Desestructuración de objetos y arreglos en JavaScript

La asignación de desestructuración es una poderosa característica que vino con ES6 (ECMAScript6). La desestructuración es una expresión de JavaScript que hace posible extraer valores de los arreglos, o propiedades de objetos, en otras variables. Es decir, podemos obtener datos de arreglos y objetos, y asignarlos a variables.

¿Por qué esto es necesario?

Imagina que queremos extraer la información de un arreglo. Anteriormente, ¿cómo se haría esto?

```
let introduccion = ["Hola", "yo" , "soy", "Sara"];
let saludo = introduccion[0];
let nombre = introduccion[3];

console.log(saludo);//"Hola"
console.log(nombre);//"Sara"
```

Podemos ver que cuando queremos obtener los datos de un arreglo, tenemos que hacer lo mismo una y otra vez.

La asignación de desestructuración hace que esto sea más fácil. ¿Cómo es esto?

Primero, discutamos la asignación de desestructuración con arreglos. Luego pasaremos a la desestructuración de objetos.

Desestructuración básica de un arreglo

Si queremos extraer datos de arreglos, es bastante simple usando la asignación de desestructuración.

Veamos nuestro primer ejemplo para arreglos. En lugar de repetir lo mismo, podemos hacer esto:

```
let introduccion = ["Hola", "yo" , "soy", "Sara"];
let [saludo, pronombre] = introduccion;

console.log(saludo);//"Hola"
console.log(pronombre);//"yo"
```

También podemos hacer esto con el arreglo.

```
let [saludo, pronombre] = ["Hola", "yo" , "soy", "Sara"];

console.log(saludo);//"Hola"
console.log(pronombre);//"yo"
```

Declarando variables antes de la asignación

Las variables pueden ser declaradas antes de su asignación, algo como esto:

```
let saludo, pronombre;
[saludo, pronombre] = ["Hola", "yo" , "soy", "Sara"];

console.log(saludo);//"Hola"
console.log(pronombre);//"yo"
```

Nota que las variables son establecidas de izquierda a derecha. De esta manera la primera variable obtiene el valor del primer elemento del arreglo, la segunda variable obtiene el valor del segundo elemento del arreglo, y así sucesivamente.

Excluyendo elementos en un arreglo

¿Qué pasa si queremos obtener el primer y el último elemento de nuestro arreglo, en lugar de obtener solo el primer y el segundo elemento, y queremos asignar estos valores solo a dos variables? Esto también se puede hacer.

Observa el siguiente ejemplo:

```
let [saludo,,,nombre] = ["Hola", "yo" ,"soy", "Sara"];

console.log(saludo);//"Hola"
console.log(nombre);//"Sara"
```

¿Qué ha sucedido?

Observa la asignación en la parte izquierda del arreglo. Nota que, en lugar de usar una coma, usamos tres. La coma es usada para omitir valores en un arreglo. Entonces, si desea omitir un elemento en un arreglo, use una coma.

Hagamos otro ejercicio. Omitamos el primer y tercer elemento en la lista. ¿Cómo haríamos esto?

```
let [,pronombre,,nombre] = ["Hola", "yo" , "soy", "Sara"];

console.log(pronombre);//"yo"
console.log(nombre);//"Sara"
```

De esta manera, la coma hace su magia. Así que si queremos excluir todos los elementos, haríamos esto.

```
let [,,,,] = ["Hola", "yo" , "soy", "Sara"];
```

Asignación del resto de un arreglo

¿Qué pasa si queremos asignar algunos valores de un arreglo a una variable y el resto a otra? En este caso, podríamos hacer esto:

```
let [saludo,...introduccion] = ["Hola", "yo" , "soy", "Sara"];

console.log(saludo);//"Hola"
console.log(introduccion);//["yo", "soy", "Sara"]
```

Usando este patrón, puedes obtener y asignar la parte del arreglo faltante a una variable.

Asignación de desestructuración con funciones

Podemos extraer los datos de un arreglo devuelto por una función. Digamos que tenemos una función que retorna un arreglo, como en el ejemplo:

```
function obtenerArreglo() {
    return ["Hola", "yo" , "soy", "Sara"];
}
let [saludo,pronombre] = obtenerArreglo();

console.log(saludo);//"Hola"
console.log(pronombre);//"yo"
```

Obtenemos los mismos resultados.

Usando valores por defecto

Se pueden asignar valores por defecto a las variables en caso de que el valor que extraemos de un arreglo sea undefined:

```
let [saludo = "Holi",nombre = "Sara"] = ["Hola"];

console.log(saludo);//"Hola"
console.log(nombre);//"Sara"
```

De esta manera `nombre` es "Sara", porque no está definido en el arreglo.

Intercambio de valores usando la asignación de desestructuración

Una cosa más. Podemos usar la asignación de desestructuración para intercambiar valores de las variables.

```
let a = 3;
let b = 6;

[a,b] = [b,a];

console.log(a);//6
console.log(b);//3
```

A continuación, pasemos a la desestructuración de objetos.

Desestructuración de objetos

Primero, miremos porque es necesaria la desestructuración de objetos.

Digamos que queremos extraer datos de un objeto, y asignarlos a nuevas variables. Antes de ES6, ¿cómo se haría esto?

```
let persona = {nombre: "Sara", pais: "Nigeria", trabajo: "Desarrolladora"};

let nombre = persona.nombre;
let pais = persona.pais;
let trabajo = persona.trabajo;

console.log(nombre);//"Sara"
console.log(pais);//"Nigeria"
console.log(trabajo);//Desarrolladora"
```

Mira lo tedioso que es hacer esto. Estamos repitiendo lo mismo. La desestructuración de ES6 realmente salva el día. Veamos cómo es esto.

Desestructuración básica de objetos

Repitamos el ejemplo anterior con ES6. En lugar de asignar los valores uno por uno podemos usar el objeto de la izquierda para extraer los datos:

```
let persona = {nombre: "Sara", pais: "Nigeria", trabajo: "Desarrolladora"};

let {nombre, pais, trabajo} = persona;

console.log(nombre);//"Sara"
console.log(pais);//"Nigeria"
console.log(trabajo);//Desarrolladora"
```

Obtendrás los mismos resultados. También es válido asignar variables a un objeto que no ha sido declarado:

```
let {nombre, pais, trabajo} = {nombre: "Sarah", pais: "Nigeria", trabajo: "Desarrollador

console.log(nombre);//"Sara"
console.log(pais);//"Nigeria"
console.log(trabajo);//Desarrolladora"
```

Variables declaradas antes de ser asignadas

Variables en objetos pueden ser declarados antes de ser asignadas con desestructuración. Intentemos esto:

```
let persona = {nombre: "Sara", pais: "Nigeria", trabajo: "Desarrolladora"};
let nombre, pais, trabajo;

{nombre, pais, trabajo} = persona;

console.log(nombre);// Error : "Unexpected token ="
```

Espera, ¡¿Qué acaba de pasar?!. Oh, olvidamos agregar `()` antes de las llaves.

Los paréntesis alrededor de la declaración de asignación es una sintaxis requerida cuando usamos la asignación de desestructuración sin una declaración. Esto es así porque los `{}` en el lado izquierdo se considera un bloque y no un objeto literal. Así que aquí está como hacer esto de la marea correcta:

```
let persona = {nombre: "Sara", pais: "Nigeria", trabajo: "Desarrolladora"};
let nombre, pais, job;

({nombre, pais, trabajo} = persona);

console.log(nombre);//"Sara"
console.log(trabajo);//"Desarrolladora"
```

También es importante tener en cuenta que al usar esta sintaxis, los `()` deberían estar precedidos por una coma. De lo contrario, podría usarse para ejecutar una función de la línea anterior.

Tenga en cuenta que las variables en el objeto del lado izquierdo deben tener el mismo nombre que las propiedades del objeto `persona`. Si los nombres son diferentes, obtendremos un `undefined`:

```
let persona = {nombre: "Sara", pais: "Nigeria", trabajo: "Developer"};

let {nombre, amigos, trabajo} = persona;

console.log(nombre);//"Sara"
console.log(amigos);//undefined
```

Pero si queremos usar un nuevo nombre de variable, bueno, podemos hacerlo.

Usando un nuevo nombre de variable

Si queremos asignar valores de un objeto a una nueva variable en lugar de usar el nombre de su propiedad, podemos hacer esto:

```
let persona = {nombre: "Sara", pais: "Nigeria", trabajo: "Desarrolladora"};

let {nombre: otroNombre, trabajo: laburo} = persona;

console.log(otroNombre);//"Sara"
console.log(laburo);//"Desarrolladora"
```

Entonces los valores extraídos son pasados a las nuevas variables `otroNombre` y `laburo`.

Usando valores por defecto

Valores por defecto también pueden ser usados en la desestructuración de objetos, en caso de que una variable sea `undefined` en un objeto del que desea extraer datos.

```
let persona = {nombre: "Sara", pais: "Nigeria", trabajo: "Developer"};

let {nombre = "myNombre", amiga = "Annie"} = persona;

console.log(nombre);//"Sara"
console.log(amiga);//"Annie"
```

Entonces, si el valor no es `undefined`, la variable guarda el valor extraído del objeto como en el caso de `nombre`. De lo contrario, es usado el valor predeterminado como en el caso de `amiga`.

También podemos establecer valores por defecto cuando asignamos valores a una nueva variable:

```
let persona = {nombre: "Sara", pais: "Nigeria", trabajo: "Desarrolladora"};

let {nombre:otroNombre = "myName", amiga: compañera = "Annie"} = persona;

console.log(otroNombre);//"Sara"
console.log(compañera);//"Annie"
```

Entonces `nombre` fue extraído de persona y asignado a una variable diferente. La variable `amiga`, por otra parte, fue `undefined` en `persona`, por lo que la variable `compañera` tiene asignado el valor por defecto.

Nombre de propiedad calculado

El nombre de propiedad calculado es otra característica de los objetos que también funciona para la desestructuración. Puedes especificar el nombre de una propiedad a través de una expresión, si lo pone entre corchetes.

```
let propiedad = "nombre";

let {[propiedad] : otroNombre} = {nombre: "Sara", pais: "Nigeria", trabajo: "Desarrolla

console.log(otroNombre);//"Sara"
```

Combinando arreglos con objetos

Arreglos también pueden ser usados con objetos en la desestructuración de objetos.

```
let persona = {nombre: "Sara", pais: "Nigeria", amigas: ["Annie", "Becky"]};

let {nombre:otroNombre, amigas: compañeras} = persona;

console.log(otroNombre);//"Sara"
console.log(compañeras);//["Annie", "Becky"]
```

Anidamiento en desestructuración de Objetos

Objetos también pueden ser anidados al desestructurar:

```
let persona = {
    nombre: "Sara",
    lugar: {
        pais: "Nigeria",
        ciudad: "Lagos" },
    amigas : ["Annie", "Becky"]
};

let {nombre:otroNombre,
     lugar: {
         pais : region,
         ciudad : x}
    } = persona;

console.log(otroNombre);//"Sarah"
console.log(region);//"Nigeria"
```

Resto en la desestructuración de objetos

La sintaxis del resto también se puede usar para obtener los valores restantes. Esas claves y sus valores se copian en un nuevo objeto:

```
let persona = {nombre: "Sara", pais: "Nigeria", trabajo: "Desarrolladora" amigas: ["Ann

let {nombre, amigas, ...otros} = persona;

console.log(nombre);//"Sara"
console.log(amigas);//["Annie", "Becky"]
console.log(otros);// {pais: "Nigeria", trabajo: "Desarrolladora"}
```

Aquí, las propiedades restantes, que no han sido extraídas, se asignan a la variable `otros`. La sintaxis del resto es `...otros`. La variable `...otros` se puede renombrar a cualquier variable que desee.

Una última cosa: veamos como se puede usar la desestructuración de objetos en funciones.

Desestructuración de objetos en funciones

La desestructuración de objetos se puede utilizar para asignar parámetros a las funciones:

```
function persona({nombre: x, trabajo: y} = {}) {
    console.log(x);
}

persona({nombre: "Michelle"});//"Michelle"
persona();//undefined
persona(amiga);//Error : amiga is not defined
```

Observa los `{}` en el lado derecho del objeto de parámetros. Nos permite llamar a la función sin pasar ningún argumento. Es por eso por lo que no obtenemos un `undefined`. Si removemos esto, obtendremos un mensaje de error.

También podemos asignar valores por defecto a los parámetros.

```
function persona({nombre: x = "Sara", trabajo: y = "Desarrolladora"} = {}) {
    console.log(x);
}

persona({nombre});//"Sara"
```

Podemos hacer unas muchas cosas con la desestructuración de Objetos y Arreglos como hemos visto en los ejemplos anteriores.

Operador de propagación

En JavaScript, los operadores de propagación (spread operators) son una característica poderosa introducida en ECMAScript 2015 (ES6) que permite expandir elementos de un iterable (como un array, string, o incluso un objeto) en lugares donde se esperan múltiples argumentos o elementos. Los operadores de propagación son:

1. Propagación en arrays:

El operador de propagación (...) permite expandir un array en elementos individuales. Esto es útil para copiar arrays, concatenar arrays, o pasar elementos de un array como argumentos a una función.

```
// Ejemplo de propagación en arrays
const arr1 = [1, 2, 3];
const arr2 = [...arr1, 4, 5, 6];
console.log(arr2); // [1, 2, 3, 4, 5, 6]

// Pasar elementos de un array como argumentos
const sum = (a, b, c) => a + b + c;
const numbers = [1, 2, 3];
console.log(sum(...numbers)); // 6
```

2. Propagación en objetos:

El operador de propagación también se puede usar con objetos para crear copias superficiales o combinar varios objetos. Esta característica se introdujo en ECMAScript 2018 (ES9).

```
// Ejemplo de propagación en objetos
const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };
const combinedObj = { ...obj1, ...obj2 };
console.log(combinedObj); // { a: 1, b: 3, c: 4 }
```

3. Rest operator (operador de descanso):

Aunque no es estrictamente un operador de propagación, el operador de descanso (rest operator), que también utiliza los tres puntos (`...`), permite agrupar múltiples elementos en un solo array. Se usa principalmente en la declaración de funciones para capturar todos los argumentos restantes en un array.

```
// Ejemplo de operador de descanso en funciones
const sumAll = (...args) => {
  return args.reduce((acc, current) => acc + current, 0);
};
console.log(sumAll(1, 2, 3, 4)); // 10
```

Estos operadores hacen que el manejo de estructuras de datos como arrays y objetos sea más conciso y legible en JavaScript.

¿Qué es JSON y su utilidad?

JSON es un formato de texto que forma parte del sistema de JavaScript y que se deriva de su sintaxis, pero no tiene como objetivo la creación de programas, sino el acceso, almacenamiento e intercambio de datos. Usualmente es conocido como una alternativa al lenguaje XML.

La sintaxis de JSON funciona de modo similar a JavaScript. Por ejemplo:

El arreglo de información se hace mediante claves (keys).

La información se separa por comas.

Las llaves agrupan objetos.

Los corchetes agrupan arreglos de datos.

Sin embargo, JSON se distingue de JavaScript porque sus claves tienen que ser strings (o secuencias de caracteres), mientras que sus valores deben ser strings, números, objetos, arreglos, boleados o null.

¿Para qué sirve un archivo JSON?

Un archivo JSON es un documento digital creado en este lenguaje que almacena información organizada, con el fin de hacer más simple su búsqueda y acceso. La ventaja de este formato es que permite obtener código legible para las personas con nombres y valores que funcionan como indicadores de la información que contienen.

¿Dónde se utiliza JSON?

Como hemos visto, los archivos JSON permiten obtener un código legible de almacenamiento y también son útiles para manipular la información de un programa a la hora de crear un software.

Originalmente, JSON era utilizado únicamente bajo la notación basada en objetos de JavaScript. Actualmente, debido a su popularidad, muchos lenguajes de programación son compatibles con el formato JSON.

Los archivos en este formato suelen tener la terminación .json y son especialmente provechosos para intercambiar o transferir información a lo largo de diferentes tipos de dispositivos digitales.

El caso más común de uso de JSON está en el diseño de sitios web. Al crear páginas en línea queremos asegurarnos de que el sitio lea correctamente la información contenida en el servidor y que la muestre de forma óptima. Además, es deseable que el programador pueda modificar el código durante la marcha para corregir errores.

Pero esta no es su única función. Por ejemplo, podemos emplear JSON para la creación de aplicaciones móviles y programas computacionales o incluso para la transferencia de documentos. Esta herramienta es tan versátil que podríamos asegurar que está prácticamente en todos lados.

Funciones anónimas en JavaScript

Las funciones anónimas en JavaScript son aquellas que no han sido declaradas con un nombre. En este lenguaje de programación, podemos declarar este tipo de elemento usando cualquiera de los modos de escribir funciones.

Es decir, podemos declarar funciones anónimas en JavaScript con arrow function, un modo de escritura que siempre es anónimo:

```
( ) => { }
```

O con la palabra clave function:

```
function ( ) { }
```

Como mencionamos antes, estas dos maneras de escribir funciones anónimas en JavaScript implican dos aproximaciones diferentes al entendimiento de this.

En la función de javascript anónima de tipo arrow function, la palabra clave this toma el valor del scope superior. Por su parte, en la función anónima de tipo function, el valor pertenece a quien ejecuta esta función. Por esto es muy importante reconocer las distintas maneras de escribir funciones anónimas en JavaScript, pues tu código puede tener resultados muy diferentes según como determines la función.

Ten en cuenta que en nuestro ejemplo anterior no hemos visto una diferencia en el resultado porque nuestra función drawTime no usa la palabra clave this y estamos atacando directamente al DOM.

Funciones no anónimas

También cabe resaltar que el efecto de escribir una función cambia incluso en las funciones declaradas por nombre. Supongamos que tenemos las siguientes dos maneras de ejecutar la función drawTime que hemos declarado antes, pero con algún valor en el parámetro:

button.addEventListener (‘click’, drawTime)

drawTime ( );

Las dos formas anteriores de ejecutar la función son muy distintas, porque en el primer caso la función drawTime se asigna como consecuencia o respuesta a un evento que sucede. Por ello, se ejecutará desde un contexto distinto al ejecutado en la segunda línea de código. En este sentido, el valor entre paréntesis de la función drawTime sería el evento:

function drawTime (event) {

document.getElementById (‘demo’).innerHTML = new Date

}

Ten presente que, aunque en las líneas anteriores hemos nombrado al parámetro automático como event, puedes llamarlo como quieras y representará el evento recibido igualmente. Sea cual sea el nombre que le pongas, este objeto nos dará información sobre el evento recibido en la función, como lo son el path y el target. Si quieres conocer más sobre este objeto, puedes insertar el comando console.log (event) dentro de la función drawTime. También puedes leer nuestro post sobre event.target en JavaScript.

Filter - map - every - reduce - some – sort

.map y .forEach se pueden llegar a confundir, ambos se ejecutarán la misma cantidad de ocasiones que la cantidad de tus elementos en tus arreglos, la diferencia principal es: el .map crea un nuevo arreglo, mientras que el .forEach no.

Es considerado un anti-pattern o mala práctica que si no utilizarás el arreglo restante debes evitar utilizar el .map; por lo tanto si no utilizarás el arreglo restante es recomendable mejor utilizar un .forEach. Veamos un ejemplo de un .map que extraiga todas las marcas de los automóviles.

```
const marcas = autos.map(auto => {
    return auto.marca
});
console.log(marcas);
```

El códigoa anterior asigna el valor del nuevo arreglo (creado por .map) a la variable marcas, y al final enviamos a la consola las marcas, si pruebas este código notarás que en tu consola aparecen todas las diferentes marcas de nuestra mini base de datos.

De nuevo el código anterior se puede simplificar gracias a otra de las características de los arrow functions: el return se da por implícito si tu código es de una sola línea.

```
const autos2018 = autos.map(auto => auto.marca);
console.log(autos2018);
```

Supongamos que deseamos extraer todos los automóviles de color Rojo de nuestra mini base de datos, podríamos intentar el siguiente código:

```
const autosRojos = autos.map(auto => {
    if(auto.color === 'Rojo') {
        return auto;
    }
});
console.log(autosRojos);
```

Si pruebas el código anterior notarás que, si bien el nuevo arreglo si extrae los automóviles de color rojo, también el nuevo arreglo tiene una serie de elementos como *undefined*, si deseas realizar una operación como esta, existe otro array method llamado *.filter*

.filter creará un nuevo arreglo con los elementos que pasen una condición o prueba de implementación, en el ejemplo anterior vimos que el código del .map no era el adecuado para filtrar todos los automóviles de color Rojo, pero el .filter si es el adecuado, veamos el código:

```
const autosRojos = autos.filter(auto => {
    return auto.color === 'Rojo'
});
console.log(autosRojos);
```

De nueva cuenta, el código anterior se puede ver beneficiado por 2 características de los arrow functions, las llaves no son necesarias cuando tienes una línea de código, y tampoco es necesaria la palabra return, y de aquí al final de esta entrada daré por hecho estas consideraciones para simplificar el código.

```
// Filtrar todos los automoviles rojos
const autosRojos = autos.filter(auto => auto.color === 'Rojo');
console.log(autosRojos);
```

Nota lo sencillo que es esta sintaxis comparada con la primera que vimos en esta entrada donde utilizabamos un for loop con un if dentro.

Veamos una serie de ejemplos de .filter (sin duda mi array method favorito):

```
// Obtener todos los autos 2018
const autos2018 = autos.filter(auto => auto.year === 2018);
console.log(autos2018);

// Obtener todos los autos de 4 puertas
const autos4puertas = autos.filter(auto => auto.puertas === 4);
console.log(autos4puertas)

// Obtener todos los autos marca Audi
const autosAudi = autos.filter(auto => auto.marca === 'Audi');
console.log(autosAudi);

// Obtener todos los autos cuyo costo es mayor a 30 mil
const mayores30mil = autos.filter(auto => auto.precio > 30000);
console.log(mayores30mil);

// Obtener todos los autos entre 2015 y 2017
const autosRango = autos.filter(auto => auto.year >= 2015 && auto.year <= 2017);
console.log(autosRango);
```

.find encontrará el primer elemento del arreglo que cumpla la condición y retornará ese valor, algo importante es que a diferencia de .filter que crea un arreglo con todos los valores que cumplen una condición, en el caso de .find solo extraerá el valor del primero que cumpla la condición.

En nuestra mini base de datos tenemos 5 automóviles de la marca BMW, veamos un ejemplo de .find.

```
const primerResultado = autos.find(auto => auto.marca === 'BMW');
console.log(primerResultado);
```

Puedes ver que si bien tenemos 5 automóviles de esta marca, .find solo retornará el primero.

.reduce

.reducer ejecutará una función que retornará un acumulado como valor único.

Antes de ver un ejemplo más práctico, veamos una implementación más simple para conocer las diferentes partes de un .reduce, supongamos que tenemos un arreglo de números y queremos saber su suma:

```
const numeros = [1,2,3];
const total = numeros.reduce((total, numero) => total + numero, 0);
console.log(total)
```

El ejemplo anterior mostrará un 6 en la consola, lo que hace .reduce en el ejemplo anterior es recorrer el arreglo de numeros e ir sumando todo en lo que se conoce como un acumulador; es por eso que se pasan 2 valores después de .reduce() el total – que es el acumulador – y el número – que es el numero actual – del arreglo llamado numeros que esta en la línea anterior. Finalmente podrás notar que hay un 0 poco antes de cerrar el paréntesis de .reduce, ese es el valor inicial, que en este caso comenzará en 0, si lo inicias en 5, podrás ver que el total será 11, ya que comenzará a sumar a partir de 5 y sumará 6 (los valores de 1 + 2 + 3).

Ahora, supongamos que el departamento de ventas nos pregunta cuanto dinero podríamos generar si vendiéramos todos los automóviles, por supuesto puedes hacer la suma con los dedos o con una calculadora, pero un .reduce será más rápido y preciso.

```
const totalInventario = autos.reduce((total, auto) => total + auto.precio, 0);
console.log(totalInventario);
```

.some

.some verificará si un elemento existe o no en el arreglo, a diferencia de los otros métodos que te retornan un arreglo nuevo o el valor, en este caso solo te dirá si existe o no un elemento que cumpla la condición o implementación.

Supongamos que un cliente llama buscando un auto de la marca Ferrari, en nuestra base de datos no tenemos ninguno pero podremos comprobarlo con un .some

```
const existe = autos.some( auto => auto.marca === 'Ferrari');
console.log(existe)
```

Si revisamos el resultado del código anterior notarás que la consola solo imprime *false*, debido a que no hubo un registro que cumpliera con esa condición.

Como vimos anteriormente en nuestra base de datos tenemos 5 autos BMW, si hacemos una búsqueda por esta marca:

```
const existe = autos.some( auto => auto.marca === 'BMW');

console.log(existe)
```

Notarás que el resultado es *true*, .some no retornará un arreglo nuevo con los elementos, tampoco te dirá en que posición se encuentra el elemento, solo revisará si existe o no en tu arreglo.