UD4
Arrays: Map, Filter y Reduce

Estos 3 métodos se usan muchísimo para facilitarnos el manejo de datos contenidos en arrays. En cada uno de los ejemplos se tratará de dar una versión más conservadora usando funciones, y por otro lado un ejemplo haciendo uso del **operador arrow (=>),** el cual es ampliamente utilizado en la nueva especificación **JavaScript ES6**.

Índice

Arrow ? => ?

En un nivel muy básico, el operador arrow (=>) nos ayuda a simplificar cierta codificación. Veremos en el siguiente cuadro algunas comparaciones respecto a un código tradicional:

Funciones Tradicionales	Operador arrow =>
`nombre = function (argumentos) { código; }`	`nombre = ( argumentos ) => { código; }`
	- Los paréntesis y las llaves son opcionales: `argumentos => código;`

`function doble ( x )`
`{`
`  return  x *  2 ;`
`}`
	¿Y si hubiera un return dentro de la función?
	¡¡¡ También es opcional !!!
	`x  =>  x * 2;`

`let interval = setInterval ( function(){ alert("Hello"); }, 3000);`	`let interval = setInterval ( () => alert("Hello"); , 3000);`

Función Map

Map() es una función que ***crea un nuevo Array con los resultados de la llamada a la función indicada como parámetro, aplicada a cada uno de los elementos del Array sobre el que itera.***

Podemos pensar en map() como en un bucle forEach que sirve específicamente para transformar los valores de los elementos sobre los que itera. Recibe como parámetro una función, la cual a su vez recibe tres parámetros: el elemento actual del Array, el índice del elemento actual y el propio Array sobre el que se llama la función map().

```
[1, 2, 3, 4].map( function (currentValue, index, array) { ... })
```

- currentValue - El valor actual en cada iteración.

- Index - El índice de la posición del array que se está iterando.

- Array - Una copia del array completo.

Vamos a verlo con un ejemplo: Tenemos un Array de números y necesitamos convertir esos números a sus cuadrados. Primero veremos como lo haríamos por el método tradicional, iterando el Array con un bucle for y transformando los valores en cada iteración para “endosárselos” a un nuevo Array.

```
let numbers = [1, 2, 3, 4, 5, 6]
let numSqrt = []

for (let i = 0; i < numbers.length; i++) {
  numSqrt[i] = numbers[i] * numbers[i];
}
// Resultado: [1, 4, 9, 16, 25, 36]
```

Hasta aquí todo funciona perfectamente pero el código no resulta muy amigable, además es demasiado complicado para una tarea tan simple. Con el método **map()** podemos simplificar nuestro código, hacerlo mucho más comprensible y agradable y por supuesto, mucho más funcional.

```
let numbers = [1, 2, 3, 4, 5, 6];
let numSqrt = numbers.map( function (number) {
      return number * number;
})
// Resultado: [1, 4, 9, 16, 25, 36]
```

Pero aún lo podemos hacer un poco más bonito con las funciones de flecha o *Arrow Functions* disponibles en la nueva sintaxis de **ECMAScript 6**.

```
let numbers = [1, 2, 3, 4, 5, 6]
let numSqrt = numbers.map( number => number * number );
// Resultado: [1, 4, 9, 16, 25, 36]
```

Con dos simples líneas de código hemos conseguido nuestro propósito, además se ha convertido en un código mucho más legible y funcional.

RESUMIENDO MAP: Es una función que toma un array deobjetos y devuelve una array con el ***mismo número de elementos*** pero con algún tipo de transformación.

Map: Te das cuen ???

- Usando map no tenemos que controlar el estado de nuestro loop (controlar el for).
- No necesitamos hacer uso de ningún índice para acceder a nuestros elementos.
- No necesitamos crear un nuevo array contenedor donde vamos almacenando los resultados, solo una variable donde recoger el resultado final.

Función Filter

filter() al igual que **map()**, es como un **forEach** pero en este caso sirve específicamente para filtrar. Es decir, como su propio nombre indica, filtrará los elementos de un array y creará un nuevo array con los items que

pasen el filtro. Este filtro será el resultado de la función que pasemos como parámetro, que siempre devolverá **true** o **false**.

```
[1, 2, 3, 4].filter( function (currentValue, index, array) { ... })
```

- **CurrentValue** - El valor actual en cada iteración.

- **Index** - El índice de la posición del array que se está iterando.

- **Array** - Una copia del array completo.

Ahora crearemos un nuevo Array con los items cuyo valor sea mayor que 3 haciendo uso de un bucle "for" como toda la vida:

```
let numbers = [1, 2, 3, 4, 5, 6];
let numFiltered = [];

for (let i = 0; i < numbers.length; i++) {
  if(numbers[i] > 3) {
    numFiltered[i] = numbers[i];
  }
}
// Resultado: [4, 5, 6]
```

Veamos como hacerlo con el método **filter()**:

```
let numbers = [1, 2, 3, 4, 5, 6];
let numFiltered = numbers.filter(function (number) {
  return number > 3 ;
})
// Resultado: [4, 5, 6]
```

Y con un poco de ***Arrow Magic***:

```
let numbers = [1, 2, 3, 4, 5, 6] ;
let numFiltered = numbers.filter(number => number > 3) ;
// REsultado: [4, 5, 6]
```

En este caso podemos ver que además del bucle **for** nos hemos ahorrado la **condición**.

RESUMIENDO FILTER: Es una función que toma un array de un objetos y devuelve una array con el ***menor o igual número de elementos*** del array original.

Función Reduce

Reduce() convierte o reduce una matriz o Array hasta un único valor por medio de la función pasada como *callback*.

```
[1, 2, 3, 4].reduce( function (acumulador, valorActual, índice, array) { ... } ,
valorInicialAcum )
```

- **Acumulador** → Se encarga de acumular valores.

- **ValorActual** - El valor actual en cada iteración.

- **Índice** - El índice de la posición del array que se está iterando.

- **Array** - Una copia del array completo.

- **ValorInicialAcumulador** → Es el valor inicial del acumulador.

Por ejemplo, vamos a realizar la suma de todos los valores de un array:

```
let numbers = [1, 2, 3, 4, 5, 6];
let total = numbers.reduce((acumulador, actual) => acumulador + actual, 0);
// Resultado: 21
```

RESUMIENDO REDUCE: Es una función que toma un array de un objetos y devuelve ***un solo valor como resultado***.

Encadenando Funciones

Vamos a realizar un ejemplo en el que calculemos la suma de los salarios de todos los trabajadores "móviles" del siguiente array:

```
var developers = [
  { name: 'Tano', type: 'mobile', salary: 4000 },
  { name: 'Inma', type: 'mobile', salary: 31000 },
  { name: 'Edgar', type: 'web', salary: 35000 },
  { name: 'Fernando', type: 'mobile', salary: 33000 }
]
```

El código sería el siguiente:

```
let sumSalariesMobileDev = developers
  .filter(developi => developi.type === 'mobile')
  .map(developo => developo.salary)
  .reduce((acumu, actual) => acu + actual, 0)
// 68.000
```

Me gustaría terminar comentando que estos métodos tienen un aire más "declarativo" que "imperativo", es decir, se obvia el entrar en detalles sobre cómo iterar y se tiende más a esperar un determinado resultado sin decir expresamente cómo obtenerlo, cierto?.

También comentar que si te acostumbras a usarlos, rara vez volverás a usar un "for" en JS .

Enlaces

Si quisieras aprender y profundizar más en la programación funcional son muy recomendables estos 3 enlaces:

- Jafar Husain magnífico conjunto de ejercicios sobre la programación funcional en JavaScript, con una sólida introducción a Rx.js.
- Instructor Jason Rhodes de Envato Tuts+ "Curso de Programación Funcional en JavaScript"
- La guía mayormente adecuada a la programación funcional, que profundiza en por qué evitamos la mutación y el pensamiento funcional en general. Esta trata puramente sobre el modelo de programación funcional en sí.