

JavaScript Avanzado

Funciones de orden superior



Agenda de hoy



- Funciones de orden superior
 - Fundamentos
 - Ejemplo
 - Aplicación en Arrays
 - iteración: forEach()
 - Búsquedas
 - find()
 - findIndex() !== indexOf()
 - filter()
- De retorno boolean
 - some()
 - every()
- De transformación
 - reduce()
 - map()
 - sort()
 - ascendente / descendente
 - ¿Qué es un método destructivo?
 - flat()



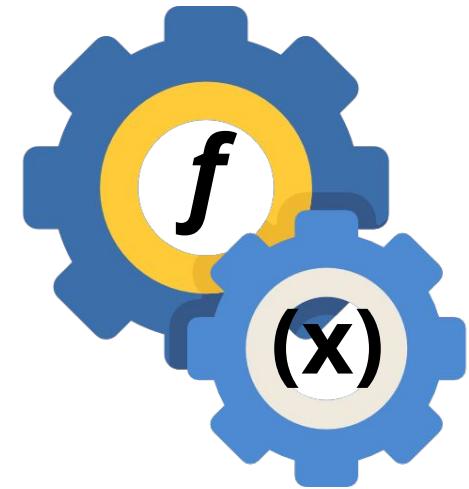
Funciones de orden superior



Funciones de orden superior

Las **funciones de orden superior** son una característica importante en JS la cual permite tratar a las funciones como cualquier otro valor, números, cadenas, u objetos.

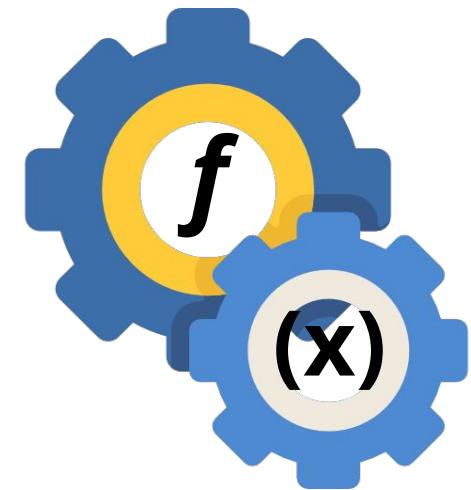
Estas funciones son capaces de recibir otras funciones como argumentos y/o devolverlas como resultado. Esto permite una mayor flexibilidad y expresividad en la escritura de código.



Funciones de orden superior (ejemplos)

Entre las diferentes categorías que se establecen en las funciones de orden superior, encontramos a:

- funciones como valores
- funciones que reciben funciones como argumentos
- funciones que devuelven funciones



Funciones de orden superior (ejemplos)

Funciones como valores

```
● ● ● Funciones de orden superior

const saludar = () => {
  console.log('¡Hola!');
};

const funcionDeOrdenSuperior = (fn) => {
  fn(); // Llamada a la función que se pasa como argumento
};

funcionDeOrdenSuperior(saludar); // Llamada a la función de
orden superior pasando la función saludar como argumento
```



Funciones de orden superior (ejemplos)

Funciones que reciben funciones como argumentos

```
••• Funciones de orden superior

const sumar = (a, b) => a + b;
const restar = (a, b) => a - b;

const operar = (fn, a, b) => {
    return fn(a, b); // Llamada a la función que se pasa como
                     argumento
};

console.log(operar(sumar, 5, 3)); // 8
console.log(operar(restar, 5, 3)); // 2
```



Funciones de orden superior (ejemplos)

Funciones que devuelven funciones

```
● ● ● Funciones de orden superior

const multiplicador = (factor) => {
  return (numero) => {
    return numero * factor;
  };
};

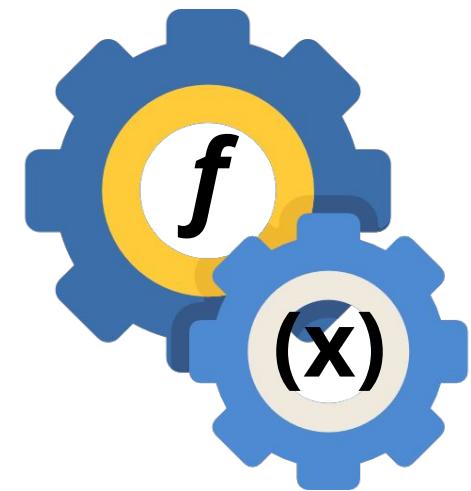
const duplicar = multiplicador(2);
const triplicar = multiplicador(3);

console.log(duplicar(5)); // 10
console.log(triplicar(5)); // 15
```



Funciones de orden superior (ejemplo)

Dentro de las ventajas de crear y/o utilizar funciones de orden superior, podemos destacar a la **Abstracción y reutilización de código**, gracias a que éstas poseen la característica de que nos abstraigamos de la lógica, y reutilicemos el código de forma más eficiente.



Así lograremos escribir código más limpio, modular, y fácil de mantener a lo largo del tiempo.

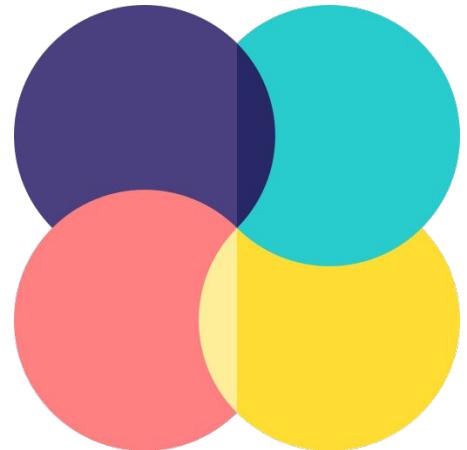
Aplicación en Arrays



Aplicación en Arrays

Si bien podemos crear nuestras propias funciones de orden superior para optimizar al máximo nuestro código, también contamos con muchas funciones de orden superior integradas en los arrays JS.

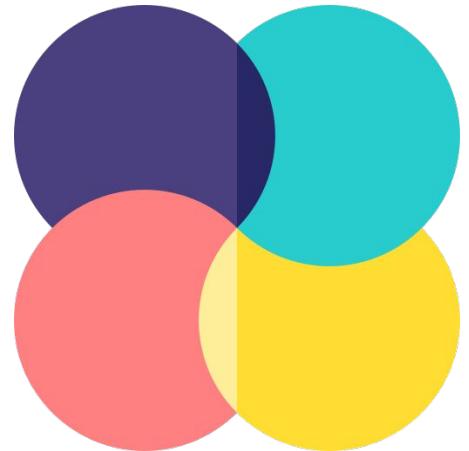
Estas nos permiten exprimir al máximo las capacidades de procesar y obtener resultados efectivos, a través de un mecanismo de abstracción fácil de comprender.



Aplicación en Arrays

La mayoría de las funciones de orden superior, aparecieron en JS desde hace varios años, gracias al modelo colaborativo de compartir conocimientos en Internet a través de foros de discusión.

Originalmente, la iteración de un array de elementos u objetos, se realizaba a través del ciclo for convencional.



Aplicación en Arrays

Ejemplo de iteración de un array de objetos, previo a la existencia del método **forEach()**.

```
● ● ● Funciones de orden superior  
  
const productos = [  
  { id: 1, nombre: 'Producto 1', precio: 100 },  
  { id: 2, nombre: 'Producto 2', precio: 200 },  
  { id: 3, nombre: 'Producto 3', precio: 150 }  
];  
  
function forEachProduct() {  
  for (let i = 0; i < productos.length; i++) {  
    console.log(productos[i].nombre); //imprime el nombre del producto  
  }  
}
```



Aplicación en Arrays

JQuery también fue parte de la inspiración de estas funciones de orden superior, a partir de la propuesta de su método `$.each()`, para iterar elementos de un objeto u array.

```
● ● ● Funciones de orden superior

const colores = [
  { 'rojo': '#f00' },
  { 'verde': '#0f0' },
  { 'azul': '#00f' }
];

$.each(colores, function() {
  $.each(this, function(name, value) {
    console.log(` ${name} = ${value}`);
  });
});
```



Aplicación en Arrays

Los siguientes ejemplos a repasar, están aplicados sobre este array modelo. Ten presente esto por sí, al repasar este contenido, quieres ir probando el código de cada explicación.



Funciones de orden superior

```
const productos = [
  { id: 1, nombre: 'Producto 1', precio: 100 },
  { id: 2, nombre: 'Producto 2', precio: 200 },
  { id: 3, nombre: 'Producto 3', precio: 150 }
];
```

Iteración: forEach()



Iteración: forEach()

ForEach() es un método incorporado en JavaScript que se utiliza para iterar sobre elementos de un arreglo y ejecutar una función (*callback*) en cada uno de ellos.

- Llegó a todos los navegadores web, entre 2013 y 2014
- Se utiliza como método de un array
- No retorna ningún resultado que pueda guardarse en una variable o constante
- No modifica el array original



Iteración: forEach()

ForEach() itera cada elemento u objeto del array en cuestión. En la función que recibe como argumento, pasa cada uno de los elementos u objetos, para que podamos tratarlo, de acuerdo a nuestra necesidad.



Funciones de orden superior

```
// Ejemplo de uso de forEach() para imprimir información de cada producto
productos.forEach((producto) => {
  console.log(`ID: ${producto.id}, Nombre: ${producto.nombre},
  Precio: ${producto.precio}`);
});
```



Búsquedas: `find()`



Búsquedas: find()

Find() es un método incorporado en JavaScript que se utiliza para encontrar el primer elemento en un arreglo que cumpla con una condición especificada, y devuelve el valor del primer elemento que cumple con esa condición.

- Retorna el elemento u objeto
- Si no encuentra coincidencia, retorna **undefined**
- No modifica el array original



Búsquedas: find()

Debemos tener presente declarar una variable o constante para recibir la respuesta del método `find()`. Allí nos dejará un objeto o elemento como resultado, que coincide con el parámetro indicado en la búsqueda.



Funciones de orden superior

```
// Ejemplo de uso de find() para encontrar un producto por su ID
const productoEncontrado = productos.find((producto) => producto.id === 2);

console.log(productoEncontrado);
```



Búsquedas: find()

Es importante tener presente el tipo de datos a buscar. Si el mismo es numérico o string, aplicarlo correctamente en la definición de la búsqueda.



Funciones de orden superior

```
// Ejemplo de uso de find() para encontrar un producto por su nombre
const productoEncontrado = productos.find((producto) => producto.nombre ===
'Producto 1');

console.log(productoEncontrado);
```



Búsquedas: find()

De igual forma, si en una propiedad del tipo String, del array en cuestión, queremos identificar por parte de la descripción, podemos recurrir a métodos de array encadenados para solventar esta necesidad.



Funciones de orden superior

```
// Ejemplo de uso de find() para encontrar un producto utilizando algún
// carácter o frase específica dentro de una propiedad del tipo string

const productoEncontrado = productos.find((producto) =>
  producto.nombre.includes('3'));

console.log(productoEncontrado);
```



Búsquedas: find()

Y más allá de que los datos estén normalizados entre lo que se recibe como parámetro de búsqueda y la información almacenada en el array, también podemos sumar una normalización de texto mediante encadenamiento de métodos *String*.

```
● ● ● Funciones de orden superior

const parametroAbuscar = 'DUCTO 3'; //se ingresa parte del nombre sin
normalizar el mismo

const productoEncontrado = productos.find((producto) =>
producto.nombre.toLowerCase().includes(parametroAbuscar.toLowerCase()));

console.log(productoEncontrado);
```



Búsquedas: find()

Por último, también podemos realizar una búsqueda, combinando dos o más propiedades en un array de objetos. E incluso aplicar operadores de comparación diferentes al igual o estrictamente igual.



Funciones de orden superior

```
const busquedaCombinada = productos.find((producto) =>
  producto.nombre.includes('ducto') && producto.precio > 199);

console.table(busquedaCombinada);
```

Búsquedas: `findIndex()`



Búsquedas: findIndex()

FindIndex() es similar al método **find()** pero, en lugar de devolver el valor del primer elemento que cumple con una condición, **devuelve el índice del primer elemento** que cumple con dicha condición en un arreglo.

- devuelve el índice donde se encuentra el elemento
- si no encuentra coincidencias, devuelve **-1**
- posee similitud con el método **indexOf()**, con la diferencia de que este último se utiliza en arrays de elementos, mientras que **findIndex()** está optimizado para array de objetos



Búsquedas: `findIndex()`

Aquí tenemos un ejemplo de aplicación del método `findIndex()`. En la variable o constante resultante obtendremos la posición o índice del objeto dentro del array que lo contiene.

```
● ● ● Funciones de orden superior  
  
// Ejemplo de uso de findIndex() para encontrar el índice de un producto por su ID  
const indiceProductoEncontrado = productos.findIndex((producto) =>  
    producto.id === 2);  
  
console.log(indiceProductoEncontrado);
```



Búsquedas: `findIndex()`

También podemos aplicar las mismas alternativas en la búsqueda, mencionadas anteriormente con el método `find()`, encadenando métodos de `string`, y utilizando diversos operadores de comparación, como también buscando hacer coincidir dos o más propiedades en un array de objetos.

```
● ● ● Funciones de orden superior  
  
// Ejemplo de uso de findIndex() para encontrar el índice de un producto por su ID  
const indiceProductoEncontrado = productos.findIndex((producto) =>  
  producto.id === 2);  
  
console.log(indiceProductoEncontrado);
```



Búsquedas: `findIndex()`

El uso de este método es completamente válido para, por ejemplo, identificar el índice de un objeto dentro de un array en cuestión, y luego combinarlo con el método `.splice()` para remover el objeto en cuestión, de forma efectiva.



Funciones de orden superior

```
const indiceProductoEncontrado = carrito.findIndex((producto) => producto.id
=== 2);

console.log(indiceProductoEncontrado);
if (indiceProductoEncontrado > -1) {
  carrito.splice(indiceProductoEncontrado, 1);
}
```



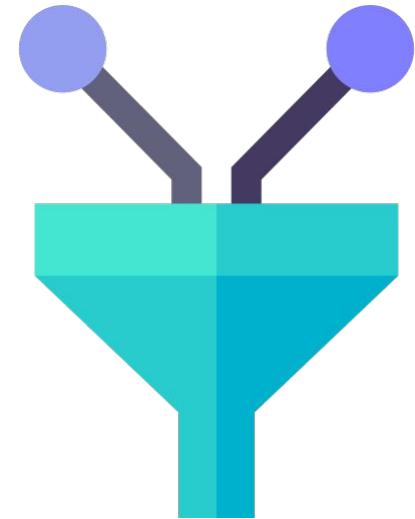
Búsquedas: filter()



Búsquedas: filter()

Filter() es un método incorporado en arrays que se utiliza para crear un nuevo arreglo que contiene sólo los elementos de un arreglo original que cumplan con una condición especificada.

- itera el array de principio a fin
- retorna un array con una o más coincidencias
- si no encuentra coincidencias, retorna un array vacío



Búsquedas: filter()

En este ejemplo, buscamos crear un nuevo array a partir del array `productos`, que sólo contenga los productos cuya propiedad `precio` sea mayor a **100**.

De acuerdo a nuestro array modelo, obtendremos dos productos en el array resultante.



Funciones de orden superior

```
// Ejemplo de uso de filter() para obtener productos con precio mayor a 100
const productosMayoresA100 = productos.filter((producto) => producto.precio >
100);

console.log(productosMayoresA100);
```



Búsquedas: filter()

De igual forma a los diferentes usos que le podemos dar al método `find()`, `filter()` también nos da todo ese tipo de flexibilidad. Podemos buscar propiedades del tipo **number**, **boolean**, **string**, utilizar operadores de comparación diversos, combinar dos o más parámetros en la búsqueda, aplicar encadenamiento de métodos de string, etcétera.



Funciones de orden superior

```
// Ejemplo de uso de filter() para obtener productos con precio mayor a 100
const productosMayoresA100 = productos.filter((producto) => producto.precio >
100 && producto.id > 2);

console.log(productosMayoresA100);
```



De retorno booleano: some()



De retorno booleano: some()

some() es una función que itera el array de principio a fin, y permite determinar o comprobar si al menos uno de los objetos del array, cumple con una característica específica.

- devuelve siempre un valor booleano
 - si al menos un objeto cumple, retorna true
 - si ninguno de ellos la cumple, retorna false



De retorno booleano: some()

Modificamos ligeramente nuestro array de objetos modelo, agregando en este caso una nueva propiedad llamada **stock**. Sobre la misma veremos los ejemplos a continuación:

```
● ● ● Funciones de orden superior

// Definimos el array de objetos de productos
const productos = [
  { id: 1, nombre: 'Producto 1', precio: 100, stock: 0 },
  { id: 2, nombre: 'Producto 2', precio: 220, stock: 5 },
  { id: 3, nombre: 'Producto 3', precio: 300, stock: 2 },
  { id: 4, nombre: 'Producto 4', precio: 350, stock: 0 }
];
```



De retorno booleano: some()

Si al menos uno de los productos cumple con esta condición, la función **some()** devuelve **true**, de lo contrario, devuelve **false**.

```
● ● ● Funciones de orden superior

// Verificamos con some() si al menos un producto está disponible en stock
const algunProductoDisponible = productos.some(producto => producto.stock > 0);

// Verificamos el resultado
if (algunProductoDisponible) {
    console.log('Al menos un producto está disponible en stock.');
} else {
    console.log('Ningún producto está disponible en stock.');
}
```



De retorno booleano: every()



De retorno booleano: every()

every() es una función que itera el array de principio a fin, y permite determinar si todos los objetos del array, cumplen con una característica específica.

- devuelve siempre un valor booleano
 - si todos los objetos cumplen, retorna true
 - si uno de ellos no cumple, retorna false



De retorno booleano: every()

En este ejemplo, recorremos el array **productos** para validar que todos los objetos almacenados en este, tengan su propiedad **stock** con un valor 0 (cero) o superior.

```
● ● ● Funciones de orden superior

const conStockNegativo = productos.every(producto => producto.stock < 0);

// Verificamos el resultado
if (conStockNegativo) {
  console.log('Todos los productos tienen el stock correcto.');
} else {
  console.log('Existen algunos productos con stock en negativo.');
}
```



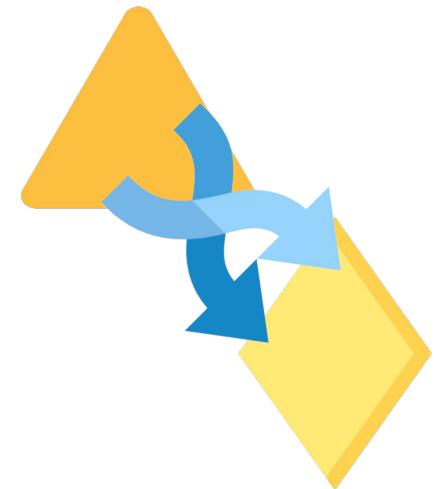
De transformación



De transformación

Algunas funciones de orden superior se enmarcan dentro de la categoría '**de transformación**'.

A través de estas podemos trabajar con un conjunto de datos y convertir los mismos en un nuevo conjunto de datos, tomando parte de su estructura original, creando nuevos datos a partir de la estructura base, o incluso obteniendo resultados acotados a nuestra necesidad.



De transformación: reduce()

La función **reduce()** permite reducir un arreglo a un solo valor, aplicando una función (callback) en cada elemento del arreglo.

1. itera el array de principio a fin
2. recibe dos parámetros:
 - a. un valor acumulador que retornará como resultado
 - b. cada elemento u objeto del array
3. posee un tercer parámetro en el desarrollo de la función el cual establece el valor inicial del acumulador



De transformación: reduce()

Trabajaremos el ejemplo del método **reduce()** sobre un array de objetos denominado **carrito**. De éste deseamos conocer cuál es el valor total del carrito, de acuerdo a los productos almacenados.



Funciones de orden superior

```
const carrito = [
  { id: 1, nombre: 'Producto 1', precio: 100 },
  { id: 2, nombre: 'Producto 2', precio: 200 },
  { id: 3, nombre: 'Producto 3', precio: 150 }
];
```



De transformación: reduce()

El parámetro **acumulador**, se utiliza en la función interna para receptionar el valor de cada propiedad del array de objetos carrito. De esta forma, sumamos uno a uno los productos almacenados en éste.



Funciones de orden superior

```
//calculamos el precio total de todos los productos con el método reduce()  
const precioTotal = carrito.reduce([acumulador] producto) => [acumulador] +  
producto.precio, 0);  
  
console.log(precioTotal);
```

De transformación: reduce()

El tercer parámetro interno, se utiliza para establecer el valor inicial del parámetro **acumulador**. Imaginemos a este como si declarásemos una variable llamada **acumulador** y le establecemos su valor inicial = **0**.



Funciones de orden superior

```
//calculamos el precio total de todos los productos con el método reduce()  
const precioTotal = carrito.reduce((acumulador, producto) => acumulador +  
producto.precio, 0);  
  
console.log(precioTotal);
```

De transformación: reduce()

Si deseamos aplicar un descuento numérico fijo sobre el valor total, podemos reemplazar el valor inicial **0**, por una variable o constante que posea un valor numérico en negativo.



Funciones de orden superior

```
const descuentoFijo = -35;

const precioTotal = carrito.reduce((acumulador, producto) => acumulador +
producto.precio, descuentoFijo);

console.log(precioTotal);
```



De transformación: reduce()

De igual forma, podemos realizar una operación matemática antepuesta al acumulador, para así calcular, por ejemplo, impuestos necesarios a incluir en el resultado acumulado.

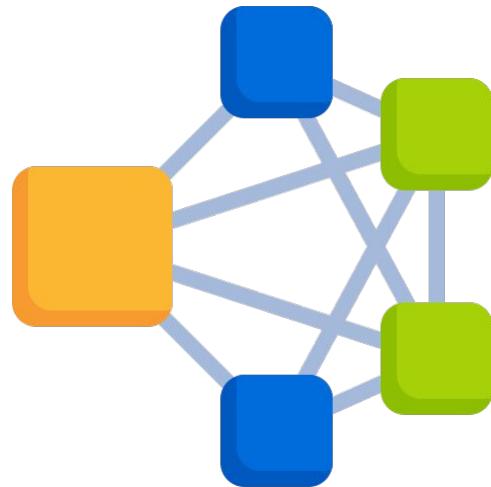
```
● ● ● Funciones de orden superior  
const IVA = 1.21;  
  
const precioConIVA = carrito.reduce((acumulador, producto) => acumulador +  
(producto.precio * IVA), 0);  
  
console.log(precioConIVA);
```



De transformación: map()

La función **map()** se utiliza en arrays para crear uno nuevo, a partir del array existente, el cual estamos iterando.

Toma una función de transformación como argumento y la aplica a cada elemento del arreglo original, retornando un nuevo arreglo con los resultados de aplicar la función a cada elemento.



De transformación: map()

Continuaremos trabajando con el array `productos` para brindar los ejemplos pertinentes pero, ahora con más propiedades incluídas en éste.

```
● ● ● Funciones de orden superior  
const productos = [  
  { id: 1, nombre: 'TV 55', precio: 1000, stock: 30, categoria: 'Video'},  
  { id: 2, nombre: 'Laptop', precio: 1500, stock: 100, categoria: 'Computación'},  
  { id: 3, nombre: 'iPhone 8', precio: 800, stock: 42, categoria: 'Telefonía'},  
  { id: 4, nombre: 'Tablet', precio: 500, stock: 71, categoria: 'Computación'},  
  { id: 5, nombre: 'Pods', precio: 100, stock: 28, categoria: 'Audio'},  
  { id: 6, nombre: 'MP3 player', precio: 200, stock: 11, categoria: 'Audio'},  
  { id: 7, nombre: 'Videocámara', precio: 300, stock: 22, categoria: 'Video'},  
  { id: 8, nombre: 'Smartwatch', precio: 250, stock: 88, categoria: 'Computación'},  
  { id: 9, nombre: 'Impresora', precio: 150, stock: 14, categoria: 'Accesorios'},  
  { id: 10, nombre: 'Altavoces', precio: 120, stock: 18, categoria: 'Audio'}  
];
```



De transformación: map()

Partiendo del array original, podemos crear uno nuevo, eligiendo solo determinadas columnas del original, y hasta agregando un campo calculado, como es el caso de **precioConIVA**.

```
● ● ● Funciones de orden superior

const preciosConIVA = productos.map(producto => {
  return {
    nombre: producto.nombre,
    stock: producto.stock,
    precio: producto.precio,
    precioConIVA: producto.precio * 1.21,
    categoria: producto.categoría
  };
});

console.table(preciosConIVA);
```



De transformación: map()

Incluso diferentes transformaciones que nos permitan proyectar el valor de un producto con un determinado descuento, o un determinado incremento.

```
... Funciones de orden superior

const productosIncrementosDescuentos = productos.map(producto => {
    return {
        nombre: producto.nombre,
        precio: producto.precio,
        precio10off: producto.precio * 0.90,
        precio12up: producto.precio * 1.12
    };
});

console.log(productosIncrementosDescuentos);
```



De transformación: map()

O, en aquellos casos donde alguna propiedad del tipo string viene sin normalizar, podemos aplicar una normalización estándar, ya sea para mostrar en pantalla y/o para luego aplicar búsquedas o filtros más efectivos.

```
••• Funciones de orden superior  
  
const productosNormalizadosMayusculas = productos.map(producto => {  
    return {  
        nombre: producto.nombre.toUpperCase(),  
        precio: producto.precio,  
    };  
});  
  
console.table(productosNormalizadosMayusculas);
```



De transformación: map()

También podemos generar un array simple, o de elementos, utilizando sólo una de las propiedades de todos los objetos que conforman el array en cuestión.



Funciones de orden superior

```
const arraySimpleCategorias = productos.map(producto => producto.categoría);

console.table(arraySimpleCategorias);
```

De transformación: map()

Y para el caso de necesitar limpiar duplicados, podemos complementar el método `.map()` con la clase JS `Set()`, para así eliminar los elementos duplicados rápidamente.



Funciones de orden superior

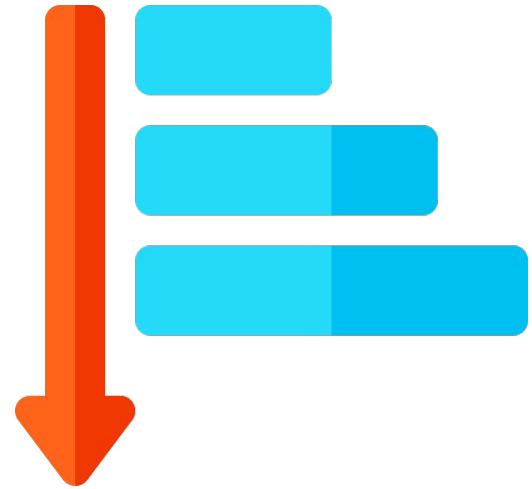
```
const arraySimpleCategorias = ['Video', 'Computación', 'Telefonía', 'Audio',  
'Accesorios', 'Video', 'Computación'];  
  
const categoriasUnicas = [...new Set(categorias)];  
  
console.log(categoriasUnicas);
```



De transformación: sort()

La función **sort()** se utiliza en arrays de objetos para ordenar su contenido in-place, a través de alguna de las propiedades que determinemos.

Este método, difiere del método homónimo aplicado en array de elementos, ya que en los arrays de elementos solo debemos invocarlo, mientras que en array de objetos, debemos determinar dos parámetros y la propiedad por la cual deseamos ordenar.



De transformación: sort()

```
productos.sort((a, b)=> {
  if (a.precio > b.precio) {
    return 1
  }
  if (a.precio < b.precio) {
    return -1
  }
  return 0
})
```

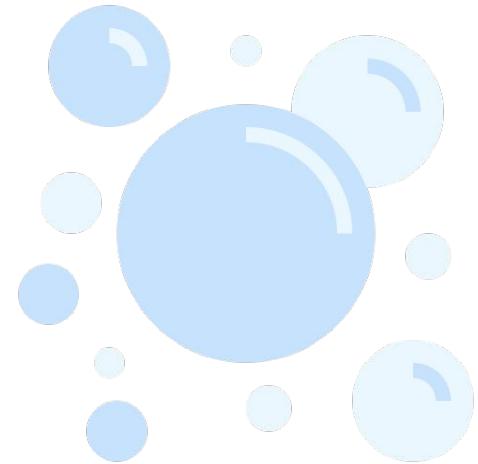
El método **sort()** recibe dos parámetros como argumento, comúnmente llamados **a** y **b**. Internamente realiza una comparación sobre estos dos elementos y devuelve un valor negativo (-1) si ‘a’ es menor que ‘b’. Si es mayor, devuelve un valor positivo (1), o un valor (0) si son iguales.



De transformación: sort()

Sort() en JS se apoya en los fundamentos del [ordenamiento burbuja](#), un mecanismo clásico utilizado para ordenar colecciones y arreglos en diferentes lenguajes de programación, de forma manual.

Es útil para pequeños arrays, pero si el array contiene cientos o miles de elementos, este mecanismo de ordenamiento se torna notablemente lento.



De transformación: sort()

Internamente, el método va ordenando los objetos del array por la propiedad especificada, y finalmente obtenemos un orden ascendente en el mismo.

```
● ● ● Funciones de orden superior  
[  
  {id: 5, nombre: 'Pods', precio: 100, stock: 28, categoria: 'Audio'},  
  {id: 10, nombre: 'Altavoces', precio: 120, stock: 18, categoria: 'Audio'},  
  {id: 9, nombre: 'Impresora', precio: 150, stock: 14, categoria: 'Accesorios'},  
  {id: 6, nombre: 'MP3 player', precio: 200, stock: 11, categoria: 'Audio'},  
  {id: 8, nombre: 'Smartwatch', precio: 250, stock: 88, categoria: 'Computación'},  
  {id: 7, nombre: 'Videocámara', precio: 300, stock: 22, categoria: 'Video'},  
  {id: 4, nombre: 'Tablet', precio: 500, stock: 71, categoria: 'Computación'},  
  {id: 3, nombre: 'iPhone 8', precio: 800, stock: 42, categoria: 'Telefonía'},  
  {id: 1, nombre: 'TV 55', precio: 1000, stock: 30, categoria: 'Video'},  
  {id: 2, nombre: 'Laptop', precio: 1500, stock: 100, categoria: 'Computación'}  
]
```



De transformación: sort()

```
productos.sort((a, b)=> {
  if (a.precio < b.precio) {
    return 1
  }
  if (a.precio > b.precio) {
    return -1
  }
  return 0
})
```

Si deseamos ordenar de forma descendente, simplemente invertimos la comparación (< ó >) que realizamos dentro de **.sort()**.

Alternativamente, podemos invertir solo el signo de los dos primeros **return**.



De transformación: flat()

Flat() es un método que se utiliza en JavaScript para "*aplanar*" un array multidimensional, es decir, convertir un array anidado en un array plano de una sola dimensión.

Este método crea un nuevo array o deja en el array principal, todos los elementos de los sub-arrays concatenados en un mismo nivel de uso.



[..., ..., ...], [...], [...]

De transformación: flat()

```
const productos = [  
  { id: 1, nombre: 'TV 55', precio: 1000, stock: 30, categoria: 'Video'},  
  { id: 2, nombre: 'Laptop', precio: 1500, stock: 100, categoria: 'Computación'},  
  { id: 3, nombre: 'iPhone 8', precio: 800, stock: 42, categoria: 'Telefonía'},  
  { id: 4, nombre: 'Tablet', precio: 500, stock: 71, categoria: 'Computación'},  
  { id: 5, nombre: 'Pods', precio: 100, stock: 28, categoria: 'Audio'},  
  { id: 6, nombre: 'MP3 player', precio: 200, stock: 11, categoria: 'Audio'},  
  { id: 7, nombre: 'Videocámara', precio: 300, stock: 22, categoria: 'Video'},  
  { id: 8, nombre: 'Smartwatch', precio: 250, stock: 88, categoria: 'Computación'},  
  { id: 9, nombre: 'Impresora', precio: 150, stock: 14, categoria: 'Accesorios'},  
  { id: 10, nombre: 'Altavoces', precio: 120, stock: 18, categoria: 'Audio'}  
];  
  
const otrosProductos = [  
  { id: 11, nombre: 'Printer HP', precio: 550, stock: 7, categoria: 'Accesorios'},  
  { id: 12, nombre: 'Altavoces BT', precio: 199, stock: 8, categoria: 'Audio'}  
];  
  
productos.push(otrosProductos);
```

Suponiendo que tenemos un array llamado **otrosProductos**, con un set de datos que necesitamos unificar en el array principal.

Podemos agregar este array dentro del principal utilizando **productos.push(otrosProductos)**. Esto nos dará como resultado un array de dos dimensiones.



De transformación: flat()

Este array multidimensional puede ser unificado generando un nuevo array mediante el método `.flat()`.

En este, podemos especificar el nivel de profundidad que deseamos aplicar sobre el array, por si el array de la segunda dimensión posee un array más como elemento contenido.

```
console.table(productos)
```

VM773:1

(índice)	id	nombre	precio	stock	categor...	0	1
0	1	'TV 55'	1000	30	'Video'		
1	2	'Laptop'	1500	100	'Comput...		
2	3	'iPhone...'	800	42	'Telefo...		
3	4	'Tablet'	500	71	'Comput...		
4	5	'Pods'	100	28	'Audio'		
5	6	'MP3 pl...	200	11	'Audio'		
6	7	'Videoc...	300	22	'Video'		
7	8	'Smartw...	250	88	'Comput...		
8	9	'Impres...	150	14	'Acceso...		
9	10	'Altavo...'	120	18	'Audio'		
10						{...}	{...}

▶ Array(11)



Funciones de orden superior

```
const productosUnificados = productos.flat(1);

console.table(productosUnificados);
```



De transformación: flat()

Así unificaremos rápidamente en una sola dimensión, un array de elementos de un mismo tipo.

VM1040:1

(índice)	id	nombre	precio	stock	categoria
0	1	'TV 55'	1000	30	'Video'
1	2	'Laptop'	1500	100	'Computación'
2	3	'iPhone 8'	800	42	'Telefonía'
3	4	'Tablet'	500	71	'Computación'
4	5	'Pods'	100	28	'Audio'
5	6	'MP3 playe...'	200	11	'Audio'
6	7	'Videocáma...'	300	22	'Video'
7	8	'Smartwatc...'	250	88	'Computación'
8	9	'Impresora'	150	14	'Accesorios'
9	10	'Altavoces'	120	18	'Audio'
10	11	'Printer H...'	550	7	'Accesorios'
11	12	'Altavoces...'	199	8	'Audio'

▶ Array(12)





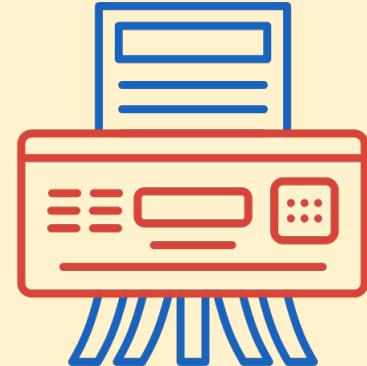
¿Qué es un método destructivo?



En JS, se conoce como método destructivo, o **in-place**, a alguna función o método que, al ejecutarse, altera la estructura original del objeto o array donde éste se aplica.

Este concepto lo tenemos que tener presente porque, de necesitar trabajar siempre con un array original, este método puede alterarlo y romper alguna parte lógica necesaria en nuestra aplicación.

Como alternativa y ante la duda, siempre podemos copiar/clonar un array, previo a ejecutar algún método o función destructiva.



Futuros Métodos de búsqueda y transformación

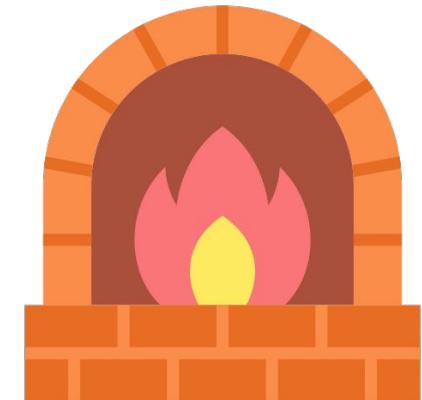
~~.group()~~ (*anteriormente llamado `groupBy()`*)

`.groupToMap()`

`.findLast()`

`.findLastIndex()`

`Object.group()` 





Espacio de prácticas



 Espacio de prácticas 

Trabajaremos con el array de productos utilizando funciones de orden superior:

- El profesor te compartirá el código base del proyecto donde debes trabajar.
 - Utiliza la función de orden superior `.map()` para generar un nuevo array con las siguientes características:
 - Campo nombre, stock, precio (*en dicho orden*)
 - Deberás crear un nuevo mapeo con productos de las categorías: ‘computación’ y ‘audio’
 - Debes agregar tres columnas nuevas: `precio10Up`, `precio25Up`, `precio45Up`
 - Estas columnas deben mostrar el precio de c/producto incrementado un 10%, 25% y 45% respectivamente
 - El nombre de los productos debe visualizarse en mayúsculas. Lo mismo la categoría de estos
- Crea un nuevo array con los productos que posean un stock mayor o igual a 30 unidades
- En ambos casos, visualiza el resultado de estos arrays utilizando `console.table(array)`



Espacio de prácticas



Puesta en común del ejercicio

¡Gracias!

