarlo es a través de una función flecha, la cual enlaza de manera automática a this al contexto del código donde se definió.

```
const Perro = function(nombre, sonido) {
this.nombre = nombre;
this.sonido = sonido;
this.hacerSonido = function() {
console.log( this.nombre + ' dice: ' + this.sonido );
};
this.molestar = function() {
let contar = 0, max = 100;
const self =this;
const t = setInterval(() => {
this.hacerSonido();
contar++;
if (contar === max) {
clearTimeout(t);
}
}, 500);
};
}
const perrito = new Perro('Firulais', 'Guau');
perrito.molestar();
```

Cierre

En este tema aprendiste que el **scope** se refiere a la visibilidad de las variables y que este puede ser local o global. También conociste que con ECMAScript 6 se introdujo al lenguaje la posibilidad de declarar diferentes tipos de variables (let y const) y que cuando una función es llamada crea un contexto que es apilable.

Por otro lado, ahora puedes distinguir algunas características de la palabra this, entre ellas:

- This no es una variable.
- This puede cambiar su valor dependiendo de la forma en que sea invocada.
- El valor de this es definido en tiempo de ejecución.
- Cuando una función es declarada, this no tiene valor, sino hasta que la función es llamada.

Referencias bibliográficas

- Flanagan, D. (2020). *JavaScript: The Definitive Guide* (7a ed.). Estados Unidos: O'Reilly Media, Inc.
- Kantor, I. (s.f.). *El lenguaje JavaScript*. Recuperado de https://es.javascript.info/js
- MDN. (2022). *Tipos de datos y estructuras en JavaScript*. Recuperado de
- https://developer.mozilla.org/es/docs/Web/JavaScript/Data_structures#objetos

Para saber más

Lecturas

- Mercadal, F. (2021). *La guía completa sobre 'this' en JavaScript*. Recuperado de https://www.freecodecamp.org/espanol/news/la-guia-completa-sobre-this-en-javascript/
- W3Schools. (s.f.). *The JavaScript **this** Keyword*. Recuperado de https://www.w3schools.com/js/js_this.asp

Videos

- La Cocina del Código. (2020, 30 de abril). *6. EL SCOPE en JAVASCRIPT | JS en ESPAÑOL* [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=s-7C09ymzK8&t=1s
- La Cocina del Código. (2021, 3 de marzo). *15. THIS EN JAVASCRIPT (bind, call, apply y más* [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=bS71_W_BDFE

Checkpoints

Asegúrate de:

- Comprender la diferencia entre alcance global y alcance local.
- Distinguir la diferencia entre var, let y const.
- Utilizar this en sus diferentes maneras.
- Implementar this a partir de ECMAScript 6.

Requerimientos técnicos

- Computadora con acceso a Internet.
- Editor de texto.
- Permisos de administrador y/o Git instalado previamente.

Prework

- Leer detenidamente y comprender el material explicado en el tema 4.
- Practicar todos los ejemplos que se describen previamente en el tema 4.
- Revisar cada uno de los recursos adicionales que se proponen en el tema 4.