



**Argentina
programa
4.0**



Ministerio de Economía
Argentina

Secretaría de
Economía del Conocimiento

*primero
la gente*

Clase 7: Funciones de orden superior

Agenda de hoy

- A. Repaso sobre array de elementos y objetos
 - a. Métodos comunes
- B. Introducción a las Funciones de Orden Superior
 - a. Qué son
 - b. Ejemplos de implementación
- C. Métodos avanzados en Arrays
 - a. el método `.forEach()`
 - b. el método `.find()`
 - c. el método `.filter()`
 - d. el método `.some()`
 - e. Combinando métodos
 - f. el método `.map()`
 - g. el método `.reduce()`
 - h. el método `.sort()`



High Order Functions

Ya juntamos el conocimiento necesario sobre servidores web, el framework Express y el manejo de rutas. Llegó el momento de integrar los arrays junto a estas tecnologías para sacar provecho de todo el potencial de las aplicaciones backend.

Es importante que domines bien, tanto los arrays de elementos como de objetos. Principalmente porque, **estos últimos, conforman la estructura elegida** por las aplicaciones de backend **para intercambiar información con otras aplicaciones.**

Pasemos a refrescar conceptos:

Repaso fugaz sobre array de elementos y objetos

High Order Functions

Aquí, un array de elementos.

```
Arrays  
  
const paises = ['Argentina', 'Bolivia', 'Chile', 'Paraguay', 'Uruguay']
```

Por aquí, un array de objetos, también llamado array de objetos JSON.

```
Arrays  
  
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},]
```

Métodos comunes

Ambos utilizan los métodos convencionales de inserción y eliminación de elementos u objetos. Si por ejemplo necesitamos incrementar su contenido, le pasamos como parámetro un elemento, o un objeto a insertar.



Arrays

```
paises.push('Brasil')
```

```
productos.push({id: 5, nombre: 'Smartwatch 1.8"', importe: 22000})
```

Métodos comunes

Para eliminar elementos, recurrimos al método **.pop()** que elimina el último, o al método **.shift()** que elimina el primer elemento del array.

Para eliminar un elemento intermedio, debemos primero identificar su posición (*índice*), y luego utilizar el método **.splice()** para removerlo.

```
Arrays

//último elemento del array
países.pop()
productos.pop()

//primer elemento del array
países.shift()
productos.shift()
```


Métodos comunes

Trabajar la quita de elementos específicos es fácil cuando trabajamos con un array de elementos pero, al trabajar con un array de objetos, la cuestión es ligeramente diferente.

Para subsanar esto último, existen métodos que parten de lo que conocemos como funciones de orden superior.

```
Arrays

//ubicar el índice de un elemento
const idx = paises.findIndex('Brasil')
if (idx > -1) {
  paises.splice(idx, 1)
}

//ubicar el índice en array de objetos
const idx = productos.findIndex(¿😨?)
productos.shift(idx, 1)
```

Introducción a las Funciones de Orden Superior

Introducción a las Funciones de Orden Superior

Las **High Order Functions**, también conocidas como **funciones de orden superior**, son funciones JavaScript que aceptan una o más funciones como argumentos y/o pueden devolver una función como resultado de una operación o tarea.

Su principal metier es operar de forma fácil y abstracta sobre otras funciones, tratando a éstas últimas como datos, o transformando su estructura básica.



Introducción a las Funciones de Orden Superior

Esto significa que, las **High Order Functions**, hacen que el código sea mucho más modular y reutilizable, ya que las funciones en cuestión pueden combinarse y anidarse para producir un comportamiento más complejo.



Introducción a las Funciones de Orden Superior

Antes de pasar a conocer las funciones de orden superior que integran el lenguaje JavaScript y que son ideales para el manejo de arrays, veamos cómo podemos crear una función de orden superior a partir de nuestra propia necesidad.

Para ello, abordemos un pequeño ejemplo.



Introducción a las Funciones de Orden Superior

Creamos una función llamada **realizarCalculo()**. La misma recibirá tres parámetros: dos numéricos y un tercer parámetro denominado **salida**.

Luego, llamamos a la función para ver su resultado...

```
High Order Functions

function realizarCalculo(numA, numB, salida) {
  salida(numA * numB)
}

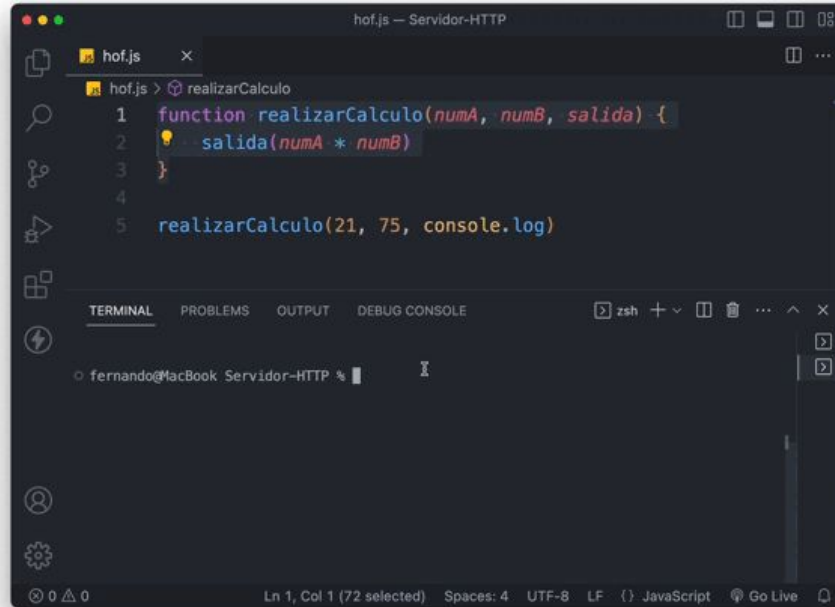
realizarCalculo(21, 75, console.log)
```

Introducción a las Funciones de Orden Superior

Los primeros parámetros, que son de entrada, reciben números, y el tercer parámetro, **salida**, recibe un objeto.

En este caso, es el objeto **console** y su método **.log()**.

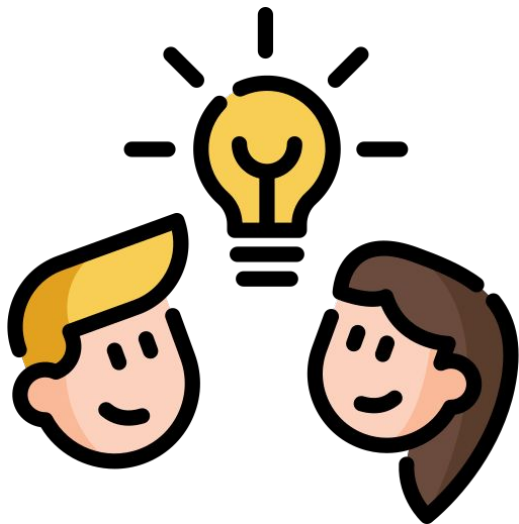
Internamente, en la función, multiplicamos los números recibidos y los mostramos como resultado utilizando la función u objeto de salida.



```
hof.js — Servidor-HTTP
hof.js x
hof.js > realizarCalculo
1 function realizarCalculo(numA, numB, salida) {
2   salida(numA * numB)
3 }
4
5 realizarCalculo(21, 75, console.log)

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

Introducción a las Funciones de Orden Superior



En definitiva, lo que logramos con las funciones de orden superior, como **realizarCalculo()**, es poder llevar adelante una tarea específica, abstrayéndonos de qué ocurre dentro de esta función en particular.

Y, la lógica de una función de orden superior es, justamente, lograr realizar una tarea y que esta se resuelva correctamente sin que tengamos que conocer técnicamente cómo se resuelve.

Métodos avanzados en arrays

Métodos avanzados en arrays

Veamos entonces cómo los arrays, tanto de elementos como de objetos, incluyen una serie de métodos, definidos como funciones de orden superior.

Estos **reciben una función como parámetro**, y nos **retornan un resultado ya procesado** sobre los datos del array en cuestión.

Métodos avanzados en arrays

Los métodos en cuestión, son:

- `forEach()`
- `find()`
- `filter()`
- `some()`
- `map()`
- `reduce()`
- `sort()`



Cada una de estas funciones, o métodos, acepta una o más funciones como argumentos y/o devuelve una función como resultado.

Métodos avanzados en arrays

Método	Descripción
forEach()	Se utiliza para iterar sobre cada elemento de un array y ejecutar una función proporcionada para cada uno de ellos.
find()	Se utiliza para buscar un elemento en un array que cumpla con una determinada condición.
filter()	Se utiliza para filtrar los elementos de un array que cumplan con una determinada condición.
some()	Se utiliza para comprobar si al menos un elemento de un array cumple con una determinada condición.
map()	Se utiliza para mapear, transformar, o convertir un array existente en un nuevo array de objetos.
reduce()	Toma un conjunto de datos numéricos de un array, y los reduce devolviendo un único resultado.
sort()	Ordena los objetos de un array, a través de una propiedad específica.

Métodos avanzados en arrays

Profundicemos cada uno de ellos para ver cómo usarlos en casos aplicados.

Trabajaremos sobre el siguiente modelo de array.

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

forEach()

forEach()

El método `forEach()` itera sobre el array en cuestión, recorriendo el mismo desde el principio hasta el fin.

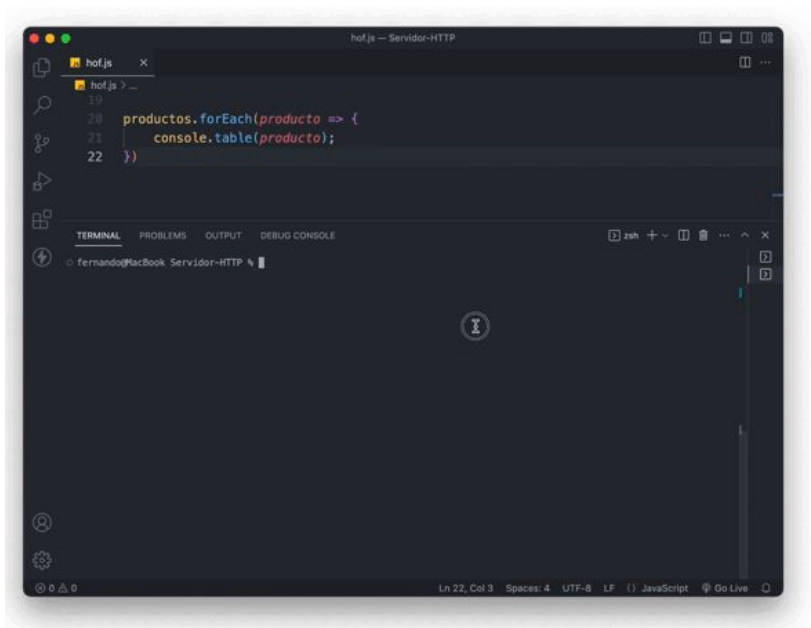
El método recibe un parámetro, denominado predicado el cual, por cada iteración, copia el elemento en el cual se está iterando.

Dentro de esta iteración podemos definir que se ejecute el código que consideremos necesario.

```
High Order Functions

productos.forEach(producto => {
  console.table(producto);
})
```

forEach()



En este ejemplo, ejecutamos `console.table()` para ver en formato tabla, cada uno de los elementos del array iterados.

Si bien el ciclo **for convencional** o el ciclo **for...of** nos permiten realizar la misma tarea, **forEach()** define en el predicado, el elemento en el cual se está iterando.

Además, está optimizado, lo cual lo hace mucho más veloz si debemos iterar decenas o centenas de elementos de un array.



forEach()

```
High Order Functions

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

paises.forEach(pais => {
  console.log(pais);
})
```

Funciona por igual con un array de elementos.

Puede que sea más apropiado su uso dentro de una aplicación frontend que en un backend, pero también lo tenemos disponible por si podemos sacar provecho de éste.

find()

find()

El método **find()** itera sobre el array en búsqueda de un elemento u objeto en particular.

En el **predicado** que recibe como parámetro, se asienta cada uno de los elementos u objetos que vamos iterando. Allí podemos aplicar una comparación con el elemento, o con la propiedad del objeto, en búsqueda de encontrar el primer elemento del array coincidente... =>

```
High Order Functions

const resultado = productos.find(producto => producto.id = 5)

if (resultado !== 'undefined') {
  console.table(resultado)
}
```



find()

... este método retorna un resultado al momento de encontrar la primera coincidencia, y dejar de iterar el array. Por ello, debemos definir una constante en la cual se asienta lo que el método find() nos retorna como resultado.

Ante la primera coincidencia que se encuentra, el método find() **devuelve el objeto o elemento como resultado** (según el tipo de array), si no, devuelve **undefined**.

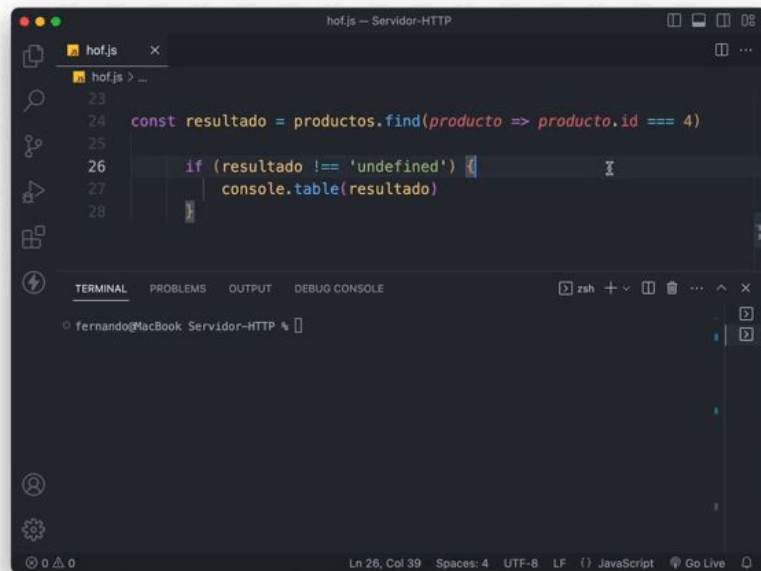
```
High Order Functions

const resultado = productos.find(producto => producto.id = 5)

if (resultado !== 'undefined') {
  console.table(resultado)
}
```



find()



```
23
24 const resultado = productos.find(producto => producto.id === 4)
25
26 if (resultado !== 'undefined')
27   console.table(resultado)
28
```

Veamos un ejemplo con los dos posibles escenarios:

- 1) Cuando el resultado es válido, el método `find()` encontró un elemento coincidente y, el mismo, se asigna a la constante **resultado**.
- 1) Caso contrario, la constante recibe el valor **'undefined'**.



find()

Es clave que, luego de realizar una búsqueda con el método `find()`, **controlemos el resultado que nos devuelve** la misma. De esta forma tendremos un control sobre el mismo.

Si el resultado es válido, lo utilizamos en alguna otra operación. Si no, mostramos un mensaje de error “*amigable para el usuario*”.

```
High Order Functions

const resultado = productos.find(producto => producto.id = 150)

if (resultado !== 'undefined') {
  console.table(resultado)
}
```

find()

```
High Order Functions

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

const resultado = paises.find(pais => pais === 'Argentina');

if (resultado !== 'undefined') {
  console.log(resultado);
}
```

Y, por aquí, un ejemplo del método `find()` trabajando sobre un array de elementos.

filter()

filter()

filter() se comporta de forma similar a **find()**, con la diferencia de que filtra todos los elementos coincidentes del array.

Recorre el array de principio a fin, capturando aquellos elementos u objetos donde encuentra coincidencia. Finalmente, los devuelve como resultado en un nuevo array.

```
High Order Functions

const resultado = productos.filter(producto => producto.categoria === 'Tablet')

if (resultado !== []) {
  console.table(resultado)
}
```

filter()

Si no encuentra coincidencia, devuelve como resultado un array vacío.

Debemos tener la precaución de evaluar qué resultado nos devuelva este método, previo a trabajar con el array resultante.

Si el array está vacío, podremos mostrar también un mensaje de error “*amigable para el usuario*”.

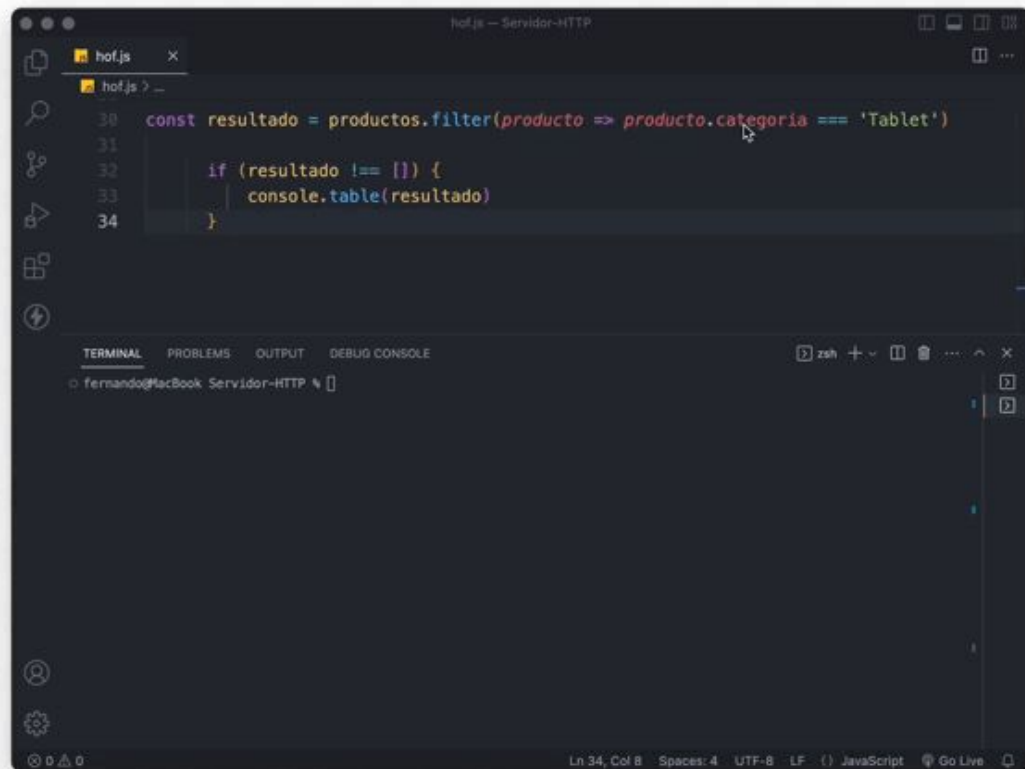
```
High Order Functions

const resultado = productos.filter(producto => producto.categoria === 'Tablet')

if (resultado !== []) {
  console.table(resultado)
}
```

filter()

El método `filter()` en acción: la propiedad **.length** del array resultante nos puede validar si se encontraron o no coincidencias en el filtro aplicado, para luego operar con los datos, de la forma más efectiva.



```
hof.js
hof.js > ...

30 const resultado = productos.filter(producto => producto.categoria === 'Tablet')
31
32 if (resultado !== []) {
33   console.table(resultado)
34 }
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

fernando@MacBook: Servidor-HTTP %

Ln 34, Col 8 Spaces: 4 UTF-8 LF JavaScript Go Live

Combinando métodos

*¿Y si quiero filtrar por una palabra específica
de un nombre de producto compuesto? 🤖*

Combinando métodos

Aquí es donde comienza la magia de JavaScript: si el elemento o la propiedad es un tipo de dato *string*, entonces el mismo dispone del método **.includes('valor')**. Este método nos permite definir un texto o parámetro parcial, para así poder aplicar un filtro que no tenga exactitud.

Mira el ejemplo a continuación:



High Order Functions

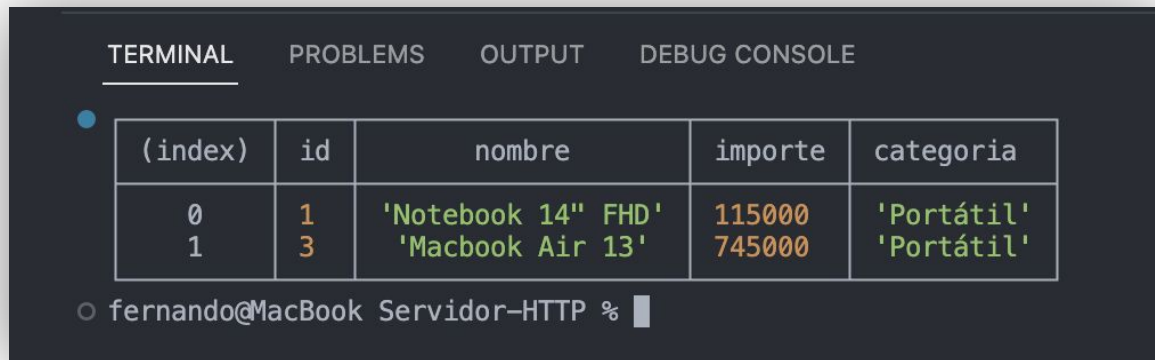
```
const resultado = productos.filter(producto => producto.nombre.includes('book'))

if (resultado !== []) {
  console.table(resultado)
}
```



Combinando métodos

En nuestro array modelo disponemos de dos productos los cuales tienen la palabra “*Macbook*” y “*Notebook*”, respectivamente. El método **.filter()** recibe en el **predicado** parte de una palabra, y termina aplicando el filtro sobre ese valor.



The image shows a terminal window with a dark background. At the top, there are four tabs: 'TERMINAL' (selected), 'PROBLEMS', 'OUTPUT', and 'DEBUG CONSOLE'. Below the tabs, there is a table with 5 columns: '(index)', 'id', 'nombre', 'importe', and 'categoria'. The table contains two rows of data. The first row has index 0, id 1, nombre 'Notebook 14" FHD', importe 115000, and categoria 'Portátil'. The second row has index 1, id 3, nombre 'Macbook Air 13', importe 745000, and categoria 'Portátil'. Below the table, there is a prompt 'fernando@MacBook Servidor-HTTP %' followed by a cursor.

(index)	id	nombre	importe	categoria
0	1	'Notebook 14" FHD'	115000	'Portátil'
1	3	'Macbook Air 13'	745000	'Portátil'

fernando@MacBook Servidor-HTTP %

Combinando métodos

¿Y si el término que usamos como
parámetro de búsqueda viene en
mayúsculas? 😞

Combinando métodos

También podemos sortear este obstáculo, guardando en una variable el parámetro en cuestión aplicando el método **.toLowerCase()** sobre está, para transformar su valor a minúsculas.

Luego, aprovechamos el encadenamiento de métodos que tiene JavaScript para hacer lo mismo con la propiedad sobre la cual hacemos la comparación.

```
High Order Functions

const parametro = 'BOOK'
const resultado = productos.filter(producto => {
  return producto.nombre.toLowerCase().includes(parametro.toLowerCase())
})

if (resultado !== []) {
  console.table(resultado)
}
```



some()

some()



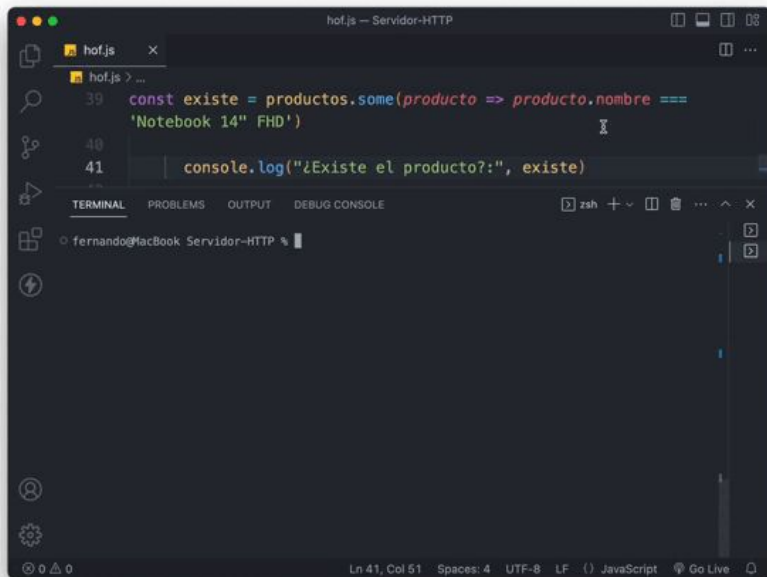
High Order Functions

```
const existe = productos.some(producto => producto.nombre === 'Notebook 14" FHD')  
  
console.log("¿Existe el producto?", existe)
```

Este método permite comprobar si, al menos, un elemento de un array cumple una determinada condición. Retorna un valor **true** si, al menos, un elemento cumple la condición, o **false** si ninguno la cumple.

Si hay varios elementos que cumplen la condición, `some()` sólo devuelve **true** para el primer elemento que la cumpla sin comprobar los demás elementos.

some()



```
hof.js
39 const existe = productos.some(producto => producto.nombre ===
40   'Notebook 14" FHD')
41   console.log("¿Existe el producto?:", existe)
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

fernando@MacBook: Servidor-HTTP

Su aplicación es efectiva para, por ejemplo, validar si un producto existe previo a darlo de alta. Así nos aseguramos de no generar duplicados en la información que se carga.

En aplicaciones de backend, es fundamental este tipo de controles sobre información previo a almacenarse.

map()

map()

```
map()

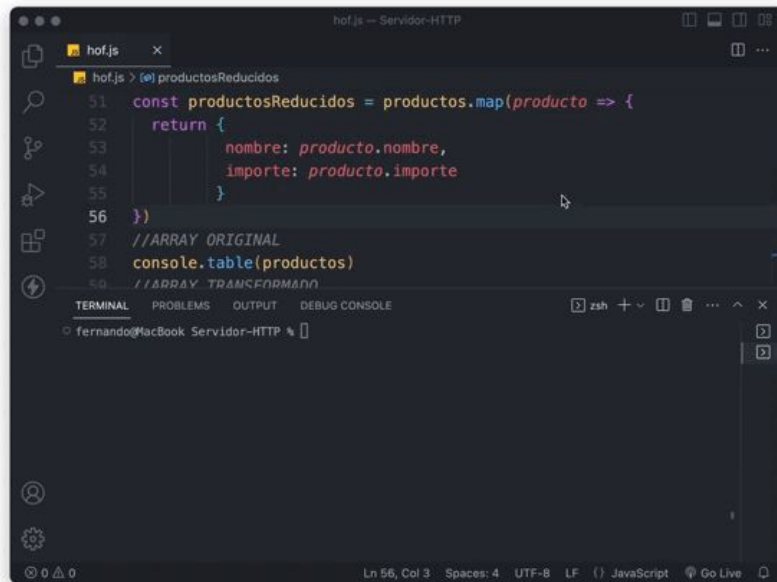
const productosReducidos = productos.map(producto => {
  return {
    nombre: producto.nombre,
    importe: producto.importe
  }
})

console.table(productosReducidos)
```

.map() se utiliza para crear un nuevo array con los resultados definidos sobre cada elemento del array original.

Su función es “*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 %
Ln 56, Col 3 Spaces: 4 UTF-8 LF JavaScript Go Live
```

Podemos trabajar con arrays reducidos sin tener que arrastrar un cúmulo de elementos u objetos innecesarios por diferentes funcionalidades de nuestra aplicación y reduciendo, a su vez, 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 como, por ejemplo, pasando a mayúsculas el nombre de todos los productos.

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)
```

Incluso, si necesitamos hacer una proyección de precios para ver cuánto es el valor de estos con un 15% de incremento y/o con un 10% de descuento, también podemos recurrir a map().

map()

Cuando hablamos de transformar, no solo debemos ajustarnos a respetar la estructura del array original. Podemos, incluso, crear un nuevo array con elementos no existentes a partir de uno que sí existe.

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE					zsh	
(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'		

reduce()

reduce()

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 de la propiedad de un objeto del array, y devolver un resultado final.**



reduce()

```
const precioTotal = carrito.reduce((acumulado, producto)=> {  
  return acumulado + (carrito.precioUnitario * carrito.cantidad, 0);  
});  
  
console.log("Total del carrito = " + precioTotal.toFixed(2))
```



reduce()

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. En este caso, multiplicando el **precio unitario por la cantidad de cada producto y sumando todos los resultados**.

```
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}
]
```

reduce()

La función 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()

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

... 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()

```
const precioTotal = carrito.reduce((acumulado, producto)=> {  
  return acumulado + (carrito.precioUnitario * carrito.cantidad, 0);  
});  
  
console.log("Total del carrito = " + precioTotal.toFixed(2))
```



reduce()

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.

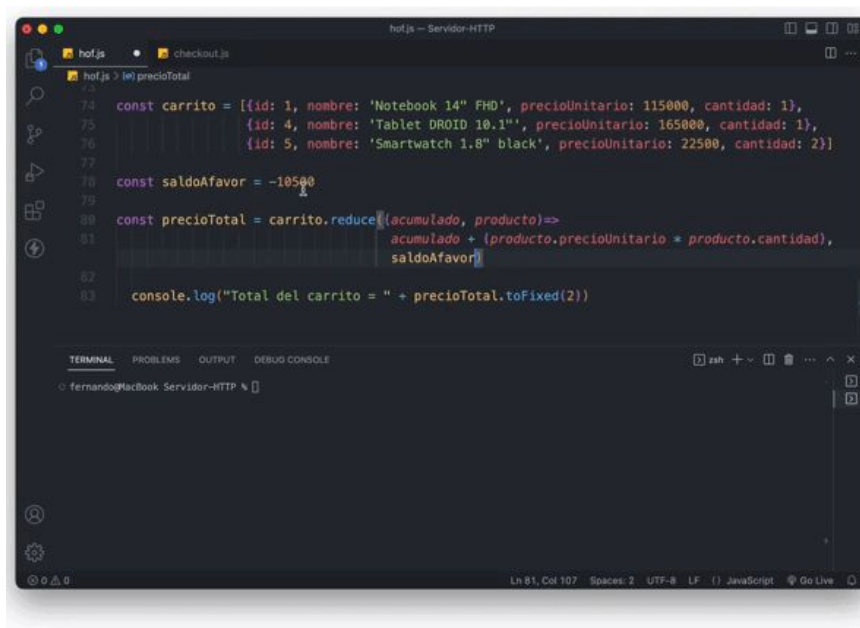
```
reduce()

const saldoAfavor = -10500

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

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


reduce()



```
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))
```

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 no requiere parámetros dentro del método. Pero, **antes de ver cómo ordenar un array de objetos, conozcamos qué lógica se encuentra en el “*detrás de escena*” de la función sort()**.

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()

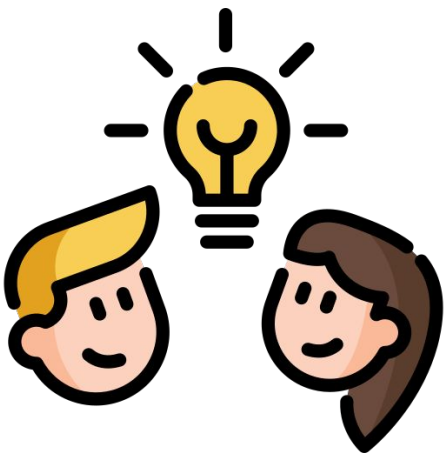
Por ejemplo, para ordenar un array en orden ascendente, se puede comenzar comparando el primer y segundo elemento. Si el segundo elemento es menor que el primero, se intercambian.

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 [esta publicación](#), donde está explicado con mucho detalle.

sort()

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()

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



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.

La iteración se hace 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 común para todos los casos en los cuales necesitemos ordenar de forma ascendente por alguna propiedad específica del array de objetos.

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

sort()

Este mismo método nos da la posibilidad de ordenar de manera invertida los objetos de un array. 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()
- 2) o 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
})
```

Prácticas

Prácticas

Utilizaremos un proyecto con Express JS que nos permita construir un backend que sirva datos mediante diferentes endpoints, a partir de un array de objetos. Éste contendrá las siguientes rutas:

- /productos
- /productos/:id
- /productos/:nombre

Dispondremos de un array de productos, que tendrá estas características:

id - nombre - importe - stock



Prácticas

- **/productos**

- Al peticionar esta ruta, debe devolver el listado de productos completo, en formato JSON. Los mismos deben estar ordenados por nombre.

- **/productos/:id**

- Al peticionar esta ruta, debe devolver un solo producto, buscando el mismo por el ID. Si no lo encuentra, debe devolver un mensaje de error en formato JSON.

- **/productos/:nombre**

- Al peticionar esta ruta, debe devolver una respuesta buscando el producto por el nombre. Puede recibir el nombre completo, o parte de este. Si encuentra más de un resultado, debe devolver un array con todas las coincidencias. Si no encuentra coincidencias, debe devolver un mensaje de error en formato JSON.



Muchas gracias.



Ministerio de Economía
Argentina

Secretaría de
Economía del Conocimiento

*primero
la gente*