
ICARO
Nuestro compromiso es con el futuro.

Introducción a la programación

Clase 8

Funciones y funciones flecha

Tipos de funciones

Cuando trabajamos con **funciones** hemos visto que podemos definirlas con un nombre, pasarles algún que otro **argumento** y luego expresar dentro del bloque de código qué es lo que esa función debería hacer cuando sea llamada o ejecutada.

En esta misma línea es interesante conocer que cuando trabajamos con **funciones** existen principalmente dos formas de declararlas:

- Las funciones **declaradas**
- Las funciones **expresadas**

Funciones declaradas

Son aquellas que se **declaran** usando la estructura básica que hemos visto. Pueden recibir un nombre, escrito luego de la palabra reservada **function**, a través del cual podremos invocarla luego.

Si poseen nombre son funciones nombradas.

```
function funcionDeclarada() {
    return true
}
```

Funciones expresadas

Son aquellas que se **asignan** como valor de una variable. En este caso la función en sí no posee un nombre, entonces se la conoce como función anónima.

Para invocarla usaremos el nombre de la variable donde la guardamos.

```
let funcionExpresada = function() {
    return true
}
```

Arrow Functions

Las **funciones flecha** o **arrow functions** son una forma de simplificar el código de una función y reducir el tamaño de las mismas, sobre todo cuando el bloque de código de su interior es una única línea de código, caso en el que podemos hacer que la **función** entera se exprese en tan solo una línea.

Además de esto, es una forma más corta de expresar ese código, lo que trae muchos beneficios, sobre todo cuando trabajamos con **callbacks** (tema que veremos en breve) ya que éstos quedan mucho más legibles.

```
let funcionFlecha = () => true
```

Arrow Functions

Una arrow function siempre es una **función anónima** y como tal, si queremos utilizar esta nomenclatura para declarar una función debemos utilizar una **variable** para almacenarla, como si de una **función expresada** se tratase.

Además no utilizaremos palabras reservadas, y entre los parámetros y el bloque de código se ubicará la famosa **flecha** o **arrow**, que no es más que un caracter **= (igual)** más uno de **> (mayor que)**: **=>**

Arrow Functions

Otra cuestión fundamental en las **arrow functions** es el **return implícito**. Esto es cuando tenemos una sola línea de código que es un **return** de **algún valor**, podemos simplificar esto colocando después de la flecha lo que se debería retornar sin nada más. Esto simplifica muchísimo la escritura y lectura de las funciones, aunque tal vez en un principio parezca más complejo de leer.

```
function returnResultado(a, b) {
    return a + b
}
```

```
let returnResultado = (a,b) => a + b
```

Arrow Functions

Si queremos utilizar múltiples líneas de código dentro de una arrow function, utilizaremos las llaves como lo hacemos con las funciones tradicionales, con la diferencia de que si existe más de una línea de código aquí, ya no podremos utilizar el return implícito.

```
let returnResultado = (a,b) => {
    let resultado = a + b
    return resultado
}
```

Arrow Functions

Uno de los casos más particulares es cuando queremos hacer un **return implícito** de un **Objeto Literal**, ya que si colocamos el **Objeto Literal** con sus llaves directamente a continuación de la flecha, **JavaScript** interpretará que estamos queriendo crear un bloque de código y no un **Objeto Literal**, para ello podemos utilizar unos paréntesis para envolver este objeto y utilizar el return implícito, o bien retornar el **Objeto Literal** dentro de un bloque de código como haríamos normalmente.

```
let returnObjetoLiteral = (nombre) => ({nombre: nombre})
```

```
let returnObjetoLiteral = (nombre) => { return {nombre: nombre}}
```

Callbacks

Callbacks

Como sabemos, en **JavaScript**, las **funciones** son un **tipo de dato** más y podemos guardarlas dentro de variables o recibirlas como argumento de una función, al igual que pasa con números, arrays o cualquier otro **tipo de dato**.

Entonces surge el concepto de **Callback**, que no es más que una función que es pasada como parámetro de otra función. Pero no su resultado, sino la función en sí, para que la función que la recibe decida cuándo ejecutar ese código.

```
let funcionPrincipal = (callback) => {
    console.log('Antes de llamar el callback')
    callback()
    console.log('Despues de llamar el callback')
}
```

Callbacks

Si tenemos

```
let funcionCallback = () => 'Hola!'
```

Sabemos que **funcionCallback** es una función cuyo único propósito es retornar un saludo. Podemos pasarle esta función a la **funcionPrincipal**, para que esta última la ejecute cuando corresponda

```
let funcionPrincipal = (callback) => {
    console.log('Antes de llamar el callback')
    console.log(callback())
    console.log('Despues de llamar el callback')
}

funcionPrincipal(funcionCallback)
```

```
$ node index
Antes de llamar el callback
Hola!
Despues de llamar el callback
```

Callbacks

Esto realmente comienza a ser útil cuando tenemos **funciones** que reciben **parámetros** y **callbacks**, pudiendo pasarle estos **parámetros** a los **callbacks**.

```
let saludar = (nombre) => `Hola ${nombre}`

let saludarRepetidamente = (callback, arrayNombres) => {
    for (let i = 0; i < arrayNombres.length; i++) {
        console.log(callback(arrayNombres[i]))
    }
}

let nombres = ['Juan', 'Roberto', 'Andres']

saludarRepetidamente(saludar, nombres)
```

```
$ node index
Hola Juan
Hola Roberto
Hola Andres
```

Métodos de Arrays II

Métodos de Arrays II

Hasta aquí vimos bastantes métodos de **Arrays**, **strings**, etc.
También vimos como recorrer estos **Arrays** utilizando un **for**.
Este método for ha caído en desuso desde que en **JavaScript** se incorporaron nuevos métodos para recorrer **Arrays** y trabajar con los datos.
Estos métodos nuevos se conocen como **High Order Functions** o **Funciones de Orden Superior**.

Son estos 4:

- **forEach**
- **map**
- **filter**
- **reduce**

Métodos de Arrays II

Antes de comenzar a ver cada uno de los métodos debemos saber que todos ellos nos van a permitir trabajar con el **contenido** de un **Array**, recorriéndolo e iterando por cada una de las posiciones de su contenido.

Cuando utilicemos estos métodos, éstos recibirán como **parámetro** un **callback**, es decir una función, que nos permitirá realizar una acción por cada uno de los elementos que contenga ese **Array**.

forEach

El primero de los métodos que veremos es el **forEach**.
Este recibirá un callback que se ejecutará por cada uno de los elementos del array. Este **callback**, a su vez, recibirá al menos un **parámetro**, que representará literalmente el elemento que es contenido por ese **Array** que está siendo iterado. También disponemos de un segundo **parámetro** opcional que funciona como **índice**, es decir, la **posición** del **Array** en la que nos encontramos.

```
let nombres = ['Virginia', 'Josefina', 'Juan', 'Francisco']

let funcionCallback = (elemento, indice) => {
    console.log(`El elemento en la posición ${indice} es ${elemento}`)
}

nombres.forEach(funcionCallback)
```

```
$ node index
El elemento en la posición 0 es Virginia
El elemento en la posición 1 es Josefina
El elemento en la posición 2 es Juan
El elemento en la posición 3 es Francisco
```

forEach

Por supuesto que este **callback** tranquilamente podemos pasarlo dentro del **forEach** y simplificar el código, además de que queda mucho más legible.
Por ejemplo, en lugar de utilizar un bucle for para recorrer un **Array** e imprimir algún de cada elemento, podemos utilizar un **forEach** para lo mismo, escribiendo mucho menos código.

```
let personas = [
    {nombre: 'Tahiel'},
    {nombre: 'Marcos'},
    {nombre: 'Javier'}
]
```

```
for (let i = 0; i < personas.length; i++) {
    console.log(personas[i].nombre)
}
```

```
personas.forEach(persona => console.log(persona.nombre))
```

```
$ node index
Tahiel
Marcos
Javier
```

forEach

Si analizamos el **forEach** encontraremos que posee una gran diferencia con el resto de las **High Order Functions**, ya que en el **forEach** no nos importa el valor del retorno, es decir, si observamos qué nos retorna el **forEach** veremos que su valor es **undefined**.
Cuando veamos los demás, como por ejemplo el map, veremos que el valor de retorno es clave para el funcionamiento del método en sí. En el caso del **forEach**, lo importante es lo que coloquemos dentro del bloque de código a ejecutar y no el retorno del método en sí.

map

El método **map** es muy parecido al **forEach** en cuanto a sus parámetros (son exactamente los mismos) pero como vimos recién difieren en el retorno. El **map** junto con el **reduce** y el **filter** retornan algo, y es esto que retornan lo que nos resulta de utilidad. El **map**, por ejemplo, retorna un **Array** con las condiciones que le pasemos dentro del **callback**. Por ejemplo:

```
let array1 = ['nombre1','nombre2','nombre3']
let array2 = array1.map((element, index) =>
  `${element} ${index}`)
```

```
console.log(array1)
console.log(array2)
```

```
$ node index
[ 'nombre1', 'nombre2', 'nombre3' ]
[ 'nombre1 0', 'nombre2 1', 'nombre3 2' ]
```

map

Entonces, vimos que en el **map** lo que nos importa es el valor del retorno, que es un **Array** nuevo. Por ello, siempre debemos guardar lo que nos retorna el **map** en una variable para almacenar este nuevo **Array** con los cambios que indicamos en el método.

Si no guardamos lo que nos retorna en una nueva variable, básicamente no estaríamos obteniendo nada del **map**, ya que el código que se ejecuta no estaría teniendo efecto en ningún lado y se perdería.

filter

El filter funciona muy parecido al map, en que retorna algo, solo que es mucho más sencillo, ya que nos devolverá un Array con los elementos del Array original que cumplan con la condición que le pasemos en el callback.
Por ejemplo:

```
let letras = ['asd','qwe','zxc','zxe','ase']
let filteredLetras = letras.filter(x => x.includes('e'))
console.log(filteredLetras)
```

```
$ node index
[ 'qwe', 'zxe', 'ase' ]
```

filter

Es importante destacar que a diferencia del map, el filter retorna un Array que puede estar vacío si ningún elemento cumple con la condición, o bien ser exactamente igual al Array original si todos la cumplen.

No obstante, es un método muy útil a la hora de simplificar Arrays según una condición dada.

reduce

Este método recorre el **Array** y devuelve un único valor.
Este valor depende del **callback** que le pasamos. El **callback** recibe dos parámetros: un acumulador y el elemento actual que se esté recorriendo.
Al ir recorriendo el **Array**, el acumulador va a ir acumulando los resultados según el código que le indiquemos en el bloque del **callback**.
Por ejemplo:

```
let nums = [3,5,10,20]
let suma = nums.reduce((acum, actual) => acum + actual)
console.log(suma)
```

```
$ node index
38
```

Muchas gracias!

ICARO Asociación Civil
CUIT 30716564815
info@icaro.org.ar
www.icaro.org.ar