
# Practica 04-07 JS Métodos Arrays para Programación funcional con funciones callback.

| Nombre | | Curso | |
|---|---|---|---|
| Apellidos | | Fecha | |

Conozcamos las funciones más importantes para programar sobre arrays

## Sumario

Practica 04-07 JS Métodos Arrays para Programación funcional con funciones callback.....1

a) Categoria: Creación.....4
0. Metodos Estaticos Array.from(fcallback) Array.of() constructor Array.....4

b) Categoria: Acceso.....4
4-5. Array.at() Array[] Array.with(índice, valor).....4
6-7 .Array.unshift(...Args )Array.shift() Array.push(...Args ) Array.pop().....5

c) Categoria: Modificar Array.....6
8. Array.fill(valor, inicio, fin ).....6
9. Array.concat(Array).....6
10. Array.copyWithin(target, start, end).....6
11. Array.entries() Array.keys() Array.values().....7
12. Array.slice(inicio [, fin]]).....8
13. Array.sort([compareFunction]) Array.toSorted().....11
14. Array.reverse() Array.toReverse().....14

d) Categoria: Booleanos.....15
15 Array.includes(elemento,fromIndex).....15
16 ESTATICOS Array.isArray ().....16
17. Array.every(funcioncallback(element, index, array ),Argumentos ).....16
6. Array.some(funcioncallback(element, index, array ),Argumentos).....16

d) Categoria: Buscar.....19
19. Array.find(funcioncallback(currentValue, index, array )).....19
20. Array.findLast( funcioncallback(currentValue, index, array )).....19
21. Array.findIndex( funcioncallback(currentValue, index, array ) ).....20
22. Array.findLastIndex(funcioncallback(currentValue, index, array )).....20
23-24. Array.lastIndexOf(searchElement, fromIndex) Array.IndexOf().....20

e) Categoria: filtros.....22


25. Array.filter(funcioncallback(currentValue, index, array ),Argumentos).....22
26. Array.flat(arrayMultidim).....23
27 .Array.flatMap(funcioncallback(currentValue, index, array )).....23
28.Array.forEach(funcioncallback(currentValue, index, array )).....23
29.Array.map(funcioncallback(currentValue, index, array )).....24

e) Categoria: Reducciones.....28
20. Array.reduce(f.Reductora(acumulador,elemento,Índice ,Array ),valorInicial ).....28
Array.reduceRight(f.Reductora(acumulador,elemento,Índice ,Array ),valorInicial ).....28

e) Categoria: Cadenas-Strings.....31
28. Array.join() Array.toString() Array.toLocaleString().....31


| Categoria | Metodo | Sintasix | Salida |
|---|---|---|---|
| Creacion | **Estatico Array.from** | Estatico Array.from( objeto Iterable) | Array nuevo apartir del objeto |
| Creacion | **Estatico Array.of** | Estatico Array.of( ...elementos) | Array nuevo apartir de los elementos |
| Creacion | **new Array** | new Array(...elementos) | Array nuevo apartir de los elementos |
| acceso | **with** | with(indice, valor) Array[indice] = valor | Array modificado con valor en indice |
| acceso | **at** | at(indice) =Array[indice] | elemento en indice |
| acceso | **shift** | shift() pop() | elemento sacado |
| acceso | **unshift** | unshift( Args ) push(...Args ) | Array con elemento |
| Array modificado | **fill** | fill(valor, inicio, fin ) | Array relleno |
| Array modificado | **concat** | concat(Array) | Array concatenado |
| Array modificado | **copyWithin** | copyWithin(target, start, end) | Array relleno |
| Array modificado | **entries** | entries() keys() values() | **Array Iterator** |
| Array modificado | **slice** | slice(inicio [, fin]]) | SubArray |
| Array modificado | **sort** | sort([compareFunction]) toSorted() | Array Ordenado |
| Array modificado | **reverse** | reverse() toReverse() | Array invertido |
| booleanos | **includes** | includes(elemento,fromIndex) | booleano |
| booleanos | **Estatico Array.isArray** | Estatico Array.isArray(Array) | booleano |
| booleanos | **every** | every(funcioncallback(element, index, array ),Argumentos ) | booleano |
| booleanos | **some** | some(funcioncallback(element, index, array ),Argumentos) | booleano |
| buscar | **find** | find(funcioncallback(currentValue, index, array )) | elemento |
| buscar | **findLast** | findLast( funcioncallback(currentValue, index, array )) | elemento |
| buscar | **findIndex** | findIndex( funcioncallback(currentValue, index, array ) ) | indice |
| buscar | **findLastIndex** | findLastIndex(funcioncallback(currentValue, index, array )) | indice |
| buscar | **lastIndexOf** | lastIndexOf(searchElement, fromIndex) | indice |
| buscar | **IndexOf** | IndexOf() | indice |
| filtros | **filter** | filter(funcioncallback(currentValue, index, array ),Argumentos) | SubArray flitrado |
| filtros | **flat** | flat(arrayMultidim) | Array aplanado |
| filtros | **flatMap** | flatMap(funcioncallback(currentValue, index, array )) | Array aplanado con funcion aplicado |
| filtros | **forEach** | forEach(funcioncallback(currentValue, index, array )) | Hace la funcion devuelve undefined |
| filtros | **map** | map(funcioncallback(currentValue, index, array )) | Array con funcion aplicada |
| reduccion | **reduce** | reduce(fReductora(acumulador,elemento,Índice ,Array ),valorInicial ) | Elemento reduccion |
| reduccion | **reduceRight** | reduceRight(fReductora(acumulador,elemento,Índice ,Array ),valorInicial ) | Elemento reduccion |
| Cadenas | **join** | join(Separador) toString() toLocaleString() | String |


# Ejercicios Propuestos 1

## a) Categoria: Creación

### 0. Metodos Estaticos
### Array.from(fcallback)
### Array.of()
### constructor Array

Estos son los metodos **ESTATICOS** de Array

El método Array.from([fcallback]) crea una nueva instancia de Array a partir de un objeto iterable si lleva la funcion de fcallback le aplica la funcion a cada elemento.

```
console.log(Array.from('foo')); // Expected output: Array ["f", "o", "o"]
console.log(Array.from([1, 2, 3], (x) => x + x)); // Expected output: Array [2, 4, 6]
```

El método Array.of() crea una nueva instancia Array con un número variable de elementos pasados como argumento, independientemente del número o del tipo.

La diferencia entre Array.of() y el constructor Array reside en como maneja los parámetros de tipo entero: Array.of(7) crea un array con un solo elemento, 7, mientras que Array(7) crea un array vacío con una propiedad length de 7 (**Nota:** esto implica un array de 7 ranuras vacías, no ranuras con valores undefined).

```
Array.of(7); // [7]
Array.of(1, 2, 3); // [1, 2, 3]
Array(7); // [ , , , , , , ]
Array(1, 2, 3); // [1, 2, 3]
```

## b) Categoria: Acceso

### 4-5. Array.at()  Array[]  Array.with(índice, valor)

Es equivalente al operador de arrays []

```
const array1 = [5, 12, 8, 130, 44];
let index = 2;
console.log(`Using an index of ${index} the item returned is ${array1.at(index)}`);

// Expected output: "Using an index of 2 the item returned is 8"
index = -2;

console.log(`Using an index of ${index} item returned is ${array1.at(index)}`);
// Expected output: "Using an index of -2 item returned is 130"
```

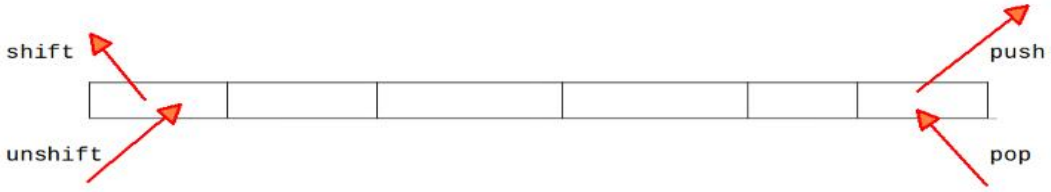
El método with() de las instancias de Array es la versión **copiada** del uso de la notación entre corchetes para cambiar el valor de un índice determinado. Devuelve una nueva matriz con el elemento en el índice dado reemplazado por el valor dado.

Sintaxis arrayInstance.with(índice, valor)

índice : Índice de base cero en el que cambiar la matriz, convertido a un número entero. El índice negativo cuenta hacia atrás desde el final de la matriz; Si el índice después de la normalización está fuera de los límites, se genera un RangeError.

valor Cualquier valor que se asignará al índice dado.

# 6-7 .Array.unshift(...Args )Array.shift() Array.push(...Args ) Array.pop()




El método unshift() agrega uno o más elementos al inicio del array, y devuelve la nueva longitud del array.

```
const array1 = [1, 2, 3];
console.log(array1.unshift(4, 5)); // Expected output: 5
console.log(array1); // Expected output: Array [4, 5, 1, 2, 3]
```

El método shift() agrega uno o más elementos al final del array, y devuelve la nueva longitud del array.

```
const array1 = [1, 2, 3];
console.log(array1.shift(4, 5));// Expected output: 5
console.log(array1); // Expected output: Array [ 1, 2, 3,4, 5]
```

El método **pop()** elimina el **último** elemento de un array y lo devuelve. Este método cambia la longitud del array.

```
const plants = ['broccoli', 'cauliflower', 'cabbage', 'kale', 'tomato'];
console.log(plants.pop()); // Expected output: "tomato"
console.log(plants); // Expected output: Array ["broccoli", "cauliflower", "cabbage", "kale"]
plants.pop();
console.log(plants); // Expected output: Array ["broccoli", "cauliflower", "cabbage"]
```

El método **push()** añade uno o más elementos al final de un array y devuelve la nueva longitud del array.

***La diferencia entre los cuatro métodos de push pop shift unshift***

Array.prototype.push: inserta un miembro al final de la matriz

Array.prototype.pop: lleva un miembro al final de la matriz

Array.prototype.shift: elimina un miembro del encabezado de la matriz

Array.prototype.unshift: inserta un miembro en el encabezado de la matriz

***La diferencia entre*** **Su valor de retorno**

Los métodos Array.prototype.push y Array.prototype.unshift devuelven la longitud del array nuevo

Los métodos Array.prototype.pop y Array.prototype.shift devuelven los miembros eliminados


# c) Categoria: Modificar Array

## 8. Array.fill(valor, *inicio, fin* )

### *arr.fill(value[, start = 0[, end = this.length]])*

El método fill() cambia todos los elementos en un arreglo por un valor estático, desde el índice start (por defecto 0) hasta el índice end (por defecto array.length). Devuelve el arreglo modificado. *inicio, fin son opcionales si no los ponemos por defectos son 0 y longitud del Array.*

```
const array1 = [1, 2, 3, 4];

// Fill with 0 from position 2 until position 4
console.log(array1.fill(0, 2, 4));
// Expected output: Array [1, 2, 0, 0]

// Fill with 5 from position 1
console.log(array1.fill(5, 1));
// Expected output: Array [1, 5, 5, 5]

console.log(array1.fill(6));
// Expected output: Array [6, 6, 6, 6]
```

## 9. Array.concat(Array)

Concatena Arrays los pone uno tras otro

```
const array1 = ['a', 'b', 'c'];
const array2 = ['d', 'e', 'f'];
const array3 = array1.concat(array2);
console.log(array3);

// Expected output: Array ["a", "b", "c", "d", "e", "f"]
```

## Ejercicios Propuestos 2

## 10. Array.copyWithin(target, start, end)

Rellena en un array desde la posicion  start a  end con  el elemento en el indice target

Claro, te lo explico de forma sencilla y con ejemplos claros.

## 🔄 ¿Qué hace copyWithin() en JavaScript?

copyWithin() es un método de los **arrays** que **copia una parte del propio array hacia otra posición dentro del mismo**, **sin cambiar su longitud**.
Es *in-place*, es decir, **modifica el array original**.


---

# 📌 Sintaxis

arr.copyWithin(target, start, end);

## Parámetros:

- **target** → índice donde quieres pegar la copia.
- **start** → índice desde donde comienzas a copiar.
- **end** (opcional) → índice donde dejas de copiar (sin incluirlo).
  Si no se indica, copia hasta el final.

  Sirve para
- Para desplazar bloques de datos dentro de un array.
- Para hacer operaciones de buffer o manipulación de datos sin crear arrays nuevos.
- Es usado en tareas de rendimiento o procesamiento de arrays grandes.

| Acción | ¿Qué hace? |
|---|---|
| Modifica el array | ✔ Sí |
| Cambia la longitud | ✘ No |
| Copia dentro del mismo array | ✔ |
| Similar a slice + asignación | ✔ pero más rápido |

---

# Ejemplos paso a paso

## Ejemplo 1: copia simple

```
let arr = [10, 20, 30, 40, 50];
arr.copyWithin(0, 3);
console.log(arr);
// Resultado: [40, 50, 30, 40, 50]
```

**Explicación:**

- target = 0 → pegamos desde la posición 0
- start = 3 → copiamos desde el valor 40
- Copia [40, 50] sobre el inicio
  → reemplaza 10 y 20

## Ejemplo 2: con end

```
let arr = [1, 2, 3, 4, 5];
arr.copyWithin(1, 3, 5);
console.log(arr);
```


// Resultado: [1, 4, 5, 4, 5]

**Explicación:**

- Copia los elementos desde índice 3 hasta antes del 5 → [4, 5]
- Los pega a partir de índice 1

## Ejemplo 3: índices negativos

Funcionan igual que en slice().

let arr = [1, **2, 3**, **4**, 5];
arr.copyWithin(-2, -4, -1);
console.log(arr);
// Resultado: [1, 2, 3, 2, 3]

**Explicación:**

- target = -2 → posición 3
- start = -4 → posición 1
- end = -1 → posición 4
  Copia [2, 3] sobre [4, 5]

```
const array1 = ['a', 'b', 'c', 'd', 'e'];

// Copy to index 0 the element at index 3
console.log(array1.copyWithin(0, 3, 4));
// Expected output: Array ["d", "b", "c", "d", "e"]

// Copy to index 1 all elements from index 3 to the end
console.log(array1.copyWithin(1, 3));
// Expected output: Array ["d", "d", "e", "d", "e"]

const array4 = ['a', 'b', 'c', 'd', 'e'];
console.log(array4.copyWithin(0, 1, 4));
// Expected output: Array ["a", "a", "a", "a", "a"]
```

## 11. Array.entries()  Array.keys() Array.values()

El método **entries()** retorna un nuevo objeto **Array Iterator** que contiene los pares clave/valor para cada índice de la matriz.

```
const array1 = ['a', 'b', 'c'];
const iterator1 = array1.entries();
console.log(iterator1.next().value);// Expected output: Array [0, "a"]
console.log(iterator1.next().value);// Expected output: Array [1, "b"]

var a = ['a', 'b', 'c'];
var iterator = a.entries();

for (let e of iterator) {
```


```
  console.log(e);
}
// [0, 'a']
// [1, 'b']
// [2, 'c']
```

El método keys() devuelve un nuevo objeto **Array Iterator** que contiene las claves para cada indice en el arreglo.

```
const array1 = ['a', 'b', 'c'];
const iterator = array1.keys();
 for (const key of iterator) {   console.log(key); }
// Expected output: 0 // Expected output: 1  // Expected output: 2
```

El método values() devuelve un nuevo objeto Array Iterator que contiene los valores para cada índice del array.

```
var a = ["w", "y", "k", "o", "p"];
var iterator = a.values();
console.log(iterator.next().value); // w
console.log(iterator.next().value); // y
console.log(iterator.next().value); // k
console.log(iterator.next().value); // o
console.log(iterator.next().value); // p
```

keys y Values se usan para poder usar los arrays con for of.


# 12. Array.slice(inicio [, fin]])

El método slice() devuelve una **copia** de una parte del array dentro de un nuevo array empezando por *inicio* hasta *fin* (*fin* no incluido). El array original no se modificará.

El código fuente de esta demostración interactiva está alojado en un repositorio Github. Si desea contribuir con ella, por favor clone https://github.com/mdn/interactive-examples y envíenos un *"pull request"*.

## Parámetros

**Inicio** **-->** Índice donde empieza la extracción. El primer elemento corresponde con el índice 0.

- Si el índice especificado es negativo, indica un desplazamiento desde el final del array.slice(-2) extrae los dos últimos elementos del array
- Si inicio es omitido el valor por defecto es 0.
- Si inicio es mayor a la longitud del array, se devuelve un array vacío.

**Fin -->** Índice que marca el final de la extracción. slice extrae hasta, pero sin incluir el final.

- slice(1,4) extrae desde el segundo elemento hasta el cuarto (los elementos con índices 1, 2, y 3).
- Con un índice negativo, fin indica un desplazamiento desde el final de la secuencia. slice(2,-1) extrae desde el tercer hasta el penúltimo elemento en la secuencia.
- Si fin es omitido, slice extrae hasta el final de la secuencia (arr.length).
- Si fin es mayor a la longitud del array, slice extrae hasta el final de la secuencia (arr.length).

## Valor de retorno

Un nuevo array con los valores extraídos.

## Descripción

slice **no modifica** el array original. Devuelve una copia plana (*shallow copy*) de los elementos especificados del array original. Los elementos del array original son copiados en el array devuelto de la siguiente manera:

- Para referencias de objeto ( **no** el objeto en sí ), slice copia la referencia dentro del nuevo array. Ambos, el array original y el nuevo, referencian al mismo objeto. Si un objeto referenciado cambia, los cambios son visibles para ambos arrays.
- Para strings, numbers y boolean (**no** objetos String y Number), slice copia los valores en el nuevo array. Los cambios a los string, numbers y boolean en un array no afectan a los del otro array.

Si un nuevo elemento es agregado a cualquiera de los arrays, el otro array no es afectado.

## Ejemplos

## Ejemplo: Devolver una porción de un array existente

```
var nombres = ["Rita", "Pedro", "Miguel", "Ana", "Vanesa"];
var masculinos = nombres.slice(1, 3);
```


```
// masculinos contiene ['Pedro','Miguel']
```

## Ejemplo: Utilizando slice

Presta especial atención a:

- Valores de tipos básicos, como string o number, son copiados al nuevo array. Cambiar estos valores en la copia no afecta al array original.
- Las referencias también se copian. Mismas referencias acceden al mismo objeto destino. Cambios en el objeto destino son compartidos por todos sus accesos.

En el siguiente ejemplo, slice crea un nuevo array, nuevoCoche, de myCoche. Los dos incluyen una referncia al objecto miHonda se cambia a púrpura, ambas matrices reflejan el cambio.

```
var miHonda = {
  color: "red",
  ruedas: 4,
  motor: { cilindros: 4, cantidad: 2.2 },
};
var miCoche = [miHonda, 2, "Buen estado", "comprado 1997"];
var nuevoCoche = miCoche.slice(0, 2);

//  Muestra los valores de myCar, newCar y el color de myHonda.
console.log("miCoche = " + JSON.stringify(miCoche));
console.log("nuevoCoche = " + JSON.stringify(nuevoCoche));
console.log("miCoche[0].color = " + miCoche[0].color);
console.log("nuevoCoche[0].color = " + nuevoCoche[0].color);

// Cambia el color de miHonda.
miHonda.color = "azul";
console.log("El nuevo color de mi Honda es " + miHonda.color);

// Muestra el color de myHonda referenciado desde ambos arrays.
console.log("miCoche[0].color = " + miCoche[0].color);

console.log("nuevoCoche[0].color = " + nuevoCoche[0].color);
```

Este script escribe:

```
miCoche = [{color: 'rojo', ruedas: 4, motor: {cilindros: 4, cantidad: 2.2}}, 2,
      'buen estado', 'comprado 1997']
nuevoCoche = [{color: 'rojo', ruedas: 4, motor: {cilindros: 4, cantidad: 2.2}}, 2]
miCoche[0].color = rojo
nuevoCoche[0].color = rojo
El nuevo color de miHonda es azul
miCoche[0].color = azul
nuevoCoche[0].color = azul
```

## Objetos array-like

**Nota:** Se dice que un objeto es **array-like** ( similar o que se asemeja a un array) cuando entre sus propiedades existen algunas cuyos nombres son **números** y en particular tiene una propiedad llamada **length**. Este hecho hace suponer que el objeto es algún tipo de colección de elementos indexados por números. Es conveniente, a veces, convertir estos objetos a arrays para otorgarles la funcionalidad que de serie se incorpora en todos los arrays a través de su prototipo.


El método slice puede ser usado para convertir objetos parecidos a arrays o colecciones a un nuevo Array. Simplemente debe enlazar el método al objeto. El arguments (en-US) dentro de una función es un ejemplo de un objeto parecido a arrays.

```
function list() {
  return Array.prototype.slice.call(arguments, 0);
}

var list1 = list(1, 2, 3); // [1, 2, 3]
```

El enlazado puede realizarse con la función .call de Function.prototype (en-US) y puede ser abreviado también usando [].slice.call(arguments) en lugar de Array.prototype.slice.call. En cualquier caso, puede ser simplificado usando bind.

```
var unboundSlice = Array.prototype.slice;
var slice = Function.prototype.call.bind(unboundSlice);

function list() {
  return slice(arguments, 0);
}

var list1 = list(1, 2, 3); // [1, 2, 3]
```

## 12 Array.splice(iniciocorte, fincorte,item1, item2 ... )

El método splice() cambia el contenido de un array eliminando elementos existentes y/o agregando nuevos elementos.

**array.splice(start[, deleteCount[, item1[, item2[, ...]]]])**

**start o iniciocorte** : Índice donde se comenzará a cambiar el array (con 0 como origen). Si es mayor que la longitud del array, el punto inicial será la longitud del array. Si es negativo, empezará esa cantidad de elementos contando desde el final.

**deleteCount o fincorte es Opcional** : Un entero indicando el número de elementos a eliminar del array antiguo.

> Si deleteCount se omite, o si su valor es mayor que arr.length - start (esto significa, si es mayor que el número de elementos restantes del array, comenzando desde start), entonces todos los elementos desde start hasta el final del array serán eliminados.
>
> Si deleteCount es igual a 0 o negativo, no se eliminará ningún elemento. En este caso, se debe especificar al menos un nuevo elemento (ver más abajo).

**item1, item2, ... Opcional** - Los elementos que se agregarán al array, empezando en el índice start. Si no se especifica ningún elemento, splice() solamente eliminará elementos del array.

## Valor devuelto

Un array que contiene los elementos eliminados. Si sólo se ha eliminado un elemento, devuelve un array con un solo elemento. Si no se ha eliminado ningún elemento, devuelve un array vacío.

## Descripción

Si especifica un número diferente de elementos a agregar que los que se eliminarán, el array tendrá un tamaño diferente al original una vez finalizada la llamada.


## Eliminar 0 elementos desde el índice 2 e insertar "drum"

```
var myFish = ['angel', 'clown', 'mandarin', 'sturgeon'];
var removed = myFish.splice(2, 0, 'drum');

// myFish is ["angel", "clown", "drum", "mandarin", "sturgeon"]
// removed is [], no elements removed
```

## Eliminar 1 elemento desde el índice 3

```
var myFish = ['angel', 'clown', 'drum', 'mandarin', 'sturgeon'];
var removed = myFish.splice(3, 1);

// removed is ["mandarin"]
// myFish is ["angel", "clown", "drum", "sturgeon"]
```

## Eliminar 1 elemento desde el índice 2 e insertar "trumpet"

```
var myFish = ['angel', 'clown', 'drum', 'sturgeon'];
var removed = myFish.splice(2, 1, 'trumpet');

// myFish is ["angel", "clown", "trumpet", "sturgeon"]
// removed is ["drum"]
```

## Eliminar 2 elementos desde el índice 0 e insertar "parrot", "anemone" y "blue"

```
var myFish = ['angel', 'clown', 'trumpet', 'sturgeon'];
var removed = myFish.splice(0, 2, 'parrot', 'anemone', 'blue');

// myFish is ["parrot", "anemone", "blue", "trumpet", "sturgeon"]
// removed is ["angel", "clown"]
```

## Eliminar 2 elementos desde el índice 2

```
var myFish = ['parrot', 'anemone', 'blue', 'trumpet', 'sturgeon'];
var removed = myFish.splice(myFish.length - 3, 2);

// myFish is ["parrot", "anemone", "sturgeon"]
// removed is ["blue", "trumpet"]
```

## Eliminar 1 elemento desde el índice -2

```
var myFish = ['angel', 'clown', 'mandarin', 'sturgeon'];
var removed = myFish.splice(-2, 1);

// myFish is ["angel", "clown", "sturgeon"]
// removed is ["mandarin"]
```

## Eliminar todos los elementos tras el índice 2 (incl.)

```
var myFish = ['angel', 'clown', 'mandarin', 'sturgeon'];
var removed = myFish.splice(2);
```


```
// myFish is ["angel", "clown"]
// removed is ["mandarin", "sturgeon"]
```

# ¿Qué hace splice()?

splice() es uno de los métodos más potentes (y a veces peligrosos) de los arrays porque **modifica el array original** y permite:

1. **Eliminar** elementos
2. **Agregar** elementos
3. **Reemplazar** elementos

Todo desde una **posición específica**.

## Sintaxis

array.splice(start, deleteCount, item1, item2, ...)

### Parámetros:

- **start** → índice donde empieza la operación
- **deleteCount** → cuántos elementos eliminar desde start
- **item1, item2, ...** (opcional) → elementos a insertar

# Comportamiento general

Según cómo lo uses, splice() puede:

✔ Eliminar elementos
✔ Insertar elementos
✔ Reemplazar elementos (eliminar + insertar)
✔ Devolver un array con los elementos eliminados

## Resumen rápido

| Acción | ¿Cómo? |
|---|---|
| Eliminar | splice(start, n) |
| Insertar | splice(start, 0, item1, item2...) |
| Reemplazar | splice(start, n, item1...) |
| Devuelve | Array con lo eliminado |


# Ejemplos prácticos

## Eliminar elementos

```
let arr = [10, 20, 30, 40, 50];
let removed = arr.splice(2, 2);

console.log(arr);      // [10, 20, 50]
console.log(removed);  // [30, 40]
```

**Explicación:**

- Desde índice **2**
- Elimina **2** elementos → 30 y 40
- Devuelve los eliminados

## Insertar elementos sin eliminar

```
let arr = [1, 2, 3];
arr.splice(1, 0, "A", "B");
console.log(arr);
// [1, "A", "B", 2, 3]
```

**Explicación:**

- start = 1
- deleteCount = 0 → no borra nada
- Inserta "A" y "B" en esa posición

## Reemplazar elementos (delete + insert)

```
let arr = [100, 200, 300, 400];
arr.splice(1, 2, "X", "Y");
console.log(arr);
// [100, "X", "Y", 400]
```

**Explicación:**

- Empieza en índice **1**
- Elimina **2** (200 y 300)
- Inserta "X" y "Y" en su lugar

## Índices negativos

```
let arr = [1, 2, 3, 4, 5];
arr.splice(-2, 1);
console.log(arr);
// [1, 2, 3, 5]
```

**Explicación:**


- start = -2 → posición 3 (el valor 4)
- Elimina 1 elemento → 4

# 13. Array.sort([compareFunction])  Array.toSorted()

El método sort() ordena los elementos de un arreglo (array) *localmente* y devuelve el arreglo ordenado. La ordenación no es necesariamente estable. El modo de ordenación por defecto responde a la posición del valor del string de acuerdo a su valor Unicode.

## Sintaxis

arr.sort([compareFunction])

## Parámetros

**CompareFunction ( FirstEl,SecondEl)-** Opcional. Especifica una función que define el modo de ordenamiento. Si se omite, el array es ordenado atendiendo a la posición del valor Unicode (en-US) de cada caracter, según la conversión a string de cada elemento.

**FirstEl  -->** El primer elemento a comparar.

**SecondEl  -->**  El segundo elemento a comparar.

## Valor devuelto

El array ordenado.

## Descripción

Si no se provee compareFunction, los elementos son ordenados convirtiéndolos a strings y comparando la posición del valor Unicode de dichos strings. Por ejemplo, "Cherry" viene antes que "banana" (porque las mayúsculas van antes que las minúsculas en la codificación Unicode) . En un ordenamiento numérico, 9 está antes que 80, pero dado que los números son convertidos a strings y ordenados según el valor Unicode, el resultado será "80" antes que "9".

```
const frutas = ["guindas", "manzanas", "bananas"];
frutas.sort(); // ['bananas', 'guindas', 'manzanas']

const puntos = [1, 10, 2, 21];
puntos.sort(); // [1, 10, 2, 21]
// Tenga en cuenta que 10 viene antes que 2
// porque '10' viene antes que '2' según la posición del valor Unicode.

const cosas = ["word", "Word", "1 Word", "2 Words"];
cosas.sort(); // ['1 Word', '2 Words', 'Word', 'word']
// En Unicode, los números vienen antes que las letras mayúsculas
// y estas vienen antes que las letras minúsculas.
```


Si se provee compareFunction, los elementos del array son ordenados de acuerdo al valor que retorna dicha función de comparación. Siendo a y b dos elementos comparados, entonces:

- Si compareFunction(a, b) es menor que 0, se sitúa a en un indice menor que b. Es decir, a viene primero.
- Si compareFunction(a, b) retorna 0, se deja a y b sin cambios entre ellos, pero ordenados con respecto a todos los elementos diferentes. Nota: el estandar ECMAscript no garantiza este comportamiento, por esto no todos los navegadores (p.ej. Mozilla en versiones que datan hasta el 2003) respetan esto.
- Si compareFunction(a, b) es mayor que 0, se sitúa b en un indice menor que a.
- compareFunction(a, b) siempre debe retornar el mismo valor dado un par especifico de elementos a y b como sus argumentos. Si se retornan resultados inconsistentes entonces el orden de ordenamiento es indefinido.

Entonces, la función de comparación tiene la siguiente forma:

```
function compare(a, b) {
  if (a es menor que b según criterio de ordenamiento) {
    return -1;
  }
  if (a es mayor que b según criterio de ordenamiento) {
    return 1;
  }
  // a debe ser igual b
  return 0;
}
```

Para comparar números en lugar de strings, la función de comparación puede simplemente restar b de a. La siguiente función ordena el array de modo ascendente:

```
const compareNumbers = (a, b) => a - b;
```

El metodo sort puede ser usado convenientemente con function expressions (y closures):

```
const numbers = [4, 2, 5, 1, 3];
numbers.sort(function (a, b) {
  return a - b;
});
console.log(numbers); // [1, 2, 3, 4, 5]
```

Los objectos pueden ser ordenados por el valor de una de sus propiedades.

```
const items = [
  { name: "Edward", value: 21 },
  { name: "Sharpe", value: 37 },
  { name: "And", value: 45 },
  { name: "The", value: -12 },
  { name: "Magnetic", value: 13 },
  { name: "Zeros", value: 37 },
];
items.sort(function (a, b) {
  if (a.name > b.name) {
    return 1;
  }
  if (a.name < b.name) {
    return -1;
  }
  // a must be equal to b
  return 0;
});
```


## Ejemplos

## Ordenando un array

Un array de elementos string, sin especificar una función de comparación:

```
const arr = ["a", "b", "Z", "Aa", "AA"];
arr.sort(); //[ 'AA', 'Aa', 'Z', 'a', 'b' ]
```

Un array de elementos numéricos, sin función de comparación:

```
const arr = [40, 1, 5, 200];
arr.sort(); //[ 1, 200, 40, 5 ]
```

Un array de elementos numéricos, usando una función de comparación:

```
const arr = [40, 1, 5, 200];
function comparar(a, b) {
  return a - b;
}
arr.sort(comparar); // [1, 5, 40, 200]
```

Lo mismo pero usando una función anónima normal:

```
const arr = [40, 1, 5, 200];
arr.sort(function (a, b) {
  return a - b;
}); // [ 1, 5, 40, 200 ]
```

Lo mismo escrito más compacto mediante una función flecha:

```
const arr = [40, 1, 5, 200];
arr.sort((a, b) => a - b); // [ 1, 5, 40, 200 ]
```

## Creando, mostrando, y ordenando un array

El siguiente ejemplo abunda en la idea de ordenar con y sin función de comparación. Además, ilustra una manera de mostrar un array una vez creado. El método join es usado para convertir el array en una cadena de texto que imprimir. Al no pasarle un argumento que indique el separador, usará la coma por defecto para separar los elementos del array dentro de la cadena.

```
const arr = ["80", "9", "700", 40, 1, 5, 200];
function comparar(a, b) {
  return a - b;
}

console.log("original:", arr.join());
console.log("ordenado sin función:", arr.sort());
console.log("ordenado con función:", arr.sort(comparar));
```

El ejemplo produce el siguiente resultado. Como muestra la salida, cuando una función de comparación es usada, los números se ordenan correctamente, sean estos valores numéricos o strings numéricos.

```
original: 80,9,700,40,1,5,200
ordenado sin función: 1,200,40,5,700,80,9
ordenado con función: 1,5,9,40,80,200,700
```


## Ordenando caracteres no ASCII

Para ordenar strings con characters no ASCII, i.e. strings con caracteres con acento (e, é, è, a, ä, etc.), strings de lenguajes diferentes al inglés: use String.localeCompare. Esta función puede comparar esos caracteres para que aparezcan en el orden correcto.

```
const items = ["réservé", "premier", "cliché", "communiqué", "café", "adieu"];
items.sort(function (a, b) {
  return a.localeCompare(b);
});

// items is ['adieu', 'café', 'cliché', 'communiqué', 'premier', 'réservé']
```

## Ordenando con map

La compareFunction puede ser invocada múltiples veces por elemento dentro del array. Dependiendo de la naturaleza de compareFunction, este puede resultar en una alta penalización de rendimiento. Cuanto más trabajo hace una compareFunction y más elementos hay para ordenar, resulta más recomendable usar una función map para ordenar. La idea es recorrer el array una sola vez para extraer los valores usados para ordenar en un array temporal, ordenar el array temporal y luego recorrer el array para lograr el orden correcto.

```
// el array a ordenar
const list = ["Delta", "alpha", "CHARLIE", "bravo"];

// array temporal contiene objetos con posición y valor de ordenamiento
const mapped = list.map(function (el, i) {
  return { index: i, value: el.toLowerCase() };
});

// ordenando el array mapeado que contiene los valores reducidos
mapped.sort(function (a, b) {
  if (a.value > b.value) {
    return 1;
  }
  if (a.value < b.value) {
    return -1;
  }
  return 0;
});

// contenedor para el orden resultante
const result = mapped.map(function (el) {
  return list[el.index];
});
```

# 14. Array.reverse() Array.toReverse()

El método reverse() invierte el orden de los elementos de un array *in place*. El primer elemento pasa a ser el último y el último pasa a ser el primero. El método reverse cruza los elementos del objeto matriz invocados en su lugar, mutando la matriz, y retornando una referencia a la misma. **Array.toReverse() es la version que te da una copia del array**


```
 const array1 = ['one', 'two', 'three'];
console.log('array1:', array1);
// Expected output: "array1:" Array ["one", "two", "three"]

const reversed = array1.reverse();
console.log('reversed:', reversed);
// Expected output: "reversed:" Array ["three", "two", "one"]

// Careful: reverse is destructive -- it changes the original array.
console.log('array1:', array1);
// Expected output: "array1:" Array ["three", "two", "one"]
```

### Colocar al revés los elementos de un array

El siguiente ejemplo crea un array a que contiene tres elementos y luego lo invierte. La llamada a reverse() devuelve una referencia al array a invertido.

```
const a = [1, 2, 3];
console.log(a); // [1, 2, 3]
a.reverse();
console.log(a); // [3, 2, 1]
```

## Ejercicios Propuestos 3

# d) Categoria: Booleanos

## 15 Array.includes(elemento,*fromIndex*)

El método includes() determina si una matriz incluye un determinado elemento, devuelve true o false según corresponda.

**elemento** es el elemento a buscar

**fromIndex** es opcional es apartir de donde buscamos . Si fromIndex es mayor o igual que la longitud de la matriz, se devuelve false. No se buscará en la matriz.

```
const array1 = [1, 2, 3];
console.log(array1.includes(2)); // Expected output: true

const pets = ['cat', 'dog', 'bat'];
console.log(pets.includes('cat')); // Expected output: true
console.log(pets.includes('at'));  // Expected output: false
```

### Ejemplos

```
[1, 2, 3].includes(2);     // true
[1, 2, 3].includes(4);     // false
[1, 2, 3].includes(3, 3);  // false
[1, 2, 3].includes(3, -1); // true
[1, 2, NaN].includes(NaN); // true
```


```
var arr = ['a', 'b', 'c'];
arr.includes('c', 3);   // false
arr.includes('c', 100); // false
```

Si **fromIndex** es negativo, el índice calculado se calcula para usarse como una posición en la matriz en la cual comenzar a buscar searchElement. Si el índice calculado es menor que 0, se buscará la matriz completa.

```
// la longitud de la matriz es 3
// fromIndex es -100
// el índice calculado es 3 + (-100) = -97

var arr = ['a', 'b', 'c'];

arr.includes('a', -100); // true
arr.includes('b', -100); // true
arr.includes('c', -100); // true
```

## 16 ESTATICOS Array.isArray ()

Estos son los metodos **ESTATICOS** de Array

El método Array.isArray() determina si el valor pasado es un Array.

```
Array.isArray([1, 2, 3]); // true
Array.isArray({ foo: 123 }); // false
Array.isArray("foobar"); // false
Array.isArray(undefined); // false
```

## 17. Array.every(funcioncallback(element, *index, array* ),*Argumentos* )

Determina si todos los elementos en el array satisfacen una condición.Llamar este método en un array vacío devuelve true para cualquier condición

La funcion **funcioncallback** debe devolver un booleano y tener al menos un parámetro **element** que representa el elemento que esta recorriendo.
Los elementos **index y array** son opcionales y se refieren al indice y a todo el array.

### *arr.every(callback(element[, index[, array]])[, thisArg])*

```
const isBelowThreshold = (currentValue) => currentValue < 40;

const array1 = [1, 30, 39, 29, 10, 13];

console.log(array1.every(isBelowThreshold));
// Expected output: true
```

## 6. Array.some(funcioncallback(element, index, array ),Argumentos)

Determina si algun los elementos en el array satisfacen una condición.Llamar este método en un array vacío devuelve false para cualquier condición


### *arr.every(callback(element[, index[, array]])[, thisArg])*

```
const array = [1, 2, 3, 4, 5];
// Checks whether an element is even
const even = (element) => element % 2 === 0;
console.log(array.some(even));
// Expected output: true
```

## Ejemplos de uso de los métodos .every() y .some() en JavaScript con arrays:

1. Método .every(): Este método verifica si todos los elementos de un array cumplen una condición determinada. Devuelve true si todos los elementos cumplen la condición, y false en caso contrario.

```
const numbers = [2, 4, 6, 8, 10];

// Verificar si todos los números son pares
const allEven = numbers.every((number) => number % 2 === 0);
console.log(allEven); // Output: true

// Verificar si todos los números son mayores que 5
const allGreaterThanFive = numbers.every((number) => number > 5);
console.log(allGreaterThanFive); // Output: false
```

En este ejemplo, usamos el método .every() para verificar si todos los elementos del array numbers son pares y si todos los elementos son mayores que 5.

2. Método .some(): Este método verifica si al menos un elemento de un array cumple una condición determinada. Devuelve true si al menos un elemento cumple la condición, y false en caso contrario.

```
const numbers = [1, 3, 5, 7, 10];

// Verificar si al menos un número es par
const hasEvenNumber = numbers.some((number) => number % 2 === 0);
console.log(hasEvenNumber); // Output: true

// Verificar si al menos un número es menor que 0
const hasNegativeNumber = numbers.some((number) => number < 0);
console.log(hasNegativeNumber); // Output: false
```

En este ejemplo, usamos el método .some() para verificar si al menos un elemento del array numbers es par y si al menos un elemento es menor que 0.

Estos métodos son muy útiles para realizar validaciones y comprobaciones en arrays. Puedes utilizarlos con funciones de comparación o condiciones personalizadas según tus necesidades.

Reto 1: Verificar si todos los elementos de un array son números positivos.

```
const numbers = [1, 2, 3, 4, 5];
const allPositive = numbers.every((number) => number > 0);
console.log(allPositive); // Output: true
```

Reto 2: Verificar si al menos un elemento de un array es un número par.

```
const numbers = [1, 3, 5, 7, 10];
```


```
const hasEvenNumber = numbers.some((number) => number % 2 === 0);
console.log(hasEvenNumber); // Output: true
```

Reto 3: Verificar si todos los elementos de un array son cadenas de texto.

```
const values = ['hello', 'world', '123', 'JavaScript'];
const allStrings = values.every((value) => typeof value === 'string');
console.log(allStrings); // Output: true
```

Reto 4: Verificar si al menos una palabra de un array de cadenas de texto contiene la letra 'a'.

```
const words = ['apple', 'banana', 'orange', 'grapefruit'];
const hasWordWithA = words.some((word) => word.includes('a'));
console.log(hasWordWithA); // Output: true
```

Estos retos son solo ejemplos para mostrarte cómo puedes utilizar los métodos .every() y .some() en diferentes situaciones. Puedes adaptarlos y crear tus propios desafíos según tus necesidades y creatividad.

Espero que estos retos te resulten interesantes y te ayuden a practicar con los métodos .every() y .some(). Si tienes más preguntas o necesitas más ejemplos, no dudes en hacerlas. ¡Disfruta resolviendo desafíos!


## d) Categoria: Buscar

## 19. Array.find(funcioncallback(currentValue, *index, array* ))

## 20. Array.findLast( funcioncallback(currentValue, *index, array* ))

El método **find()** devuelve el **valor** del **primer elemento** del array que cumple la función de prueba proporcionada.

```
const array1 = [5, 12, 8, 130, 44];
const found = array1.find(element => element > 10);
console.log(found);
// Expected output: 12
```

el método find ejecuta la función callback una vez por cada índice del array hasta que encuentre uno en el que el callback devuelva un valor verdadero. Si es así, find devuelve inmediatamente el valor del elemento. En caso contrario, find devuelve undefined.

callback se invoca con tres argumentos: el valor del elemento, el índice del elemento y el objeto Array que está siendo recorrido.

Si un parámetro thisArg es proporcionado al método find, este será utilizado como this para cada invocación del callback. Si no se proporciona el parámetro, entonces se utiliza undefined.

El método find no transforma el array desde el cual es llamado, pero la función proporcionada en callback sí. En ese caso, los elementos procesados por find son establecidos *antes* de la primera invocación de callback. Por lo tanto:

- callback no visitará ningún elemento añadido al array después de que comience la llamada a find.
- Si un elemento existente no visitado del array es modificado por callback, su valor que se pasa al callback que lo visita será el valor en el momento en que find visita ese índice del elemento.
- Los elementos que sean deleted (eliminados) aún se visitan.

### Ejemplos  Encontrar un objeto en un array por una de sus propiedades

```
const inventario = [
   {nombre: 'manzanas', cantidad: 2},
   {nombre: 'bananas', cantidad: 0},
   {nombre: 'cerezas', cantidad: 5}
];

function esCereza(fruta) {
   return fruta.nombre === 'cerezas';
}

console.log(inventario.find(esCereza));
// { nombre: 'cerezas', cantidad: 5 }

const inventario = [
   {nombre: 'manzanas', cantidad: 2},
   {nombre: 'bananas', cantidad: 0},
   {nombre: 'cerezas', cantidad: 5}
];
```


```
const resultado = inventario.find( fruta => fruta.nombre === 'cerezas' );

console.log(resultado); // { nombre: 'cerezas', cantidad: 5 }
```

## 21. Array.findIndex( funcioncallback(currentValue, *index, array* ) )

## 22. Array.findLastIndex(funcioncallback(currentValue, *index, array* ))

El método **findIndex()** devuelve el **índice** del **primer elemento** de un array que cumpla con la función de prueba proporcionada. En caso contrario devuelve -1.

funcioncallback(currentValue, index, array ) funciona igual que en funciones pasadas.

```
const array1 = [5, 12, 8, 130, 44];

const isLargeNumber = (element) => element > 13;

console.log(array1.findIndex(isLargeNumber));
// Expected output: 3
```

## 23-24. Array.lastIndexOf(searchElement, fromIndex)  Array.IndexOf()

El método lastIndexOf() devuelve el último / primer índice en el que un cierto elemento puede encontrarse en el arreglo, ó -1 si el elemento no se encontrara. El arreglo es recorrido en sentido contrario, empezando por el índice fromIndex.

### Sintaxis

```
arr.lastIndexOf(searchElement) arr.lastIndexOf(searchElement, fromIndex)

const animals = ['Dodo', 'Tiger', 'Penguin', 'Dodo'];

console.log(animals.lastIndexOf('Dodo'));
// Expected output: 3

console.log(animals.lastIndexOf('Tiger'));
// Expected output: 1
```

### Parámetros

**SearchElement -** Elemento a encontrar en el arreglo.

**fromIndex Opcional** - El índice en el que empieza la búsqueda en sentido contrario. Por defecto la longitud del arreglo menos uno (arr.length - 1), es decir, todo el arreglo será recorrido. Si el índice es mayor o igual que la longitud del arreglo, todo el arreglo será recorrido. Si es un valor negatigo, se usará como inicio del desplazamiento el final del arreglo. Darse cuenta que aún cuando el índice es negativo, el arreglo todavía será recorrido desde atrás hacia delante. Si el índice calculado es menor de 0, se devolverá -1, es decir, el arreglo no será recorrido.


## Valor de retorno

El último índice del elemento en el arreglo; -1 si no se encuentra.

## Descripción

lastIndexOf compara searchElement con los elementos del arreglo usando igualdad estricta (en-US) (el mismo método es usado para la ===, operador triple igualdad).

## Usando lastIndexOf

El siguiente ejemplo usa lastIndexOf para encontrar valores en un arreglo.

```
var array = [2, 5, 9, 2];
array.lastIndexOf(2); // 3
array.lastIndexOf(7); // -1
array.lastIndexOf(2, 3); // 3
array.lastIndexOf(2, 2); // 0
array.lastIndexOf(2, -2); // 0
array.lastIndexOf(2, -1); // 3
```

## Encontrar todas las apariciones de un elemento

El siguiente ejemplo uses lastIndexOf encuentra todos los índices de un elemento en un arreglo dado, usando push añadiéndolos a otro arreglo como elementos encontrados.

```
var indices = [];
var array = ["a", "b", "a", "c", "a", "d"];
var element = "a";
var idx = array.lastIndexOf(element);
while (idx != -1) {
  indices.push(idx);
  idx = idx > 0 ? array.lastIndexOf(element, idx - 1) : -1;
}

console.log(indices);
// [4, 2, 0]
```


# Ejercicios Propuestos 4

## e) Categoria: filtros

## 25. Array.filter(funcioncallback(currentValue, *index, array* ),*Argumentos*)

El método **filter()** crea un nuevo array con todos los elementos que cumplan la condición implementada por la función dada.

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];
const result = words.filter(word => word.length > 6);

console.log(result);
// Expected output: Array ["exuberant", "destruction", "present"]
```

### Parámetros

**callback :** Función que **comprueba cada elemento** del array para ver si cumple la condición (también llamada predicado). Retorna true si el elemento la cumple o en caso contrario retornará false. Acepta tres parámetros:

- currentValue : El elemento actual del array que está siendo procesado.
- index Opcional : El índice del elemento actual del array que está siendo procesado.
- array Opcional :El array sobre el que se ha llamado filter.

thisArg Opcional

Opcional. Valor a utilizar como this cuando se ejecuta callback.

### Valor devuelto

Un nuevo array con los elementos que cumplen la condición. Si ningún elemento cumple la condición, se devolverá un array vacío.

```
function esSuficientementeGrande(elemento) {
  return elemento >= 10;
}
var filtrados = [12, 5, 8, 130, 44].filter(esSuficientementeGrande);
// filtrados es [12, 130, 44]
```


## 26. Array.flat(arrayMultidim)

El método **flat()** crea una nueva matriz con todos los elementos de sub-array concatenados recursivamente hasta la profundidad especificada.

```
var newArray = arr.flat([depth]);
var arr1 = [1, 2, [3, 4]];
arr1.flat();                    // [1, 2, 3, 4]

var arr2 = [1, 2, [3, 4, [5, 6]]];
arr2.flat();                    // [1, 2, 3, 4, [5, 6]]

var arr3 = [1, 2, [3, 4, [5, 6]]];
arr3.flat(2);                   // [1, 2, 3, 4, 5, 6]
```

## 27 .Array.flatMap(funcioncallback(currentValue, index, array ))

El método **flatMap()** hace dos cosas

- primero mapea cada elemento usando una función de mapeo
- segundo aplana el resultado en una nueva matriz. Es idéntico a un map seguido de un flatten de profundidad 1, pero flatMap es a menudo útil y la fusión de ambos en un método es ligeramente más eficiente.

```
const arr1 = [1, 2, 1];
const result = arr1.flatMap((num) => (num === 2 ? [2, 2] : 1));

console.log(result);
// Expected output: Array [1, 2, 2, 1]
```

## 28.Array.forEach(funcioncallback(currentValue, *index, array* ))

El método **forEach()** ejecuta la función indicada una vez por cada elemento del array.

```
const array1 = ['a', 'b', 'c'];

array1.forEach(element => console.log(element));

// Expected output: "a"
// Expected output: "b"
// Expected output: "c"

arr.forEach(function callback(currentValue, index, array) {
   // tu iterador
}[, thisArg]);
```

### Parámetros

**Fcallback** :Función a ejecutar por cada elemento, que recibe tres argumentos:


- currentValue : El elemento actual siendo procesado en el array.
- index Opcional : El índice del elemento actual siendo procesado en el array.
- array Opcional : El vector en el que forEach() esta siendo aplicado.

**thisArg Opcional** -Valor que se usará como this cuando se ejecute el callback.

**Diferencias forEach() map() reduce()**

forEach() ejecuta la función callback una vez por cada elemento del array; a diferencia de map() o reduce() este siempre devuelve el valor undefined y no es encadenable. El típico uso es ejecutar los efectos secundarios al final de la cadena.

## 29.Array.map(funcioncallback(currentValue, *index, array* ))

El método **map()** crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos.

```
var numbers = [1, 5, 10, 15];
var doubles = numbers.map(function(x) {
  return x * 2;
});
// doubles is now [2, 10, 20, 30]
// numbers is still [1, 5, 10, 15]

var numbers = [1, 4, 9];
var roots = numbers.map(function(num) {
   return Math.sqrt(num);
});
// roots is now [1, 2, 3]
// numbers is still [1, 4, 9]
```

### Sintaxis

```
var nuevo_array = arr.map(function callback(currentValue, index, array) {
   // Elemento devuelto de nuevo_array
}[, thisArg])
```

### Parámetros

callback Función que producirá un elemento del nuevo array, recibe tres argumentos:

currentValue El elemento actual del array que se está procesando.

index El índice del elemento actual dentro del array.

array El array sobre el que se llama map.

thisArg Opcional. Valor a usar como this al ejecutar callback.


## Valor devuelto

Un nuevo array en la que cada elemento es el resultado de ejecutar callback.

## Descripción

map llama a la función callback provista **una vez por elemento** de un array, en orden, y construye un nuevo array con los resultados. callback se invoca sólo para los índices del array que tienen valores asignados; no se invoca en los índices que han sido borrados o a los que no se ha asignado valor.

callback es llamada con tres argumentos: el valor del elemento, el índice del elemento, y el objeto array que se está recorriendo.

Si se indica un parámetro thisArg a un map, se usará como valor de this en la función callback. En otro caso, se pasará undefined como su valor this. El valor de this observable por el callback se determina de acuerdo a las reglas habituales para determinar el valor this visto por una función.

map no modifica el array original en el que es llamado (aunque callback, si es llamada, puede modificarlo).

El rango de elementos procesado por map es establecido antes de la primera invocación del callback. Los elementos que sean agregados al array después de que la llamada a map comience no serán visitados por el callback. Si los elementos existentes del array son modificados o eliminados, su valor pasado al callback será el valor en el momento que el map lo visita; los elementos que son eliminados no son visitados.

## Ejemplos

## Procesar un array de números aplicándoles la raíz cuadrada

El siguiente código itera sobre un array de números, aplicándoles la raíz cuadrada a cada uno de sus elementos, produciendo un nuevo array a partir del inicial.

```
var numeros= [1, 4, 9];
var raices = numeros.map(Math.sqrt);
// raices tiene [1, 2, 3]
// numeros aún mantiene [1, 4, 9]
```

## Usando map para dar un nuevo formato a los objetos de un array

El siguiente código toma un array de objetos y crea un nuevo array que contiene los nuevos objetos formateados.

```
var kvArray = [{clave:1, valor:10},          {clave:2, valor:20},          {clave:3, valor: 30}];

var reformattedArray = kvArray.map(function(obj){
  var rObj = {};
  rObj[obj.clave] = obj.valor;
  return rObj;
});

// reformattedArray es ahora [{1:10}, {2:20}, {3:30}],
// kvArray sigue siendo:
// [{clave:1, valor:10},
//  {clave:2, valor:20},
//  {clave:3, valor: 30}]
```


## Mapear un array de números usando una función con un argumento

El siguiente código muestra cómo trabaja map cuando se utiliza una función que requiere de un argumento. El argumento será asignado automáticamente a cada elemento del arreglo conforme map itera el arreglo original.

```
var numeros = [1, 4, 9];
var dobles  = numeros.map(function(num) {
  return num * 2;
});

// dobles es ahora [2, 8, 18]
// numeros sigue siendo [1, 4, 9]
```

## Usando map de forma genérica

Este ejemplo muestra como usar map en String para obtener un arreglo de bytes en codifcación ASCII representando el valor de los caracteres:

```
var map = Array.map;
var valores = map.call('Hello World', function(char) { return char.charCodeAt(0); });
// valores ahora tiene [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100]
```

## 18. Array.from(Array, Funcioncallback)

El método **Array.from()** es un metodo **ESTATICO** que se llama siempre desde **Array** crea una nueva instancia de Array a partir de un objeto iterable.

```
console.log(Array.from('foo'));
// Expected output: Array ["f", "o", "o"]
console.log(Array.from([1, 2, 3], x => x + x));
// Expected output: Array [2, 4, 6]
```

Sintaxis Funcioncallback es opcional y hace un mapeo al array final

### a. Array desde un Set

```
const set = new Set(['foo', 'bar', 'baz', 'foo']);
Array.from(set); // [ "foo", "bar", "baz" ]
```

### Array desde un String

```
Array.from('foo');
// [ "f", "o", "o" ]
```

### Array desde un Map

```
const map = new Map([[1, 2], [2, 4], [4, 8]]);
Array.from(map); // [[1, 2], [2, 4], [4, 8]]

const mapper = new Map([['1', 'a'], ['2', 'b']]);
Array.from(mapper.values());     // ['a', 'b'];
Array.from(mapper.keys());  // ['1', '2'];
```

### Array desde un objeto Array-like (argumentos)

```
function f() {   return Array.from(arguments); }
```


```
f(1, 2, 3); // [ 1, 2, 3 ]
```

## Usando una función de flecha y Array.from

```
// Usando una función de flecha como función
// para manipular los elementos
Array.from([1, 2, 3], x => x + x);
// [2, 4, 6]

// Generar secuencia de números
// Puesto que el array se inicializa con `undefined` en cada posición,
// el valor de `v` a continuación será `undefined`
Array.from({length: 5}, (v, i) => i);
// [0, 1, 2, 3, 4]
```

**Diferencias forEach()  map() reduce()**

forEach() ejecuta la función callback una vez por cada elemento del array; a diferencia de map() o reduce() este siempre devuelve el valor undefined y no es encadenable. El típico uso es ejecutar los efectos secundarios al final de la cadena


# Ejercicios Propuestos 5

## e) Categoria: Reducciones

### 30. Array.reduce(f.Reductora(acumulador,elemento,*Índice ,Array* ),valorInicial )
### 31.Array.reduceRight(f.Reductora(acumulador,elemento,*Índice ,Array* ),valorInicia)

El método **reduce()** ejecuta una función **reductora** sobre cada elemento de un array, devolviendo como resultado un único valor.

**reduceRight()** ejecuta la función una vez para cada elemento presente en el array, excluyendo los huecos del mismo, recibiendo cuatro argumentos:  el valor inicial (o valor de la llamada previa de funcion), el valor del elemento actual, el índice actual y el array sobre el que ocurre la iteración

**valorInicial** es el valor primero que coge el  acumulador suele ser

- 0 si estamos reduciendo a un numero
- []  si estamos reduciendo a un array
- {} si estamos reduciendo a un objeto
- "" si estamos reduciendo a un objeto

Un valor a usar como primer argumento en la primera llamada de la función callback. Si no se proporciona el valorInicial, el primer elemento del array será utilizado y saltado. Llamando a reduce() sobre un array vacío sin un valorInicial lanzará un TypeError.

```
const array1 = [1, 2, 3, 4];

// 0 + 1 + 2 + 3 + 4
const initialValue = 0;
const sumWithInitial = array1.reduce(
  (accumulator, currentValue) => accumulator + currentValue,   initialValue
);

console.log(sumWithInitial);
// Expected output: 10
```

### arr.reduce(callback(acumulador, valorActual[, índice[, array]])[, valorInicial])

La función **reductora** recibe cuatro argumentos:

1. Acumulador (acc)
2. Valor Actual (cur)
3. Índice Actual (*idx*)
4. Array (*src*)

El valor devuelto de la función **reductora** se asigna al acumulador, cuyo valor se recuerda en cada iteración de la matriz y, en última instancia, se convierte en el valor final, único y resultante.

```
[0,1,2,3,4].reduce(
                function(valorAnterior, valorActual, indice, vector){
                                return valorAnterior + valorActual;
                });
```


```
// Primera llamada
valorAnterior = 0, valorActual = 1, indice = 1

// Segunda llamada
valorAnterior = 1, valorActual = 2, indice = 2

// Tercera llamada
valorAnterior = 3, valorActual = 3, indice = 3

// Cuarta llamada
valorAnterior = 6, valorActual = 4, indice = 4

// el array sobre el que se llama a reduce siempre es el objeto [0,1,2,3,4]
// Valor Devuelto: 10
```

Reto 1: Calcular la suma de todos los elementos de un array de números.

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
console.log(sum); // Output: 15
```

Reto 2: Encontrar el número más grande en un array de números.

```
const numbers = [10, 5, 8, 3, 12];
const maxNumber = numbers.reduce((accumulator, currentValue) => Math.max(accumulator, currentValue), -Infinity);
console.log(maxNumber); // Output: 12
```

Reto 3: Convertir un array de palabras en una sola cadena de texto separada por comas.

```
const words = ['apple', 'banana', 'orange', 'grapefruit'];
const result = words.reduce((accumulator, currentValue) => accumulator + ', ' + currentValue);
console.log(result); // Output: apple, banana, orange, grapefruit
```

Reto 4: Contar la cantidad de veces que se repite cada elemento en un array.

```
const fruits = ['apple', 'banana', 'orange', 'apple', 'orange', 'banana', 'apple'];
const count = fruits.reduce((accumulator, currentValue) => {
  accumulator[currentValue] = (accumulator[currentValue] || 0) + 1;
  return accumulator;
}, {});
console.log(count);
/* Output: {   apple: 3,   banana: 2,   orange: 2 } */
```

Reto 5: Concatenar los valores de una matriz de objetos en un solo objeto.

```
const data = [
  { id: 1, name: 'John' },
  { id: 2, name: 'Alice' },
  { id: 3, name: 'Bob' }
];
const result = data.reduce((accumulator, currentValue) => {
  accumulator[currentValue.id] = currentValue.name;
  return accumulator;
}, {});

console.log(result);
// Output: {   1: 'John',   2: 'Alice',   3: 'Bob' }
```


Bootstrap
HTML
CSS
Java
Script


## e) Categoria: Cadenas-Strings

# 28. Array.join() Array.toString() Array.toLocaleString()

El método **join()** une todos los elementos de una matriz (o un objeto similar a una matriz) en una cadena y devuelve esta cadena. Es una version avanzada del toString() que solo te los puede unir con comas.

```
const elements = ['Fire', 'Air', 'Water'];
console.log(elements.join());   // Expected output: "Fire,Air,Water"
console.log(elements.join('')); // Expected output: "FireAirWater"
console.log(elements.join('-')) ; // Expected output: "Fire-Air-Water"
```

El método **toString()** devuelve una cadena de caracteres representando el array especificado y sus elementos.

```
const array1 = [1, 2, 'a', '1a'];
console.log(array1.toString()); // Expected output: "1,2,a,1a"
```