CLASE 03 | JS AVANZADO II

POR ALEJO BENGOECHEA

JAVASCRIPT AVANZADO II

▼ CLOSURES

Un ***Closure*** es una función que retorna otra función.

```
function saludar( saludo ){
  return function( nombre ){
    console.log(saludo + ' ' + nombre);
  }
}
var saludarHola = saludar('Hola'); // Esto devuelve una función
saludarHola('Toni'); // 'Hola Toni'
```

En este ejemplo podemos ver que hay una función "*padre*" y una función "*hija*". El parámetro de la función padre la definimos en una variable que ejecuta a la función padre.

Luego de terminar de ejecutar y retornar una función (la que estamos guardando en `saludarHola`), ese contexto es destruido. ¿Pero qué pasa con la variable **saludo**?. Bueno, el interprete saca el contexto del stack, pero deja en algún lugar de memoria las variables que se usaron adentro (hay un proceso dentro de JavaScript que se llama `garbage collection` que eventualmente las va limpiando si no las usamos.). Por lo tanto, esa variable todavía va a estar en la memoria.

```
var creaFuncion = function(){
  var arreglo = [];
  for ( var i = 0; i < 3; i++){
    arreglo.push(function(){console.log(i);})
  }
  return arreglo;
}

var arr = creaFuncion();
```

```
arr[0]() // 3
arr[1]() // 3
```

En este ejemplo, se crea una función **creaFuncion** en la cual se declara un *arreglo* vacío y un *bucle for*. En cada iteración del bucle se pushea una nueva función al arreglo que consologea el índice **i** de esa iteración. Finalmente retorna el arreglo. Luego se crea una variable **arr** que ejecuta la función padre.

Entonces, cuando imprimimos **arr[0]()** se ejecutará la primera función del arreglo, lo que debería mostrar el índice **0**. En este caso no lo hace porque la variable **i** en el *bucle for* fue definida con **var**.

```
var creaFuncion = function(){
  var arreglo = [];
  for ( let i=0; i < 3; i++){
    arreglo.push(function(){console.log(i);})
  }
  return arreglo;
}

var arr = creaFuncion();
arr[0]() // 0
arr[1]() // 1
```

En este caso si se imprimen los índices correspondientes porque declaramos la variable **i** con **let**.

A continuación veremos otra forma de hacer lo mismo.

```
var creaFuncion = function(){
  var arreglo = [];
  for ( var i=0; i < 3; i++){
    // IIFE
    arreglo.push((function(j){return function() {console.log(j);}}(i)))
  }
  return arreglo;
}

var arr = creaFuncion();
arr[0]() // 0
arr[1]() // 1
arr[2]() // 2
```

Si queremos que cada función guardase el valor de `i`, deberíamos crear un *execution content* donde se cree una variable nueva en cada iteración. Para eso vamos a usar una IIFE a la cuál le vamos a pasar como parámetro `i`. Como

estamos ejecutando la función, se va a a crear un contexto nuevo por cada ejecución, y por ende van a existir tres variables `j` (cada una en un contexto distinto) que contendrán los valores recibidos por parámetro.

```
function hacerSaludo( lenguaje ){
  if ( lenguaje === 'en'){
    return function(){console.log('Hi!');}
  }

  if ( lenguaje === 'es'){
    return function(){console.log('Hola!');}
  }
}

var saludoIngles = hacerSaludo('en');
var saludoEspaniol = hacerSaludo('es');

saludoIngles()
saludoEspaniol()
```

Este caso es igual al anterior, solo que se agrega un condicional dentro de la función padre. Por lo que ahora puede ejecutar, la misma función, una u otra función hija, dependiendo del parámetro.

▼ BIND, CALL & APPLY

Estos son métodos nativos de JavaScript.

▼ .BIND()

Este método sirve para direccionar la palabra clave **this**. Crea una nueva función, que cuando es llamada, asigna a su operador *this* el valor entregado, con una secuencia de argumentos dados precediendo a cualquiera entregados cuando la función es llamada.

```
var persona = {nombre: 'Guille', apellido: 'Aszyn',}

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

var logNombrePersona = logNombre.bind(persona);
// el primer parametro de bind es el this!
logNombrePersona();
// BIND DEVUELVE UNA FUNCION!
```

Si nosotros queremos ejecutar directamente la función **logNombre()** la terminal nos arrojaría *undefined*. Esto es porque esta función está declarada en el contexto global, y allí no hay ninguna variable llamada *nombre*. Por lo

tanto, creamos una nueva variable que contiene una copia de esa función y le agregamos el método **.bind()** con el nombre del objeto al que queremos hacer referencia (*persona*). Así, **this** sabrá que tiene que ir a ese objeto.

```
function multiplica(a, b){
  return a * b;
}
var multiplicaPorDos = multiplica.bind(this, 2);

console.log(multiplica(5));
// 2 * 5 ---> 10
```

En este caso, también se puede utilizar el método **.bind()** pero con la diferencia que **this** no tendrá ningún uso. Para hardcodear los parámetros de la función **multiplica**, hay que crear una nueva variable, la cual también contenga una copia de la función original. A esta se le agrega el método, y en el argumento ***siempre*** se pone primero el **this**, porque el método funciona así. Los siguientes parámetros del método son los que reemplazaran a los parámetros de la función. Es importante destacar que se reemplazarán en orden de sucesión.

▼ **.CALL()**

El método **call** es muy parecido al **bind** con la diferencia que este directamente invoca a la función y le pasa el valor de **this**.

```
var persona = {nombre: 'Guille', apellido: 'Aszyn',}
var logNombre = function(){console.log(this.nombre);}

logNombre.call(persona);
```

Aquí, el método **call** se asegura de que la palabra **this** sea reemplazada por la palabra *persona* que es el nombre del objeto al que queremos hacer referencia.

```
var persona = {nombre: 'Guille', apellido: 'Aszyn',}

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

logNombre.call(persona, 'Hola', ', Cómo estas?');
```

De la misma forma que en *bind*, podemos hardcodear parámetros cuando se trata de funciones. Por ejemplo, aquí el método **call** recibe primero (***siempre***) el nombre del objeto al que queremos que **this** refiera. Luego, podremos agregar parámetros que reemplazarán a los de la función.

▼ **.APPLY()**

Call y **Apply** son iguales. La única diferencia es en cómo reciben sus parámetros.

```
var logNombre = function(arg1, arg2){
  console.log(arg1 +' '+ this.nombre +' '+ arg2);
}
logNombre.apply(persona, ['Hola', 'Cómo estas?']);
```

El primer parámetro es el objeto al que **this** debe referirse. El segundo parámetro siempre es una arreglo, dentro del cuál pondremos los valores que queramos que la función tome como parámetros.