PROGRAMACIÓN AVANZADA EN JAVASCRIPT

Módulo 3

OTEC PLATAFORMA 5 CHILE

MÓDULO 3

CLASE 5

Introducción

A partir de estas clases que nos quedan del Módulo 03, vamos a trabajar con los fundamentos sobre los que se basa JavaScript. Con ellos va a ganar comprensión sobre el funcionamiento de las herramientas que el lenguaje provee.

No sólo va a trabajar con teoría, sino, también va a tener ejercicios para que haga y puedas testear.

Modificar el *this*

Cuando vimos el keyword this, dijimos que el intérprete era el que manejaba su valor. Sin embargo, esto no era del todo cierto: hay una serie de variables que permiten setear el keyword this.

Como en JavaScript las funciones son un tipo de objeto especial (vimos que tenían algunas propiedades específicas como length y name), éstas también contienen métodos propios. Así como los arreglos tienen acceso a métodos como push, pop y slice, todas las funciones tienen acceso a los métodos:

- ***bind***
- ***call***
- ***apply***

Justamente, invocando estos métodos, se puede tener control sobre el contexto de ejecución de la función. Es decir, de la variable this.

Usemos un objeto de ejemplo:

```
var persona = {
 nombre: "George",
 apellido: "Lucas",
 getNombre: function () {
   var nombreCompleto =
this.nombre + " " +
this.apellido;
   return nombreCompleto;
 },
};

var logNombre = function () {
 console.log(this.getNombre());
};
```

Bind

Utilizando el código del ejemplo anterior, va a usar el keyword this para invocar el método del objeto persona. Como verá, el objeto anterior produce un error, ya que cuando ejecuta `logNombre()`, el `this` que está adentro hace referencia al objeto global. Y ese objeto no tiene un método `getNombre`.

```
var logNombrePersona = logNombre.bind(persona);
logNombrePersona();
```

La función `bind()` devuelve una copia de la función, la cual tiene internamente asociado el keyword this al objeto que le pases por parámetro. Si la llama sobre `logNombre` y le pasa persona como argumento, va a ver que al ejecutar la nueva función `logNombrePersona()` se va a loguear correctamente el nombre de persona.

```
// no hace falta decir nada más, el this ya está grabado
logNombrePersona() // George Lucas
```

Si usamos *bind()* la nueva función queda siempre ligada al objeto que pasó como argumento. En cambio, si quisiera usarla para otro objeto, tendría que crear una nueva copia de la función y bindiarle un nuevo objeto.

Si ese es el caso, podría usar el método call().

Call

Siguiendo con el ejemplo, a diferencia de `bind`, el método `call` no retorna una nueva función: la invoca con el contexto que le pasemos por parámetro en ese mismo momento.

```
logNombre.call(persona);
```

En este caso, estamos invocando la función original `logNombre`, pero con `call` le estamos indicando a qué objeto tiene que hacer referencia `this` dentro de esa función.

El primer argumento de `call` es el objeto a usar cómo `this`. Después de este, se pueden pasar otros argumentos, que serán -a su vez- pasados a la función que estamos invocando. Por ejemplo, si nuestra función recibiera argumentos, usamos `call` de la siguiente manera:

```
var logNombre = function(arg1, arg2){
  console.log(arg1 +' '+ this.getNombre() + arg2);
}

logNombre.call(persona, 'Hola', ', Cómo estás?');
'Hola George Lucas, Cómo estás?'
```

Apply

La función `apply` es casi igual a `call`, excepto que recibe los argumentos de distinta manera. En este caso, `apply` necesita dos argumentos: el primero, es el objeto a bindear con `this` (igual que call); y el segundo parámetro es un arreglo.

En este arreglo pasamos los argumentos que va a usar la función que invocamos.

Por ejemplo, para obtener el mismo comportamiento que hicimos con `call`, pero con `apply`:

```
var logNombre = function(arg1, arg2){
  console.log(arg1 +' '+ this.getNombre() + arg2);
}

logNombre.apply(persona, ['Hola', ', Cómo estas?']);
'Hola George Lucas, Cómo estas?'
```

Un arreglo puede ser más fácil de pasar cuando no sabemos, a priori, cuántos argumentos serán (previo a ES6).

Closures

Un closure es la habilidad de una función para recordar y acceder a su lexical scope cuando es invocada fuera de este.

Mire este ejemplo:

```
function saludar(saludo) {
 return function (nombre) {
   console.log(saludo + " " +
nombre);
 };
}

var saludarHola = saludar("Hola"); //
Esto devuelve una función

saludarHola("Quentin"); // 'Hola
Quentin'
```

Closures

Veamos paso a paso lo que va a ocurrir cuando ejecutemos el código anterior:

- **Primero se creará el contexto de ejecución global: en esta etapa el intérprete guardará espacio para la declaración de la función saludar.**
- **Luego, cuando se encuentra con la invocación a la función saludar, va a crear un nuevo contexto y dentro de este la variable saludo va a tomar el valor que le pasamos por parámetro: 'Hola'.**
- **Luego de terminar de ejecutar y retornar una función (la que estamos guardando en saludarHola), ese contexto es destruido.**

¿Qué pasa, entonces, con la variable saludo?

- El intérprete saca el contexto del stack pero deja en algún lugar de la memoria las variables que se usaron dentro (hay un proceso dentro de JavaScript que se llama garbage collection que eventualmente las va limpiando si no son utilizadas). Por lo tanto, esa variable todavía va a estar en memoria (segunda parte de la imagen).

- Por último se ejecuta la función saludarHola y se pasa como parámetro el string 'Hola'. Por lo tanto, se crea un nuevo contexto de ejecución, con la variable mencionada. Ahora, cómo dentro de la función saludarHola se hace referencia a la variable saludo, el intérprete intenta buscarla en su scope. Como saludo no está definida en ese contexto, el intérprete sale a buscarla siguiendo la scope chain y a pesar que el contexto ya no existe, la referencia al ambiente exterior y a sus variables todavía persisten: a este fenómeno se lo llama closure.

Closures And Callbacks

Ahora que sabemos qué son los closures, y rememoramos todo lo que hicimos alguna vez con JavaScript, es muy probable que nos demos cuenta que ya lo veníamos usando.

Por ejemplo:

```
function saludarMasTarde() {
 var saludo = "Hola";

 setTimeout(function () {
   console.log(saludo);
 }, 3000);
}

saludarMasTarde();
```

En el ejemplo anterior, cuando invocamos a saludarMasTarde, estamos creando un execution context, en el que invocamos a la función setTimeout y donde definimos la variable saludo. Ese execution context es destruido, pero setTimeout contiene una referencia a saludo.

Lo que realmente ocurre es que cuando pasan los tres segundos (esto lo hace algún componente externo al intérprete), se lanza un evento diciendo que hay que ejecutar el callback, que es justamente una function expression. De esta manera, se crea un execution context para esa función, y dentro de ella se usa saludo. Como no está en ese contexto, entonces, el intérprete sale a buscarla afuera y la encuentra en el closure.

MÓDULO 3

GRACIAS

Aquí finaliza la clase n°5 del módulo 3

OTEC PLATAFORMA 5 CHILE