

# Desarrollo Backend

Bienvenid@s

High Order Functions II

Clase 09



Pon a grabar la clase



# Temario

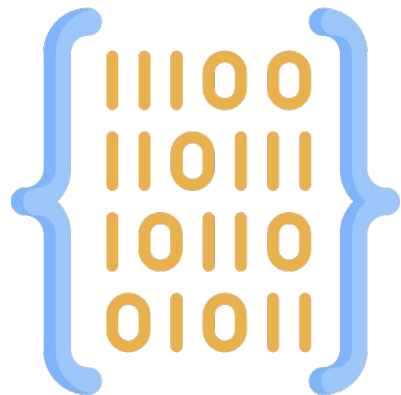
- **High Order Functions 2º parte**
  - el método **.every()**
  - el método **.map()**
  - el método **.reduce()**
  - el método **.sort()**
  - el método **.flat()**
- **Archivos JSON**
  - Estructura
  - Consideraciones
  - Lectura de archivos JSON con Node.js
- **Prácticas con funciones de orden superior**



# High Order Functions - 2ª parte

Son varias las opciones con funciones de orden superior que tenemos disponibles para trabajar con JavaScript.

En este encuentro, veremos algunas funciones adicionales que siguen complementando nuestra tarea como programadores backend, ayudándonos a manipular contenido de arrays de una forma efectiva y simple.



**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

every()

# every()

El método **.every()** se utiliza para comprobar si todos los elementos de un array cumplen una determinada condición.

```
every()

const productos = [
  {id: 1, nombre: 'Notebook 14" FHD', importe: 115000},
  {id: 2, nombre: 'Tablet PAD 9.7"', importe: 195000},
  {id: 3, nombre: 'Macbook Air 13', importe: 745000},
  {id: 4, nombre: 'Tablet DROID 10.1"', importe: 165000}]

const mayor100mil = productos.every(producto => producto.importe > 100_000)
```

# every()

Este método devuelve un valor booleano: **true** si todos los elementos del array cumplen la condición, o **false** si al menos uno de los elementos no cumple la condición.

```
every()

const productos = [
  {id: 1, nombre: 'Notebook 14" FHD', importe: 115000},
  {id: 2, nombre: 'Tablet PAD 9.7"', importe: 195000},
  {id: 3, nombre: 'Macbook Air 13', importe: 745000},
  {id: 4, nombre: 'Tablet DROID 10.1"', importe: 165000}]

const mayor100mil = productos.every(producto => producto.importe > 100_000)
```

# every()

Este método realiza una iteración sobre el total de elementos u objetos del array. Evalúa la expresión definida (*si importe es mayor a 100 mil*), retornando el resultado de la expresión evaluada.



```
every()

productos.every(producto => producto.importe > 100_000)
```

Si, al menos, uno de los elementos no cumple la condición, el valor será **false**.



# every()

En nuestro código de ejemplo, todos los **productos** del array tienen la propiedad **importe** definida con un valor mayor a 100 mil, por lo tanto, el valor final resultante será **true**.

```
every()

const productos = [
  {id: 1, nombre: 'Notebook 14" FHD', importe: 115000},
  {id: 2, nombre: 'Tablet PAD 9.7"', importe: 195000},
  {id: 3, nombre: 'Macbook Air 13', importe: 745000},
  {id: 4, nombre: 'Tablet DROID 10.1"', importe: 165000}]

const mayor100mil = productos.every(producto => producto.importe > 100_000)
```

# every()

Si uno de ellos cambia su valor por un dato menor, entonces hará que el array retorne **false** como valor booleano general.

```
every()

const productos = [
  {id: 1, nombre: 'Notebook 14" FHD', importe: 85000},
  {id: 2, nombre: 'Tablet PAD 9.7"', importe: 195000},
  {id: 3, nombre: 'Macbook Air 13', importe: 745000},
  {id: 4, nombre: 'Tablet DROID 10.1"', importe: 165000}]

const mayor100mil = productos.every(producto => producto.importe > 100_000)
```

# every()

```
every()

const numeros = [2, 4, 6, 8];
const sonPares = numeros.every(numero => numero % 2 === 0)

console.log(sonPares); // true
```

El método **.every()** también funciona con arrays de elementos. En este ejemplo, tenemos un array de números y utilizamos el método `every` para comprobar si todos los números son pares, verificando en el argumento si el número actual es divisible por 2.

map()

# map()

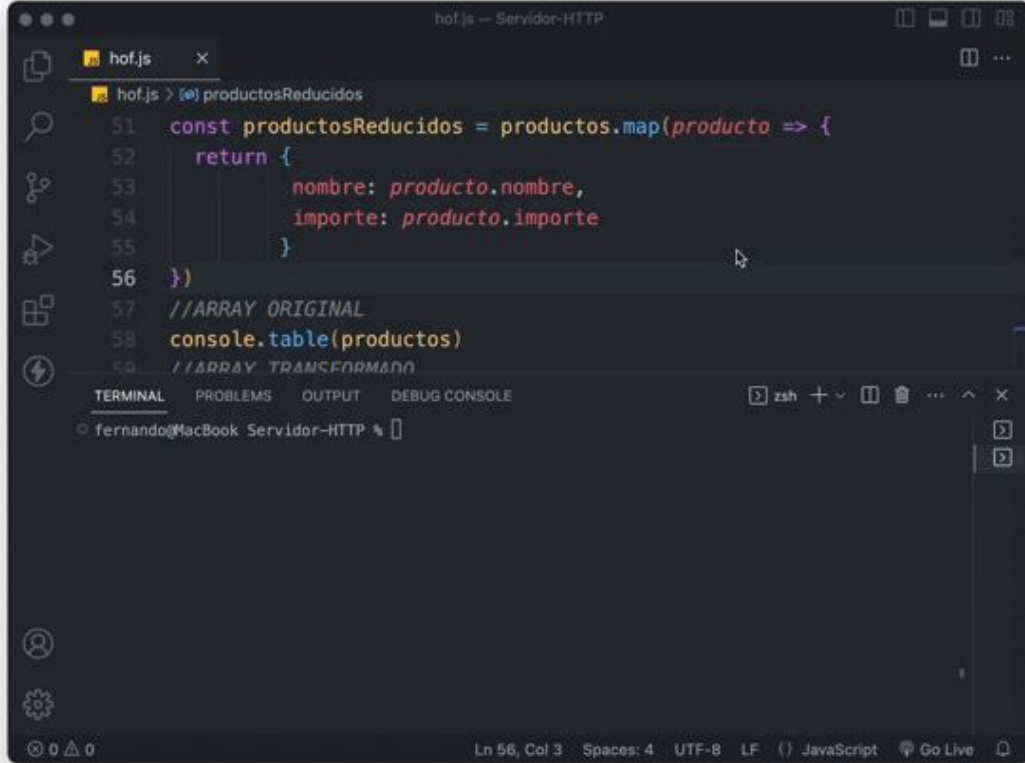
```
map()

const productosReducidos = productos.map(producto => {
  return {
    nombre: producto.nombre,
    importe: producto.importe
  }
})

console.table(productosReducidos)
```

El método **.map()** se utiliza para crear un nuevo array con los resultados definidos sobre cada elemento del array original. Su función es básicamente “*transformar*” cada elemento del array original en un nuevo elemento del array resultante.

# map()



```
hof.js — Servidor-HTTP
hof.js x
hof.js > productosReducidos
51 const productosReducidos = productos.map(producto => {
52   return {
53     nombre: producto.nombre,
54     importe: producto.importe
55   }
56 })
57 //ARRAY ORIGINAL
58 console.table(productos)
59 //ARRAY TRANSFORMADO

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE
fernando@MacBook Servidor-HTTP %
```

De esta forma, podemos trabajar con arrays reducidos sin tener que arrastrar un cúmulo de elementos u objetos innecesarios por diferentes funcionalidades de nuestra aplicación.

De igual forma, simplificamos la forma de entregar resultados, minimizando el trabajo del servidor.

# map()

```
map()

const productosMayusculas = productos.map(producto => {
  return {
    id: producto.id,
    nombre: producto.nombre.toUpperCase(),
    importe: producto.importe,
    categoria: producto.categoria
  }
})

console.table(productosMayusculas)
```

También podemos normalizar determinada información, a partir de la información de un array original.

En el ejemplo, pasamos a mayúsculas el nombre de todos los productos. Esto resolvería parte de la problemática que podemos encontrar al utilizar métodos como **.find()** o **.filter()**, respecto a cómo se ingresa el parámetro de búsqueda.

# map()

```
map()

const productosProyeccion = productos.map(producto => {
  return {
    nombre: producto.nombre,
    importe: producto.importe,
    importe10up: (producto.importe * 1.15).toFixed(2),
    importe10off: (producto.importe * 0.90).toFixed(2)
  }
})

console.table(productosProyeccion)
```

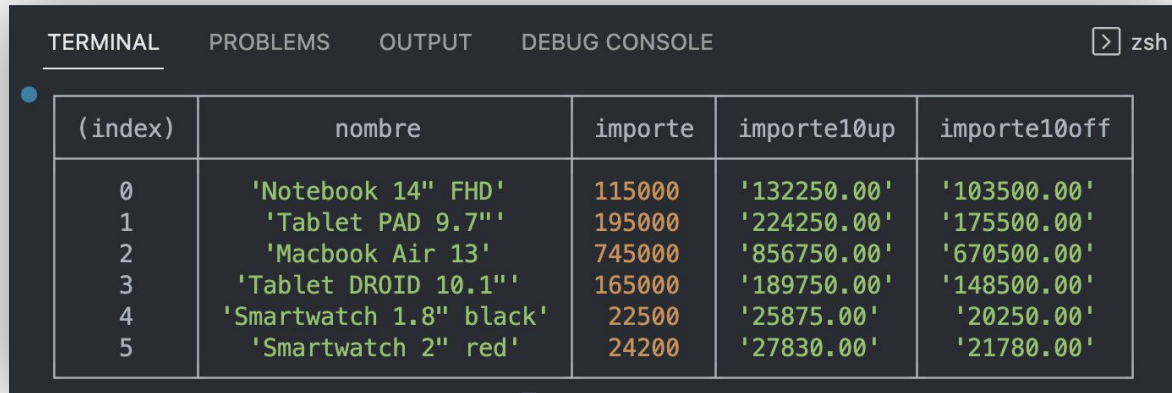
Aquí otro ejemplo de implementación del método **.map()**: necesitamos hacer una proyección de precios de productos, para ver cuánto es su valor con un 15% de incremento y/o con un 10% de descuento.

También es factible hacerlo con **.map()**.



# map()

En este resultado vemos que, **cuando hablamos de transformar**, no solo debemos ajustarnos a respetar la estructura del array original y nada más. Podemos, incluso, crear un nuevo array creando elementos no existentes a partir de uno que sí existe.

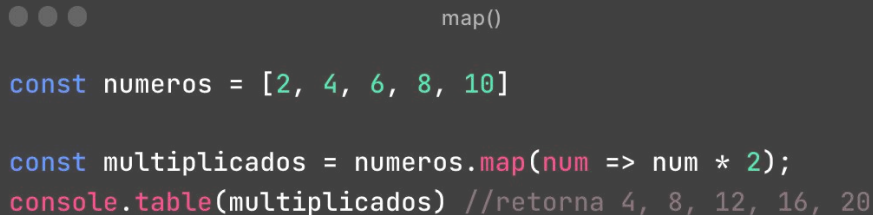


The image shows a terminal window with a dark background. At the top, there are tabs labeled 'TERMINAL', 'PROBLEMS', 'OUTPUT', and 'DEBUG CONSOLE'. The 'TERMINAL' tab is active. In the top right corner of the terminal, there is a 'zsh' prompt. Below the terminal header, a table is displayed. The table has five columns: '(index)', 'nombre', 'importe', 'importe10up', and 'importe10off'. The rows contain data for various electronic devices, including notebooks, tablets, and smartwatches. The 'importe' column shows original prices, while 'importe10up' and 'importe10off' show transformed values (increased and decreased by 10% respectively).

(index)	nombre	importe	importe10up	importe10off
0	'Notebook 14" FHD'	115000	'132250.00'	'103500.00'
1	'Tablet PAD 9.7"'	195000	'224250.00'	'175500.00'
2	'Macbook Air 13'	745000	'856750.00'	'670500.00'
3	'Tablet DR0ID 10.1"'	165000	'189750.00'	'148500.00'
4	'Smartwatch 1.8" black'	22500	'25875.00'	'20250.00'
5	'Smartwatch 2" red'	24200	'27830.00'	'21780.00'

# map()

Por supuesto que **.map()**, al igual que el resto de los métodos, es totalmente utilizable con un array de elementos JS.

A code editor window with a dark background and light text. The title bar at the top shows three window control buttons (red, yellow, green) on the left and the text 'map()' on the right. The code inside is as follows:

```
const numeros = [2, 4, 6, 8, 10]

const multiplicados = numeros.map(num => num * 2);
console.table(multiplicados) //retorna 4, 8, 12, 16, 20
```

# reduce()

# reduce()

```
reduce()

const precioTotal = carrito.reduce((acumulado, producto)=> {
  return acumulado + (carrito.precioUnitario * carrito.cantidad, 0);
});

console.log("Total del carrito = " + precioTotal.toFixed(2))
```

El método **.reduce()** se utiliza para reducir un array a un único valor, aplicando una función acumuladora a cada elemento del array. En otras palabras, se encarga de acumular los valores del array (*o de la propiedad de un objeto del array*), y devolver un resultado final.

# reduce()

```
reduce()

const carrito = [
  {id: 1, nombre: 'Notebook 14" FHD', precioUnitario: 115000, cantidad: 1},
  {id: 4, nombre: 'Tablet DROID 10.1"', precioUnitario: 165000, cantidad: 1},
  {id: 5, nombre: 'Smartwatch 1.8"', precioUnitario: 22500, cantidad: 2}
]
```

En el caso de un array de objetos que representan productos en un carrito, se podría utilizar el método **reduce()** para calcular el precio total del carrito, multiplicando el **precio unitario** por la **cantidad** de cada producto y sumando todos los resultados.

# reduce()

```
reduce()

const precioTotal = carrito.reduce((acumulado, producto) => {
  return acumulado + (carrito.precioUnitario * carrito.cantidad, 0);
});

console.log("Total del carrito = " + precioTotal.toFixed(2))
```

La función acumuladora que se proporciona en este ejemplo, como argumento del método **.reduce()**, recibe dos parámetros: el valor **acumulado** hasta el momento y el **objeto carrito actual**.

En cada iteración, la función multiplica el **precio unitario** de cada producto del carrito por su **cantidad**, y lo suma al valor **acumulado**, devolviendo el nuevo valor acumulado...

# reduce()

```
reduce()

const precioTotal = carrito.reduce((acumulado, producto)=>
    acumulado +
    (producto.precioUnitario * producto.cantidad)
    , 0)

console.log("Total del carrito = " + precioTotal.toFixed(2))
```

... El valor **0** (cero) indicado al final, le indica al parámetro **acumulado**, con qué valor debe comenzar. De esta forma se garantiza que el uso del método **reduce()** inicie su acumulador en cero.

# reduce()

```
reduce()

const saldoAfavor = -10500

const precioTotal = carrito.reduce((acumulado, producto)=>
    acumulado +
    (producto.precioUnitario * producto.cantidad)
    , saldoAfavor)

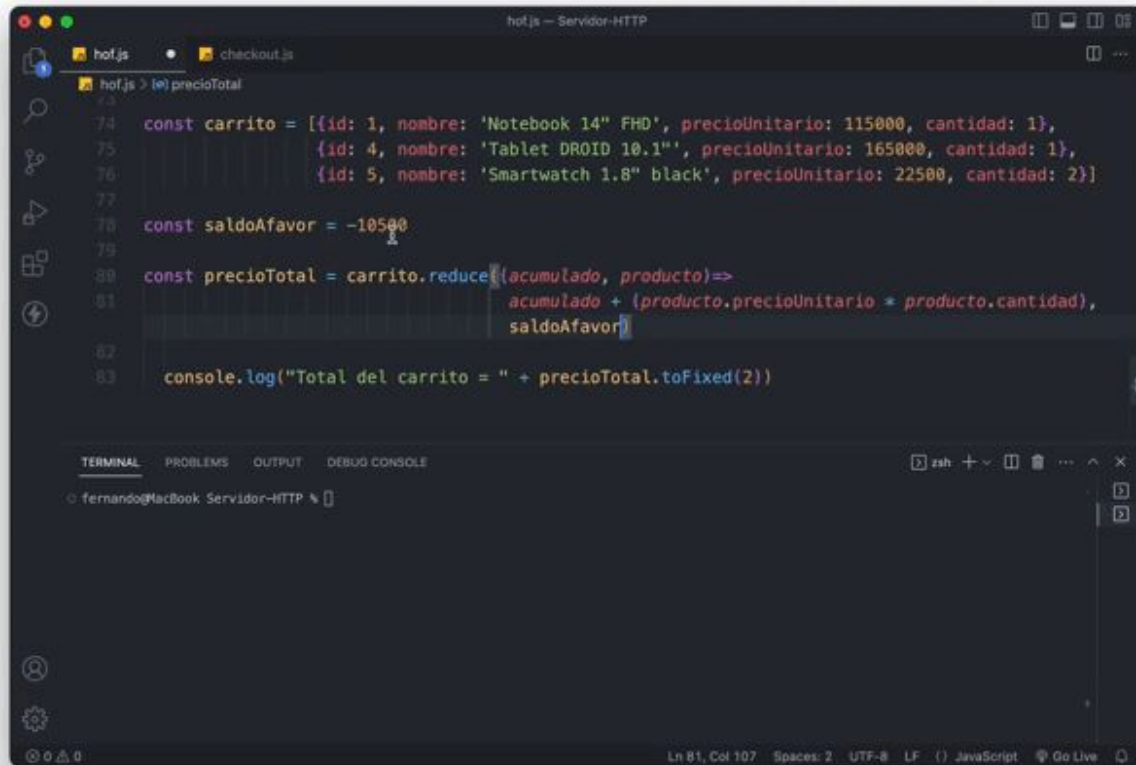
console.log("Total del carrito = " + precioTotal.toFixed(2))
```

El valor **0** (cero) puede ser reemplazado por una variable o constante la cual, contiene un saldo a favor (*debe estar en negativo*).

Por lo tanto, cuando se aplique el cálculo del método, si había un saldo a favor, este será descontado del total del cálculo.



# reduce()



```
hof.js — Servidor-HTTP
hof.js • checkout.js
hof.js > |0| precioTotal
#3
74 const carrito = [{id: 1, nombre: 'Notebook 14" FHD', precioUnitario: 115000, cantidad: 1},
75                  {id: 4, nombre: 'Tablet DROID 10.1"', precioUnitario: 165000, cantidad: 1},
76                  {id: 5, nombre: 'Smartwatch 1.8" black', precioUnitario: 22500, cantidad: 2}]
77
78 const saldoAfavor = -10500
79
80 const precioTotal = carrito.reduce((acumulado, producto)=>
81                                   acumulado + {producto.precioUnitario * producto.cantidad},
82                                   saldoAfavor)
83
84 console.log("Total del carrito = " + precioTotal.toFixed(2))

TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE
fernando@MacBook Servidor-HTTP %
```

Veamos en acción al método

**.reduce()**, aprovechando una variable denominada **saldoAfavor**, la cual posee el valor correspondiente al punto desde donde el parámetro **acumulador** debe comenzar a totalizar.

sort()

# sort()

```
reduce()

const carrito = [
  {id: 1, nombre: 'Notebook 14" FHD', precioUnitario: 115000, cantidad: 1},
  {id: 4, nombre: 'Tablet DROID 10.1"', precioUnitario: 165000, cantidad: 1},
  {id: 5, nombre: 'Smartwatch 1.8"', precioUnitario: 22500, cantidad: 2}
]
```

El método **.sort()** se utiliza para ordenar los elementos de un array de acuerdo a algún criterio específico. El funcionamiento de este método con un array de elementos es directo, o sea, no requiere parámetros dentro del método.

Pero antes de ver cómo ordenar un array de objetos, veamos un poco qué lógica aplica esta función de orden superior.

# sort()

El algoritmo de ordenamiento burbuja es un algoritmo simple que se utiliza para ordenar elementos de un array en orden ascendente o descendente.

El funcionamiento del algoritmo consiste en comparar pares de elementos adyacentes del array, intercambiándolos si no están en el orden correcto. Este proceso se repite varias veces hasta que todos los elementos estén ordenados.

**En la imagen contigua vemos una iteración repetitiva sobre los elementos hasta lograr ordenarlos.**



6 5 3 1 8 7 2 4

# sort()

Por ejemplo, para ordenar un array en orden ascendente, se puede comenzar comparando el primer y segundo elemento. Si el segundo elemento es menor que el primero, se intercambian.

Luego se compara el segundo y tercer elemento, y así sucesivamente hasta llegar al final del array. En este punto, el último elemento del array será el mayor, por lo que ya estará en su posición correcta.

El proceso se repite ahora desde el principio del array hasta el penúltimo elemento, y así sucesivamente hasta que todos los elementos estén ordenados.

6 5 3 1 8 7 2 4

# sort()

Este algoritmo es sencillo y fácil de implementar, pero tiene un **tiempo de ejecución  $O(n^2)$** , lo que **lo hace poco eficiente para arrays grandes**.

Puedes encontrar más ejemplos en la publicación de [Wikipedia](#), donde está explicado con mucho detalle.



**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# sort()

```
sort()

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

Veamos entonces cómo se estructura esta función de orden superior. Como muestra el ejemplo contiguo, la misma recibe dos parámetros; estos suelen denominarse **a** y **b**.

El parámetro **a** recibe como valor el primer objeto del array, mientras que el parámetro **b** recibe el siguiente.

Se aplica la comparación interna y luego, el parámetro **b**, asume en la iteración, el siguiente objeto del array...

# sort()

```
sort()

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

...Internamente, por cada iteración, se define un **if()** que compara si la propiedad elegida para ordenar del parámetro **a**, es menor a la propiedad similar del parámetro **b**.

Si es menor, **retorna el valor -1**, cortando el código de la iteración para pasar al siguiente elemento a comparar.

Sino, a través de otro **if()** compara si la misma propiedad del parámetro **a**, es mayor a la propiedad homónima del parámetro **b**. Si lo es, **retorna el valor 1**.



# sort()

```
sort()

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

Si ninguna de las comparaciones definida en cada **if()** se cumple, entonces **retorna 0**.

Esta comparativa retornando **-1**, **1** ó **0**, se ocupa de definir el ordenamiento de los objetos dentro del array. Invierte su posición de izquierda a derecha, de derecha a izquierda, o los deja donde están.

Esta iteración se hará repetidas veces, hasta terminar de definir el orden de los objetos en el array.

# sort()

```
sort()

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

Hemos elegido la propiedad **nombre** para aplicar un ordenamiento sobre el array modelo denominado **carrito**.

Esta misma lógica es estricta para todos los casos en los cuales necesitemos ordenar de forma ascendente por alguna propiedad específica en un array de objetos.

Lo único que debemos cambiar aquí, es la propiedad sobre la cual elegimos aplicar este ordenamiento.

# sort() inverso

# sort() inverso

```
sort()

const paises = ['Uruguay', 'Brasil', 'Argentina', 'Chile', 'Bolivia'];

paises.reverse() //invierte el orden anterior de los elementos

paises.sort()    //ordena los elementos de manera ascendente

paises.sort().reverse() //ordena los elementos de forma ascendente
                        //luego lo invierte, logrando así el ordenamiento inverso
```

Cuando trabajamos con un array de elementos convencional, JavaScript cuenta con el método **.reverse()** para revertir el orden actual de los elementos, y el método **.sort()** de forma directa, para darle un orden ascendente a los mismos.

Sobre este tipo de array podemos encadenar ambos métodos y así aplicar un ordenamiento inverso sobre los elementos en cuestión.

# sort() inverso

No existe una función de orden superior denominada **.reverse()** para ordenar inversamente un array de objetos, el mismo método **.sort()** nos da la posibilidad de aplicarlo.

Para ello, contamos con dos formas posibles:

- 1) cambiar el orden de los dos primeros return, devolviendo **1** para el primer **if()** y **-1** para el segundo **if()**. Así lograremos nuestro cometido
- 2) Invertir el orden de los operadores de comparación **<** y **>**, en cada bloque **if()**

Cualquiera de estas dos técnicas es válida para invertir el orden ascendente por la propiedad elegida.

```
sort()

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

flat()

# flat()

Nos queda por ver el método **.flat()** aplicado en un array. Este se utiliza en arrays para "*aplanar*" sub-arrays dentro del array principal. En otras palabras, si el array principal contiene uno o más sub-arrays, **.flat()** los extrae y los coloca en una sola dimensión.

Veamos, a continuación, un ejemplo:



# flat()

Tenemos nuestro array **productos**, usado para ejemplificar algunos de los métodos anteriores.

Por otro lado, contamos con un array denominado **nuevosProductos**, y necesitamos incorporar en un único array, el contenido de estos dos.

Allí es donde el método **.flat()** será nuestro aliado, ayudándonos a simplificar esta tarea.

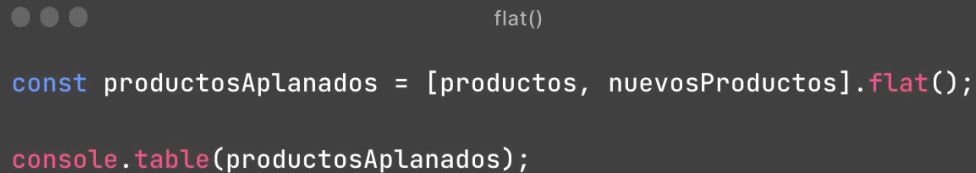
```
sort()

const productos = [
  {id: 1, nombre: 'Notebook 14" FHD', importe: 115000, categoria: 'Portátil'},
  {id: 2, nombre: 'Tablet PAD 9.7"', importe: 195000, categoria: 'Tablet'},
  {id: 3, nombre: 'Macbook Air 13', importe: 745000, categoria: 'Portátil'},
  {id: 4, nombre: 'Tablet DROID 10.1"', importe: 165000, categoria: 'Tablet'},
  {id: 5, nombre: 'Smartwatch 1.8" black', importe: 22500, categoria: 'Relojes'},
  {id: 6, nombre: 'Smartwatch 2" red', importe: 24200, categoria: 'Relojes'}
];

const nuevosProductos = [
  {id: 7, nombre: 'Tablet DROID 7"', importe: 110500, categoria: 'Tablet'},
  {id: 8, nombre: 'Smartwatch 1.5" white', importe: 22500, categoria: 'Relojes'}
];
```



# flat()



```
flat()

const productosAplanados = [productos, nuevosProductos].flat();

console.table(productosAplanados);
```

Debemos entonces combinar ambos arrays por separado, en un tercer array, encerrando los mismos entre corchetes. A través de dichos corchetes ya tenemos la posibilidad de invocar al método **.flat()** quien se ocupará de “*aplanar*”, ambos arrays, en un nuevo array denominado **productosAplanados**.

# flat()

Aquí vemos en acción el uso del método **.flat()**. Este es completamente funcional a los arrays de objetos, tal como vemos en este ejemplo, como también con arrays de elementos.

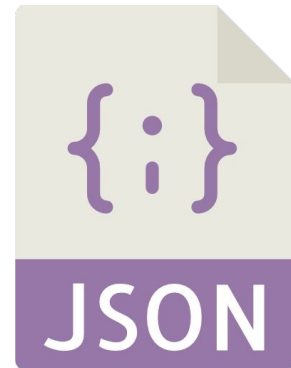
Este mecanismo nos evita tener que iterar el array **nuevosProductos** con el método **.forEach()**, y realizar en cada iteración un **.push()** del elemento actual hacia el array **productos**.

# Archivos JSON

# Archivos JSON

Un archivo JSON es un formato de texto utilizado para almacenar y transmitir datos estructurados.

El nombre JSON es una sigla proveniente de **JavaScript Object Notation**, aunque se utiliza en muchos otros lenguajes de programación además de JavaScript.



# Archivos JSON

Un archivo JSON está formado por una colección de **pares de clave/valor**.

**Los datos se almacenan en un formato similar al de un objeto JavaScript**, con propiedades y valores separados por dos puntos (:), y cada par clave/valor separado por una coma (,).

Los datos también pueden estar anidados, permitiendo la creación de estructuras de datos complejas.



# Archivos JSON

```
Archivos JSON

{
  "nombre": "Juan",
  "apellido": "Pérez",
  "edad": 35,
  "direccion": {
    "calle": "Calle Falsa",
    "numero": 123,
    "ciudad": "Ciudad Ficticia"
  },
  "telefonos": [
    {
      "tipo": "celular",
      "numero": "555-1234"
    },
    {
      "tipo": "trabajo",
      "numero": "555-5678"
    }
  ]
}
```

Aquí tenemos un ejemplo de un archivo JSON, en un formato algo más complejo que el de los array de objetos que venimos trabajando.

Como podemos ver, la información almacenada en la propiedad **telefonos** contiene a su vez un array interno.

**¿Encuentras alguna diferencia más respecto a un array de objetos JavaScript?**

**Miremos el siguiente ejemplo...**

# Archivos JSON

## Array de Objetos

```
Array de objetos JS

const productos = [
  {id: 1, nombre: 'Notebook 14" FHD', importe: 115000, categoria: 'Portátil'},
  {id: 2, nombre: 'Tablet PAD 9.7"', importe: 195000, categoria: 'Tablet'},
  {id: 3, nombre: 'Macbook Air 13', importe: 745000, categoria: 'Portátil'},
  {id: 4, nombre: 'Tablet DROID 10.1"', importe: 165000, categoria: 'Tablet'},
  {id: 5, nombre: 'Smartwatch 1.8" black', importe: 22500, categoria: 'Relojes'},
  {id: 6, nombre: 'Smartwatch 2" red', importe: 24200, categoria: 'Relojes'}
];
```

## Archivo JSON

```
{.} productos.json X
{.} productos.json > ...
1  [
2    {"id": 1, "nombre": "Notebook 14 FHD", "importe": 115000, "categoria": "Portátil"},
3    {"id": 2, "nombre": "Tablet PAD 9.7", "importe": 195000, "categoria": "Tablet"},
4    {"id": 3, "nombre": "Macbook Air 13", "importe": 745000, "categoria": "Portátil"},
5    {"id": 4, "nombre": "Tablet DROID 10.1", "importe": 165000, "categoria": "Tablet"},
6    {"id": 5, "nombre": "Smartwatch 1.8 black", "importe": 22500, "categoria": "Relojes"},
7    {"id": 6, "nombre": "Smartwatch 2 red", "importe": 24200, "categoria": "Relojes"}
8  ]
```

**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Archivos JSON

## Archivo JSON

```
{ } productos.json ×
{ } productos.json > ...
1  [
2    { "id": 1, "nombre": "Notebook 14 FHD", "importe": 115000, "categoria": "Portátil",
3      { "id": 2, "nombre": "Tablet PAD 9.7", "importe": 195000, "categoria": "Tablet",
4        { "id": 3, "nombre": "Macbook Air 13", "importe": 745000, "categoria": "Portátil" },
5        { "id": 4, "nombre": "Tablet DROID 10.1", "importe": 165000, "categoria": "Tablet"},
6        { "id": 5, "nombre": "Smartwatch 1.8 black", "importe": 22500, "categoria": "Relojes"},
7        { "id": 6, "nombre": "Smartwatch 2 red", "importe": 24200, "categoria": "Relojes"}
8    ]
```

En el formato JSON, se utiliza la misma estructura de objetos y arrays que en JavaScript, pero con la diferencia de que las claves y los valores se escriben entre comillas dobles y que no se permite el uso de comillas simples.

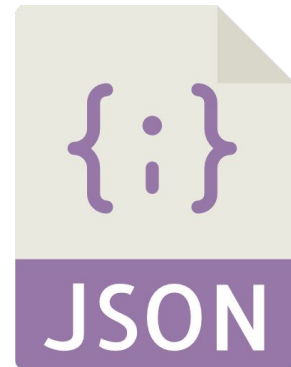
Además, el último elemento de un objeto o array no debe tener una coma después de él.



# Archivos JSON

## ¿Para qué suelen utilizarse estos?

Los archivos JSON son ampliamente utilizados en aplicaciones web para intercambiar datos entre el servidor y el cliente, ya que son fáciles de leer y escribir, y son compatibles con la mayoría de los lenguajes de programación.



**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Archivos JSON

## ¿Podemos editar archivos JSON desde Node.js?

Es posible su edición, utilizando el **módulo FileSystem de Node.js**. Debemos trabajarlo combinando su lectura, conversión al formato de array de objetos, y luego de agregar o modificar sus elementos, volver a escribir el contenido actualizado en el archivo en cuestión.



**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Archivos JSON

Aquí te compartimos un ejemplo de cómo leer un archivo JSON, convertirlo al formato de array de objetos JavaScript, y leer su contenido.

```
Archivo JSON

const fs = require('fs');

// Leer archivo JSON
const data = fs.readFileSync('productos.json', 'utf-8');

// Convertir a objeto JavaScript
const productos = JSON.parse(data);

// Utilizar los datos
productos.forEach(producto => {
  console.table(producto);
});
```

# Archivos JSON

¿Notas alguna complicación o impedimento para utilizar el formato de archivo JSON como si fuese una base de datos?



**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Archivos JSON

**Es importante tener en cuenta que esta forma de utilizar un archivo JSON como base de datos en una aplicación Node.js puede no ser adecuada para aplicaciones de producción con grandes cantidades de datos y múltiples usuarios.**

**En ese caso, se recomienda utilizar una base de datos real como MongoDB, MySQL o PostgreSQL.**



**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Funciones de orden superior



**El poder que nos dan las funciones de orden superior es muy diverso. Nos permiten manipular y/o transformar los datos, de acuerdo a nuestra necesidad, y son verdaderamente útiles para trabajar todo desde JS, de forma rápida y precisa.**

**Esto genera, en proyectos de alta concurrencia de usuarios, un alivio notable en los tiempos de interacción con diversas bases de datos, permitiendo que Node.js moldee la información desde su lógica.**

# Espacio de trabajo

# Espacio de trabajo

Practiquemos la implementación de algunas de estas funciones de orden superior en un proyecto Node.js.

**Tiempo estimado: 20 minutos.** 



```
const questions = [ 'dudas', 'consultas', '🤔' ]
```



```
> node gracias.js
```