Desestructuración

Desestructuración: se trata de **separar una estructura**, que es o un array o un objeto.

Destructuración básica

```
const elements = [5, 2];
const [first, last] = elements; // first = 5, last = 2
const elements = [5, 4, 3, 2];
const [first, second] = elements; // first = 5, second = 4, rest = discard
const elements = [5, 4, 3, 2];
const [first, , third] = elements; // first = 5, third = 3, rest = discard
const elements = [4];
const [first, second] = elements; // first = 4, second = undefined
```

El truco en estos cuatro casos está en que en la parte derecha utilizamos los [] para indicar que se trata de un array, pero en la parte izquierda los utilizamos para indicar que hacemos una **desestructuración**.

Intercambio de variables

```
// Sin desestructuración
let a = 10; let b = 5; let aux = a; a = b; b = aux;
```

Sin embargo, si utilizamos desestructuración, este ejemplo es mucho más sencillo:

```
// Con desestructuración este método es más sencillo:
let a = 10; let b = 5;
[a, b] = [b, a];
```

Desestructuración

Spread (Expandir)

```
//Se usa el operador … para separar o expandir los elementos del array
function debug(param) { console.log(param); }
// O lo que es lo mismo:
const debug = (param) => console.log(param);
//Con operador spread:
const debug = (param) => console.log(...param);
const array = [1, 2, 3, 4, 5];
debug(array); // 1 2 3 4 5
```

Parámetros Rest (Agrupar)

Una función puede ser llamada con cualquier número de argumentos sin importar cómo sea definida.

Por ejemplo:

```
function sum(a, b) {
        return a + b;
        }
alert( sum(1, 2, 3, 4, 5) );
```

No habrá ningún error por “exceso” de argumentos. Pero sólo los dos primeros serán tomados en cuenta, entonces el resultado del código anterior es 3.

Desestructuración

El resto de los parámetros pueden ser referenciados en la definición de una función con ... seguidos por el nombre del array que los contendrá. Literalmente significa “Reunir los parámetros restantes en un array”.

Por ejemplo:

```
function sumAll(...args) { // args es el nombre del array
        let sum = 0;
        for (let arg of args) sum += arg;
        return sum;
}
Console.log(sumAll(1) ); // 1
Console.log( sumAll(1, 2) ); // 3
Console.log( sumAll(1, 2, 3) ); // 6
```

Podemos elegir obtener los primeros parámetros como variables, y juntar solo el resto:

```
function showName(firstName, lastName, ...titles) {
        console.log( firstName + ' ' + lastName ); // Julio Cesar // el resto va en el array
                titles // por ejemplo titles = ["Cónsul", "Emperador"]
        console.log( titles[0] ); // Cónsul
        console.log( titles[1] ); // Emperador
        console.log( titles.length ); // 2
        }
showName("Julio", "Cesar", "Cónsul", "Emperador")
```

Desestructuración

Reestructuración de arrays

Estas características de desestructuración, podemos aprovecharlas para **reestructurar** un array y **recrear arrays:**

Tenemos un array de 2 elementos [3, 4] y queremos aprovecharlo para crear un nuevo array del 1 al 5. Vamos a hacer uso de la desestructuración para reaprovecharlo:

```
const pair = [3, 4]; // Usando desestructuración + spread
const complete = [1, 2, ...pair, 5]; // [1, 2, 3, 4, 5]
// Sin usar desestructuración
const complete = [1, 2, pair, 5]; // [1, 2, [3, 4], 5]
```

Arrays: método from

Convertir a array

El método estático Array.from() se suele utilizar para convertir variables «parecidas» a los **arrays** (*pero que no son arrays*) en un Array. Este es el caso de variables como Strings o de lista de elementos del DOM:

```
const text = "12345";
text.constructor.name; // "String"
const letters = Array.from(text); // ["1", "2", "3", "4", "5"]
const letters = [...text]; // ["1", "2", "3", "4", "5"]

const divs = document.querySelectorAll("div");
divs.constructor.name; // "NodeList"
const elements = Array.from(divs); // [div, div, div]
const elements = [...divs]; // [div, div, div]
```

Es algo muy similar a lo que hacemos con el operador spread (…). Pero no todos los elementos se pueden convertir a arrays. Si intentamos convertir un undefined o un null, nos dará un error similar a **Uncaught TypeError: null is not iterable**.

Arrays: método from

De forma opcional, Array.from(obj) puede recibir un parámetro adicional: una función que actuará de forma idéntica a una función map().
Veamos el funcionamiento:

```
const text = "12345";
const numbers = Array.from(text, (number) => Number(number)); // [1, 2, 3, 4, 5]
const numbers = Array.from(text, Number); // Equivalente al anterior
// Equivalente a los dos anteriores
const numbers = [...text].map(Number);
```

Observa que en este caso, la función pasada por segundo parámetro del Array.from() se ejecutará por cada uno de los elementos de text, y en este caso concretamente, la función (number) => Number(number) fuerza a convertir cada elemento en un número.

La diferencia respecto al ejemplo anterior, es que en este caso obtienes un array de números, mientras que el anterior obtenías un array de textos.

Array functions

¿Qué son las Array functions?

Son métodos que tiene cualquier variable que sea de tipo Array , y a los que se les pasa, en general, a dichos métodos se les pasa por parámetro una **función callback** y unos parámetros opcionales. A cada método se le pasará una callback que se ejecutará para cada uno de los elementos del Array.
Estas son las **Array functions** que podemos encontrarnos en Javascript:

Método	Descripción
.forEach(ƒ)	Ejecuta la función definida en ƒ por cada uno de los elementos del array.
Comprobaciones	
.every(ƒ)	Comprueba si **todos** los elementos del array cumplen la condición de ƒ.
.some(ƒ)	Comprueba si **al menos** un elemento del array cumple la condición de ƒ.
Transformadores y filtros	
.map(ƒ)	Construye un array con lo que devuelve ƒ por cada elemento del array.
.filter(ƒ)	Filtra un array y se queda sólo con los elementos que cumplen la condición de ƒ.
.flat(level)	Aplana el array al nivel level indicado.
.flatMap(ƒ)	Aplana cada elemento del array, transformándolo según ƒ. Equivale a .map().flat(1).
Búsquedas	
.findIndex(ƒ)	Devuelve la posición del elemento que cumple la condición de ƒ.
.find(ƒ)	Devuelve el elemento que cumple la condición de ƒ.
.findLastIndex(ƒ)	Idem a findIndex(), pero empezando a buscar desde el último elemento al primero.
.findLast(ƒ)	Idem a find(), pero empezando a buscar desde el último elemento al primero.
Acumuladores	
.reduce(ƒ, initial)	Ejecuta ƒ con cada elemento (de izq a der), acumulando el resultado.
.reduceRight(ƒ, initial)	Idem al anterior, pero en orden de derecha a izquierda.

Array functions

Bucles .forEach()

El método forEach() no devuelve nada y espera que se le pase por parámetro una función que se ejecutará **por cada elemento del array**. La función se puede pasar como función tradicional o como función flecha:

```
const letters = ["a", "b", "c", "d"]; // Con funciones por expresión
const f = function () {
            console.log("Un elemento.");
            };
letters.forEach(f);
// Con funciones anónimas
letters.forEach(function () { console.log("Un elemento."); });
// Con funciones flecha letters.forEach(() => console.log("Un elemento."));
```

Este ejemplo no tiene demasiada utilidad. A la **callback** se le pueden pasar varios parámetros opcionales:

- Si se le pasa un **primer parámetro**, este será el elemento del array.
- Si se le pasa un **segundo parámetro**, este será la posición en el array.
- Si se le pasa un **tercer parámetro**, este será el array en cuestión.

Array functions

Veamos un ejemplo:

```
const letters = ["a", "b", "c", "d"];
letters.forEach((element) => console.log(element)); // Devuelve 'a' / 'b' / 'c' / 'd'
letters.forEach((element, index) => console.log(element, index)); // Devuelve 'a' 0 / 'b' 1 / 'c' 2 / 'd' 3
letters.forEach((element, index, array) => console.log(array[0])); // Devuelve 'a' / 'a' / 'a' / 'a‘
```

Al método forEach() se le puede pasar un segundo parámetro arg, que representa el valor que sobreescribiría a la palabra clave this en el código dentro de la función **callback**. De necesitar esta funcionalidad, no podrías utilizar las funciones flecha, ya que el this no tiene efecto en ellas.

El método .every() (Todos)

Permite comprobar si **todos y cada uno** de los elementos cumple una condición

```
const letters = ["a", "b", "c", "d"];
letters.every((letter) => letter.length === 1); // true
```

En este caso, la magia está en el **callback**. La condición es que la longitud de cada elemento del array sea 1. Si todos los elementos del array devuelven true, entonces every() devolverá true.

Array functions

Expandimos el ejemplo anterior:

```
const letters = ["a", "b", "c", "d"];
// Esta función se ejecuta por cada elemento del array
const condition = function (letter) { // Si el tamaño del elemento (string) es igual a 1
        if (letter.length == 1) {
                return true; }
        else { return false; }
}; // Si todos los elementos devuelven true, devuelve true
letters.every(condition); // true
```

El método .some() (Al menos uno)

Devuelve true si **al menos uno** de los elementos del array, cumple la condición definida por el **callback**:

```
const letters = ["a", "bb", "c", "d"];
letters.some((element) => element.length == 2); // true
```

Observa que en este ejemplo, el método some() devuelve true porque existe al menos un elemento del array con una longitud de 2 caracteres.

Array functions

El método .map()

Devuelve un nuevo array donde cada uno de sus elementos será lo que devuelva la función **callback** por cada uno de los elementos del array original:

```
const names = ["Ana", "Pablo", "Pedro", "Pancracio", "Heriberto"];
const nameSizes = names.map((name) => name.length);
nameSizes; // Devuelve [3, 5, 5, 9, 9]
```

El método .filter()

Permite filtrar los elementos de un array y devolver un nuevo array con sólo los elementos que queramos. Para ello, utilizaremos la función **callback** para establecer una condición que devuelve true sólo en los elementos que nos interesen:

```
const names = ["Ana", "Pablo", "Pedro", "Pancracio", "Heriberto"];
const filteredNames = names.filter((name) => name.startsWith("P"));
filteredNames; // Devuelve ['Pablo', 'Pedro', 'Pancracio']
```

Array functions

El método .flatMap()

Revisa todos los elementos del array en busca de arrays anidados, y los **aplana** hasta el nivel level indicado por parámetro.

```
const values = [10, 15, 20, [25, 30], 35, [40, 45, [50, 55], 60]];
values.flat(0); // [10, 15, 20, [25, 30], 35, [40, 45, [50, 55], 60]];
values.flat(1); // [10, 15, 20, 25, 30, 35, 40, 45, [50, 55], 60];
values.flat(2); // [10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60]; // Idem al anterior, pero si hubieran más niveles los aplanaría todos
values.flat(Infinity);
```

Inicialmente el array values tiene arrays hasta un **nivel 3**. Con .flat() podemos indicar hasta que nivel queremos «aplanarlo».

Flat es la base del método .flatMap(f), que es el equivalente a realizar la operación .map(f).flat(1):

```
const values = [10, 15, 20, [25, 30], 35, [40, 45, [50, 55], 60]];
values.flatMap(element => Array.isArray(element) ? element.length : 1 ); // [1, 1, 1, 2, 1, 4]
```

Primero recorre cada uno de los elementos mediante un map() para transformarlos: si son un array, devuelve su cantidad de elementos, si no es un array, devuelve 1.

Finalmente, si el array resultante tuviera algún array entre sus elementos (*que en este caso es imposible*), le aplicaría un flat(1).

Array functions

El método .find() y .findIndex()

Se utilizan para buscar elementos de un array mediante una condición.El primero devuelve el elemento mientras que el segundo devuelve su posición en el array original.
Veamos como funcionan:

```
const names = ["Ana", "Pablo", "Pedro", "Pancracio", "Heriberto"];
names.find((name) => name.length == 5); // 'Pablo'
names.findIndex((name) => name.length == 5); // 1
```

En el caso de no encontrar ningún elemento, find() devolverá Undefined , mientras que findIndex(), que debe devolver un Number (-1).

El método .findLast() y .findLastIndex()

Equivalentes a las anteriores pero buscando elementos desde derecha a izquierda

```
const names = ["Ana", "Pablo", "Pedro", "Pancracio", "Heriberto"];
names.findLast((name) => name.length == 5); // 'Pedro'
names.findLastIndex((name) => name.length == 5); // 2
```

Array functions

El método .reduce()

Tanto reduce() como reduceRight() se encargan de recorrer todos los elementos del array, e ir acumulando sus valores (*o alguna operación diferente*) y sumarlo todo, para devolver su resultado final.

La **callback** en lugar de tener los clásicos parámetros opcionales (element, index, array) tiene (first, second, iteration, array).

En la primera iteración, first contiene el valor del primer elemento del array y second del segundo. En siguientes iteraciones, first es el acumulador que contiene lo que devolvió el **callback** en la iteración anterior, mientras que second es el siguiente elemento del array, y así sucesivamente.

```
const numbers = [95, 5, 25, 10, 25];
numbers.reduce((first, second) => {
        console.log(`F=${first} S=${second}`);
        return first + second;
        }); // F=95 S=5 (1ª iteración: elemento 1: 95 + elemento 2: 5) = 100
        // F=100 S=25 (2ª iteración: 100 + elemento 3: 25) = 125
        // F=125 S=10 (3ª iteración: 125 + elemento 4: 10) = 135
        // F=135 S=25 (4ª iteración: 135 + elemento 5: 25) = 160
        // Finalmente, devuelve 160
```

Array functions

El método .reduceRight()

Se utiliza como acumulador de elementos de derecha a izquierda.

```
const numbers = [95, 5, 25, 10, 25];
numbers.reduce((first, second) => first - second); // 95 - 5 - 25 - 10 - 25. Devuelve 30
numbers.reduceRight((first, second) => first - second); // 25 - 10 - 25 - 5 - 95. Devuelve -110
```

Parámetro inicial

Es posible indicar un segundo parámetro opcional en el .reduce(). Es es el valor inicial que quieres tomar en el **reduce**. En el primer ejemplo anterior, se realizan 4 iteraciones. Al indicar este valor inicial de **cero** se realizan 5 iteraciones:

```
const numbers = [95, 5, 25, 10, 25];
numbers.reduce((accumulator, nextElement) => {
        console.log(`F=${accumulator} S=${nextElement}`);
        return accumulator + nextElement;
        }, 0); // F=0 S=95 (iteración inicial): 0 + elemento 1: 95) = 95
        // F=95 S=5 (1ª iteración: elemento 1: 95 + elemento 2: 5) = 100
        // F=100 S=25 (2ª iteración: 100 + elemento 3: 25) = 125
        // F=125 S=10 (3ª iteración: 125 + elemento 4: 10) = 135
        // F=135 S=25 (4ª iteración: 135 + elemento 5: 25) = 160
        // Finalmente, devuelve 160
```

Algoritmo Ordenación Arrays

Los métodos .Sort() y .toSorted() sirven para ordenar los elementos de un *array* según un criterio. Este criterio se puede definir con el uso opcional de un *callback.*

El *callback* se basará en **la comparación de dos valores para el método sort.** Estos dos valores serán los elementos del *array*. Es decir, el método *sort* en JavaScript recorre el *array* comparando un elemento con el otro y, de esta forma, los ordena. Por ello, el *callback* debe tener dos valores para el sort en JavaScript. Dependiendo con lo que devuelve esa función de comparación, sé si tengo que dar la vuelta a esos elementos o no.

Ordenación ascendente:

```
const c = [1, 3, 2, 4];
c.sort ((a,b) => a-b); // resultado esperado [1, 2, 3, 4]
```

El *callback* revisa si el valor *a* es mayor o menor que *b.* En función de este resultado, establece un número que a su vez definirá la posición del elemento:

```
const sorted = array1.sort (a, b) => {
        if (a < b) { return -1 }
        else if (a > b) { return 1 }
        else { return 0 }
        }
```

- *Si a* es menor que *b, devolverá -1, y a* se colocará al principio, hacia la izquierda.
- Si *a* es mayor que *b, devolverá 1, y a* se colocará hacia la derecha.
- Si *a* y *b* son del mismo valor, devolverá 0, entonces no importa el orden y no hace falta moverlos.

Esta función equivale a la escrita anteriormente, más resumida.

Funciones Arrow

Ámbito léxico de this

Una de las principales diferencias de las **arrow functions** respecto a las funciones tradicionales, es el valor de la palabra clave this, que no siempre es el mismo.
Por ejemplo: si utilizamos una función de forma global en nuestro programa, no notaremos ninguna diferencia:

```
const a = function () { console.log(this); };
a(); // Window
const b = () => { console.log(this); };
b(); // Window
```

En ambos casos, el valor de this es el objeto global Window o globalThis, que es una referencia al objeto que representa la ventana o pestaña del navegador: si estamos en un contexto global, this hace referencia a **esta ventana del navegador**. Sin embargo, si utilizamos una función en el interior de un objeto, como suele ser el caso más habitual, si encontraremos diferencias.
Observa el siguiente código:

```
padre = { a: function () { console.log(this); },
          b: () => { console.log(this); },
};
padre.a(); // { a: f(), b: f() } (el padre)
```