

## DWEC – Javascript Web Cliente.

JavaScript 02 – Arrays (I).....	2
Introducción.....	2
Objetos JavaScript.....	2
Propiedades de un objeto.....	2
Métodos de un objeto .....	3
Los arrays en JavaScript .....	4
Arrays de objetos .....	5
Operaciones con Arrays.....	5
length .....	5
Añadir elementos.....	5
Eliminar elementos .....	5
splice .....	6
slice .....	6
Arrays y Strings .....	6
sort.....	7
Otros métodos comunes.....	8
Functional Programming.....	9
filter.....	9
find .....	10
findIndex .....	11
every / some .....	11
map .....	11
reduce .....	11
forEach.....	14
includes .....	14
Array.from.....	14
Referencia vs Copia.....	15
Rest y Spread .....	16
Desestructuración de arrays .....	17
Los objetos Map y Set.....	18
Map.....	18
Set .....	19

# JavaScript 02 – Arrays (I)

## Introducción

### Objetos JavaScript

Los objetos son uno de los tipos de datos de JavaScript.

Los objetos se utilizan para almacenar colecciones de clave/valor (nombre/valor).

Se puede decir que un objeto JavaScript es una colección de valores con nombre.

En Javascript podemos definir cualquier variable como un objeto declarándola con **new** (NO se recomienda) o creando un *literal object* (usando notación **JSON**).

El siguiente ejemplo crea un objeto JavaScript con tres propiedades clave/valor.

Creando con *new* (no recomendado):

```
let alumno = new Object()  
alumno.nombre = 'Carlos'      // se crea la propiedad 'nombre' y se le asigna un valor  
alumno['apellidos'] = 'Pérez Ortiz'  // se crea la propiedad 'apellidos'  
alumno.edad = 19
```

Creando un *literal object* (es la forma **recomendada**). El ejemplo anterior quedaría:

```
let alumno = {  
  nombre: 'Carlos',  
  apellidos: 'Pérez Ortiz',  
  edad: 19,  
};
```

### Propiedades de un objeto

Podemos acceder a las propiedades con `.` (punto) o `[ ]`:

```
console.log(alumno.nombre)      // imprime 'Carlos'  
console.log(alumno['nombre'])   // imprime 'Carlos'  
let prop = 'nombre'  
console.log(alumno[prop])       // imprime 'Carlos'
```

Si intentamos acceder a propiedades que no existen, no se produce un error: se devuelve *undefined*:

```
console.log(alumno.ciclo)       // muestra undefined
```

Sin embargo, se genera un error si intentamos acceder a propiedades de algo que no es un objeto:

```
console.log(alumno.ciclo)       // muestra undefined  
console.log(alumno.ciclo.descrip) // se genera un ERROR
```

Para evitar ese error antes había que comprobar que existían las propiedades previas:

```
console.log(alumno.ciclo && alumno.ciclo.descrip)
```

```
// si alumno.ciclo es un objeto muestra el valor de
// alumno.ciclo.descrip. Si no muestra undefined
```

Con ES2020 (ES11) se ha incluido el operador `?.` para evitar tener que comprobar esto nosotros:

```
console.log(alumno.ciclo?.descrip)
// si alumno.ciclo es un objeto muestra el valor de
// alumno.ciclo.descrip. Si no muestra undefined
```

Podremos recorrer las propiedades de un objeto con `for...in`:

```
for (let prop in alumno) {
  console.log(prop + ': ' + alumno[prop])
}
```

Nota: En este caso no se puede aplicar el bloque de código `for...of` porque un objeto de este tipo no es iterable, sí son iterables los arrays y strings

Si el valor de una propiedad es el valor de una variable que se llama como la propiedad, no es necesario ponerlo:

```
let nombre = 'Carlos'

let alumno = {
  nombre,           // es equivalente a nombre: nombre
  apellidos: 'Pérez Ortiz',
  ...
}
```

## Métodos de un objeto

Llamamos **método de un objeto** cuando una **propiedad** de ese objeto es una **función**:

```
alumno.getInfo = function() {
  return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad + ' años'
}
```

Nota: La palabra clave **this** dentro de un método hace referencia al objeto en cuestión al que se aplica dicho método.

Y para llamarlo se hace como con cualquier otra propiedad:

```
console.log(alumno.getInfo()) // imprime 'El alumno Carlos Pérez Ortiz tiene 19 años'
```

Ojo!! No se puede utilizar *this* con sintaxis *arrow function*. Está explicado en el documento **Anexo – Uso de this en contexto**

## Algunos métodos y propiedades genéricas de objetos de JavaScript

- `constructor` Returns the function that created an object's prototype
- `keys()` Returns an Array Iterator object with the keys of an object
- `prototype` Let you to add properties and methods to JavaScript objects
- `toString()` Converts an object to a string and returns the result. Este método se usa sobrescrito.
- `valueOf()` Returns the primitive value of an object

Prueba las siguientes sentencias:

```
console.log(alumno.constructor);
console.log(Object.keys(alumno));
console.log(Object.values(alumno));
console.log(Object.entries(alumno));
console.log(alumno.toString());
alumno.edad=33;
console.log(alumno.valueOf());
console.log(Object.prototype);
console.log(Object.constructor);
```

EJERCICIO: Crea un objeto llamado **tvSamsung** con las propiedades **nombre** (“TV Samsung 42”), **categoría** (“Televisores”), **unidades** (4), **precio** (345.95) y con un método llamado **importe** que devuelve el valor total de las unidades (nº de unidades \* precio).

## Los arrays en JavaScript

Los arrays, también llamados arreglos, vectores, matrices, listas..., son un tipo de objeto que no tiene tamaño fijo, podemos añadirle elementos en cualquier momento.

Para hacer referencia a (referenciar) los elementos, se hace con un índice numérico. A diferencia de los objetos que se referencian con un nombre.

Podemos crearlos como instancias del objeto Array (No se recomienda):

```
let a = new Array()           // a = []
let b = new Array(2,4,6)      // b = [2, 4, 6]
```

Lo recomendable es crearlos usando notación JSON:

```
let a = []
let b = [2,4,6]
```

Sus elementos pueden ser de cualquier tipo, incluso podemos tener elementos de tipos distintos en un mismo array.

Si no está definido un elemento, su valor será *undefined*. Ej.:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
console.log(a[0]) // imprime 'Lunes'
console.log(a[4]) // imprime 6
a[7] = 'Juan'     // ahora a = ['Lunes', 'Martes', 2, 4, 6, , , 'Juan']
console.log(a[7]) // imprime 'Juan'
console.log(a[6]) // imprime undefined
```

Acceder a un elemento de un array que no existe no provoca un error (devuelve *undefined*), pero sí lo provoca acceder a un elemento de algo que no es un array. Con ES2020 (ES11) se ha incluido el operador **?.** para evitar tener que comprobar nosotros que sea un array:

```
console.log(alumnos?.[0])
// si alumnos es un array muestra el valor de su primer
// elemento y si no muestra undefined pero no lanza un error
```

## Arrays de objetos

Es habitual almacenar datos en arrays en forma de objetos, por ejemplo:

```
let alumnos = [  
  {  
    id: 1,  
    name: 'Carlos Pérez',  
    course: '2DAW',  
    age: 21  
  },  
  {  
    id: 2,  
    name: 'Ana García',  
    course: '2DAW',  
    age: 23  
  },  
];
```

## Operaciones con Arrays

Los arrays tienen las mismas propiedades y métodos que los objetos, y muchos más que son propios de los arrays.

Vamos a ver los principales métodos y propiedades de los arrays.

### length

Esta propiedad devuelve la longitud de un array:

```
let a = ['Lunes', 'Martes', 2, 4, 6]  
console.log(a.length) // imprime 5
```

Podemos reducir el tamaño de un array cambiando esta propiedad:

```
a.length = 3 // ahora a = ['Lunes', 'Martes', 2]
```

### Añadir elementos

Podemos añadir elementos al final de un array con el método push, o al principio con unshift:

```
let a = ['Lunes', 'Martes', 2, 4, 6]  
a.push('Juan') // ahora a = ['Lunes', 'Martes', 2, 4, 6, 'Juan']  
a.unshift(7) // ahora a = [7, 'Lunes', 'Martes', 2, 4, 6, 'Juan']
```

### Eliminar elementos

Podemos borrar el elemento del final de un array con pop, o del principio con shift. Ambos métodos devuelven el elemento que hemos borrado:

```
let a = ['Lunes', 'Martes', 2, 4, 6]  
let ultimo = a.pop() // ahora a = ['Lunes', 'Martes', 2, 4] y ultimo = 6  
let primero = a.shift() // ahora a = ['Martes', 2, 4] y primero = 'Lunes'
```

## splice

Permite eliminar elementos de cualquier posición del array y/o insertar otros en su lugar. Devuelve un array con los elementos eliminados. Sintaxis:

```
Array.splice(posicion, num. de elementos a eliminar, 1º elemento a insertar, 2º elemento a insertar, 3º...)
```

### Ejemplo:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let borrado = a.splice(1, 3) // ahora a = ['Lunes', 6] y borrado = ['Martes', 2, 4]
a = ['Lunes', 'Martes', 2, 4, 6]
borrado = a.splice(1, 0, 45, 56) // ahora a = ['Lunes', 45, 56, 'Martes', 2, 4, 6] y borrado = []
a = ['Lunes', 'Martes', 2, 4, 6]
borrado = a.splice(1, 3, 45, 56) // ahora a = ['Lunes', 45, 56, 6] y borrado = ['Martes', 2, 4]
```

EJERCICIO: Guarda en un array la lista de la compra con Peras, Manzanas, Kiwis, Plátanos y Mandarinas. Haz lo siguiente con splice:

- Elimina las manzanas (debe quedar Peras, Kiwis, Plátanos y Mandarinas)
- Añade detrás de los Plátanos, Naranjas y Sandía. (Debe quedar: Peras, Kiwis, Plátanos, Naranjas, Sandía y Mandarinas)
- Quita los Kiwis y pon en su lugar Cerezas y Nísperos. (Debe quedar: Peras, Cerezas, Nísperos, Plátanos, Naranjas, Sandía y Mandarinas)

## slice

Devuelve un subarray con los elementos indicados, pero sin modificar el array original. Sería como hacer un substr pero de un array en vez de una cadena.

Sintaxis: `Array.slice(posicion, num. de elementos a devolver)`

### Ejemplo:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let subArray = a.slice(1, 3) // ahora a=['Lunes', 'Martes', 2, 4, 6] y subArray=['Martes', 2, 4]
```

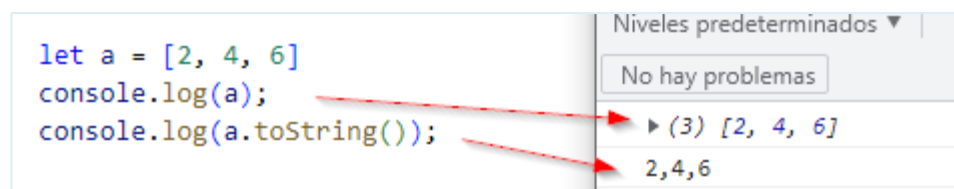
Es muy útil para hacer una copia de un array:

```
let a = [2, 4, 6]
let copiaDeA = a.slice() // ahora ambos arrays contienen lo mismo pero son diferentes arrays
```

## Arrays y Strings

Cada objeto, y los arrays son un tipo de objeto, tienen definido el método `.toString()` que lo convierte en una cadena. Este método es llamado automáticamente cuando, por ejemplo, queremos mostrar un array por la consola.

En el caso de los arrays, esta función devuelve una cadena con los elementos del array separados por coma.



Además, podemos convertir los elementos de un array a una cadena con `.join()` especificando el carácter separador de los elementos. Ej.:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let cadena = a.join('-') // cadena = 'Lunes-Martes-2-4-6'
```

El método `.join()` es el contrario del `.split()` que convierte una cadena en un array. Ej.:

```
let notas = '5-3.9-6-9.75-7.5-3'
let arrayNotas = notas.split('-') // arrayNotas = ['5', '3.9', '6', '9.75', '7.5', '3']
let cadena = 'Que tal estás'
let arrayPalabras = cadena.split(' ') // arrayPalabras = ['Que', 'tal', 'estás']
let arrayLetras = cadena.split('') // arrayLetras = ['Q','u','e',' ','t','a','l',' ','e','s','t','á','s']
```

## sort

Ordena **alfabéticamente** los elementos del array.

OJO!! El array original queda modificado con el nuevo orden.

```
let a = ['hola', 'adios', 'Bien', 'Mal', 2, 5, 13, 45]
let b = a.sort() // b = [13, 2, 45, 5, "Bien", "Mal", "adios", "hola"]
```

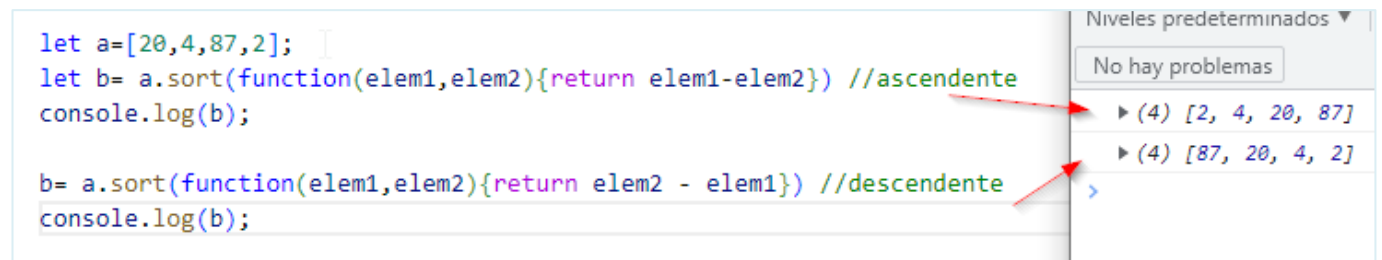
Si no se especifica otra cosa, el orden que se sigue es el de los códigos ascii, por lo que los dígitos numéricos van antes que las letras, y las mayúsculas antes que las minúsculas.

También podemos pasarle una función que le indique cómo ordenar. Esta función debe devolver un valor negativo si el primer elemento es menor, positivo si es mayor, o 0 si son iguales.

Ejemplo: ordenar un array de cadenas sin tener en cuenta si son mayúsculas o minúsculas:

```
let a = ['hola', 'adios', 'Bien', 'Mal']
let b = a.sort(function(elem1, elem2) {
  if (elem1.toLocaleLowerCase() < elem2.toLocaleLowerCase()) return -1;
  if (elem1.toLocaleLowerCase() > elem2.toLocaleLowerCase()) return 1;
  if (elem1.toLocaleLowerCase() = elem2.toLocaleLowerCase()) return 0;
});
// b = ["adios", "Bien", "hola", "Mal"]
```

También se utiliza para ordenar números, tanto de forma ascendente como descendente:



The screenshot shows a code editor with the following JavaScript code:

```
let a=[20,4,87,2];
let b= a.sort(function(elem1,elem2){return elem1-elem2}) //ascendente
console.log(b);

b= a.sort(function(elem1,elem2){return elem2 - elem1}) //descendente
console.log(b);
```

On the right, a console output window titled "Niveles predeterminados" shows the results of the sorting operations:

- (4) [2, 4, 20, 87] (indicated by a red arrow from the ascending sort line)
- (4) [87, 20, 4, 2] (indicated by a red arrow from the descending sort line)

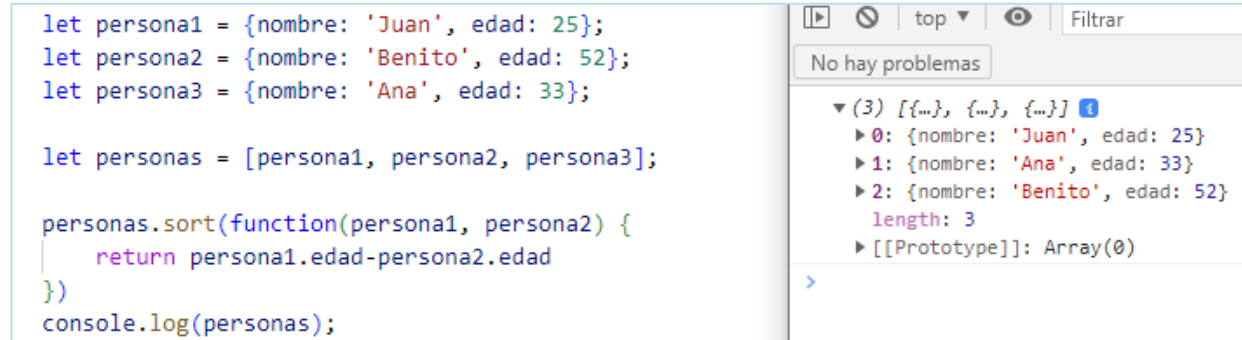
Es frecuente utilizar esta función es para ordenar arrays de objetos. Por ejemplo, si tenemos un objeto *persona* con los campos *nombre* y *edad*, para ordenar un array de objetos persona por su edad haremos:

```
let persona1 = {nombre: 'Juan', edad: 25};
```

```
let persona2 = {nombre: 'Benito', edad: 52};
let persona3 = {nombre: 'Ana', edad: 33};

let personas = [persona1, persona2, persona3];

let personasOrdenado = personas.sort(function(persona1, persona2) {
  return persona1.edad-persona2.edad
});
```



```
let persona1 = {nombre: 'Juan', edad: 25};
let persona2 = {nombre: 'Benito', edad: 52};
let persona3 = {nombre: 'Ana', edad: 33};

let personas = [persona1, persona2, persona3];

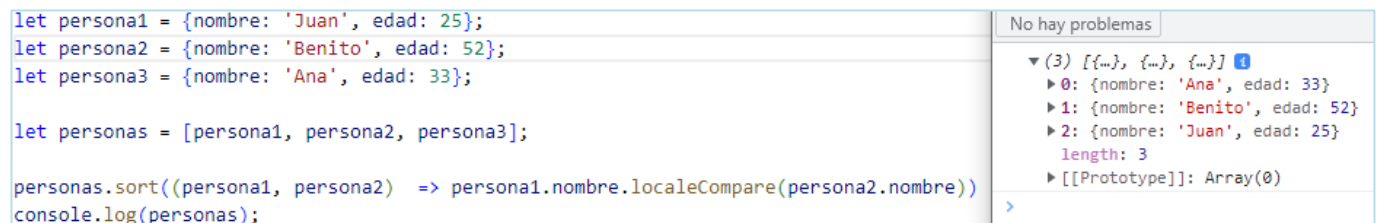
personas.sort(function(persona1, persona2) {
  return persona1.edad-persona2.edad
});
console.log(personas);
```

Usando *arrow functions* quedaría más sencillo:

```
let personasOrdenado = personas.sort((persona1, persona2) => persona1.edad-persona2.edad)
```

Si lo que queremos es ordenar por un campo de texto podemos usar la función *toLocaleCompare*:

```
let personasOrdenado = personas.sort((persona1, persona2) => persona1.nombre.localeCompare(persona2.nombre))
```



```
let persona1 = {nombre: 'Juan', edad: 25};
let persona2 = {nombre: 'Benito', edad: 52};
let persona3 = {nombre: 'Ana', edad: 33};

let personas = [persona1, persona2, persona3];

personas.sort((persona1, persona2) => persona1.nombre.localeCompare(persona2.nombre))
console.log(personas);
```

**EJERCICIO:** Haz una función que ordene las notas de un array pasado como parámetro. Si le pasamos [4,8,3,10,5] debe devolver [3,4,5,8,10]. Pruébalo en la consola

## Otros métodos comunes

Otros métodos que se usan a menudo con arrays son:

- `.concat()`: concatena arrays
 

```
let a = [2, 4, 6]
let b = ['a', 'b', 'c']
let c = a.concat(b) // c = [2, 4, 6, 'a', 'b', 'c']
```
- `.reverse()`: invierte el orden de los elementos del array
 

```
let a = [2, 4, 6]
let b = a.reverse() // b = [6, 4, 2]
```
- `.indexOf()`: devuelve la primera posición del elemento pasado como parámetro o -1 si no se encuentra en el array



- `.lastIndexOf()`: devuelve la última posición del elemento pasado como parámetro o -1 si no se encuentra en el array

```
let a = [2, 4, 6, 4]
console.log(a.indexOf(4));           // 1
console.log(a.lastIndexOf(4));       // 3
console.log(a.lastIndexOf('4'));     // -1
```

## Functional Programming

Se trata de un paradigma de programación (una forma de programar) donde se intenta que el código se centre más en qué debe hacer una función que en cómo debe hacerlo.

El ejemplo más claro es que intenta evitar los bucles *for* y *while* sobre arrays o listas de elementos.

Normalmente, cuando hacemos un bucle es para recorrer la lista y realizar alguna acción con cada uno de sus elementos. Lo que hace *functional programming* es que a la función que debe hacer eso, además de pasarle como parámetro la lista sobre la que debe actuar, se le pasa como segundo parámetro la función que debe aplicarse a cada elemento de la lista.

Desde la versión 5.1 javascript incorpora métodos de *functional programming* en el lenguaje, especialmente para trabajar con arrays:

### filter

Devuelve un nuevo array con los elementos que cumplen determinada condición del array al que se aplica.

Su parámetro es una función, habitualmente anónima, que va interactuando con los elementos del array.

Esta función recibe como primer parámetro el elemento actual del array (sobre el que debe actuar).

Opcionalmente puede tener como segundo parámetro su índice y como tercer parámetro el array completo.

La función debe devolver **true** para los elementos que se incluirán en el array a devolver como resultado, y **false** para el resto.

Ejemplo: dado un array con notas, devolver un array con las notas de los aprobados.

Usando programación *imperativa* (la que se centra en *cómo se deben hacer las cosas*) sería algo como:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = []
for (let i = 0; i < arrayNotas.length; i++) {
  let nota = arrayNotas[i]
  if (nota >= 5) {
    aprobados.push(nota)
  }
} // aprobados = [5.2, 6, 9.75, 7.5]
```

Usando *functional programming* (la que se centra en *qué resultado queremos obtener*) sería:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = arrayNotas.filter(function(nota) {
  if (nota >= 5) {
    return true
  }
})
```

```
} else {  
  return false  
}  
}) // aprobados = [5.2, 6, 9.75, 7.5]
```

Podemos refactorizar esta función para que sea más compacta:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]  
let aprobados = arrayNotas.filter(function(nota) {  
  return nota >= 5 // nota >= 5 se evalúa a 'true' si es cierto, o 'false' si no lo es  
})  
// aprobados = [5.2, 6, 9.75, 7.5]
```

Y usando funciones tipo flecha la sintaxis queda mucho más simple:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]  
let aprobados = arrayNotas.filter(nota => nota >= 5)  
// aprobados = [5.2, 6, 9.75, 7.5]
```

Las 7 líneas del código usando programación *imperativa* quedan reducidas a sólo una.

EJERCICIO: Dado un array con los días de la semana obtén todos los días que empiezan por 'M'

## find

Como *filter* pero NO devuelve un **array**, devuelve el primer **elemento** que cumpla la condición (o *undefined* si no la cumple ninguno). Ejemplo:

```
let arrayNotas = [4, 5.2, 3.9, 6, 9.75, 7.5, 3]  
let primerAprobado = arrayNotas.find(nota => nota >= 5) // primerAprobado = 5.2
```

Este método tiene más sentido con objetos. Por ejemplo, si queremos encontrar un coche de color rojo dentro de un array llamado coches cuyos elementos son objetos con un campo 'color', haremos:

```
let coches = [  
  {  
    "color": "morado",  
    "tipo": "berlina",  
    "capacidad": 7  
  },  
  {  
    "color": "rojo",  
    "tipo": "camioneta",  
    "capacidad": 5  
  },  
  {  
    "color": "rojo",  
    "tipo": "furgón",  
    "capacidad": 7  
  }  
]
```

```
let cocheBuscado = coches.find(coche => coche.color === 'rojo') // devolverá el
objeto completo del primer elemento que cumpla la condición.
```

Si queremos que nos devuelva el objeto en la posición “1” dentro del array, haremos:

```
let cocheBuscado=coches.find((coche,indice) => indice ===1); // devolverá el objeto coches[1]
```

EJERCICIO: Dado un array con los días de la semana obtén el primer día que empieza por ‘M’

## findIndex

Como *find* pero, en vez de devolver el elemento, devuelve su posición (-1 si ningún elemento cumple la condición).

```
let cocheBuscado = coches.findIndex(coche => coche.color === 'rojo') // devolverá 1
```

En el ejemplo de los coches el valor devuelto sería 1, ya que el segundo elemento cumple la condición.

Al igual que el anterior, *findIndex* tiene más sentido con arrays de objetos.

EJERCICIO: Dado un array con los días de la semana, obtén la posición en el array del primer día que empieza por ‘M’.

## every / some

- **every** devuelve **true** si **TODOS** los **elementos** del array **cumplen la condición** y **false** en caso contrario.
- **some** devuelve **true** si **ALGÚN** elemento del array **cumple la condición**. Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let todosAprobados = arrayNotas.every(nota => nota >= 5) // false
let algunAprobado = arrayNotas.some(nota => nota >= 5) // true
```

EJERCICIO: Dado un array con los días de la semana indica si algún día empieza por ‘S’. Dado un array con los días de la semana indica si todos los días acaban por ‘s’

## map

**map** permite modificar cada elemento de un array y devuelve un nuevo array con los elementos del original modificados.

Ejemplo: queremos subir un 10% cada nota:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3];
let arrayNotasSubidas = arrayNotas.map(nota => nota + nota * 0.1);
```

EJERCICIO: Dado un array con los días de la semana devuelve otro array con los días en mayúsculas.

## reduce

El método **reduce**:

- Ejecuta una función tipo **callback** para cada elemento del array, y devuelve **un único valor calculado a partir de los elementos del array**, este **valor es el resultado acumulado de las llamadas a la función**.
- No se ejecuta la función para elementos vacíos del array.

- No se cambia el contenido del array original.

Sintaxis: `array.reduce(function(total, currentValue, currentIndex, vector), initialValue)`

Hay dos parámetros: una **función** e **initialValue** (este segundo parámetro es opcional).

La función admite 4 parámetros:

- **total o valorAnterior** (obligatorio), contiene el valor devuelto por la función en cada llamada.
- **currentValue** (obligatorio), contiene el valor del elemento actual.
- **currentIndex** (opcional), contiene el índice del elemento actual.
- **vector** (opcional), es el array al que pertenece el elemento actual.

La primera vez que se llama la función, **valorAnterior** y **valorActual** pueden tener uno de dos posibles valores:

- Si se suministró un **valorInicial** al llamar a **reduce**, entonces **valorAnterior** será igual al **valorInicial** y **valorActual** será igual al primer elemento del array.
- Si no se proporcionó un **valorInicial**, entonces **valorAnterior** será igual al primer valor en el array y **valorActual** será el segundo.

Ejemplo: queremos obtener la suma de las notas de un array:

```
let arrayNotas = [4,7,5];
let suma=0;

suma= arrayNotas.reduce((total, valor) => total+= valor, 0);
console.log(suma); // obtiene 16

// Ahora la misma función sin ser tipo flecha
suma= arrayNotas.reduce((total, valor) => {
    return total+= valor
}, 0);
console.log(suma); // obtiene 16
```

La función segunda permite añadir fácilmente otras funcionalidades

Para observar cómo funciona **reduce()**, estudia el siguiente caso para sumar el array de notas. Se ha añadido el parámetro del índice a la función callback.

Si añadimos un **valorInicial**, hace un recorrido por el índice cero. Si este valor es cero, el resultado es el mismo que sin no ponemos **valorInicial**.

```
let arrayNotas = [4,7,5];
let suma=0;

// Misma función para sumar con el parámetro índice
suma= arrayNotas.reduce((total, valor, i) => {
    console.log('Valor: ' + valor);
    console.log('Índice: ' + i);
    return total+= valor;
}, 0);
console.log('Resultado: ' + suma); // obtiene 16
```

```
let arrayNotas = [4,7,5];
let suma=0;

// Misma función para sumar con el parámetro índice
suma= arrayNotas.reduce((total, valor, i) => {
    console.log('Valor: ' + valor);
    console.log('Índice: ' + i);
    return total+= valor;
}, 3);
console.log('Resultado: ' + suma); // obtiene 16
```

En el siguiente ejemplo hay dos casos. Entre el caso de la izda y el de la derecha solo cambia el valor del parámetro opcional **initialValue** que en este caso vale 3, por lo que el resultado cambia.

```

let arrayNotas = [4,7,5];
let suma=0;

// Ahora la misma función sin ser tipo flecha
suma= arrayNotas.reduce((total, valor, i) => {
  console.log('Valor: ' + valor);
  console.log('Índice: ' + i);
  return total+= valor;
},);
console.log('Resultado: ' + suma); // obtiene 16

```

```

let arrayNotas = [4,7,5];
let suma=0;

// Ahora la misma función sin ser tipo flecha
suma= arrayNotas.reduce((total, valor, i) => {
  console.log('Valor: ' + valor);
  console.log('Índice: ' + i);
  return total+= valor;
}, 3);
console.log('Resultado: ' + suma); // obtiene 19

```

Otros ejemplos similares de suma de notas:

```

let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3];
let sumaNotas = arrayNotas.reduce((total,nota) => total += nota, 30) // total = 65.35
sumaNotas = arrayNotas.reduce((total,nota) => total += nota) // total = 35.35

```

Podemos tener la función declarada y utilizarla con reduce(). Ejemplo:

```

function suma(a, b) {
  return a + b;
}

const numeros = [10, 20, 30, 40, 50, 60, 70, 80, 90];
const resultado = numeros.reduce(suma);
console.log(resultado); // 450

```

Otro ejemplo: queremos obtener la nota más alta:

```

let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let maxNota = arrayNotas.reduce((max,nota) => nota > max ? nota : max) // max = 9.75

```

Ejemplo muy útil: Integrar un array a partir de varios arrays utilizando el método array.concat()

```

let integrado = [[0,1], [2,3], [4,5]].reduce(function(a,b) {
  return a.concat(b);
});
// integrado es [0, 1, 2, 3, 4, 5]

```

Ejemplo clarificador de la idea: Cadena de montaje, se van añadiendo piezas a la cadena de texto inicial.

```

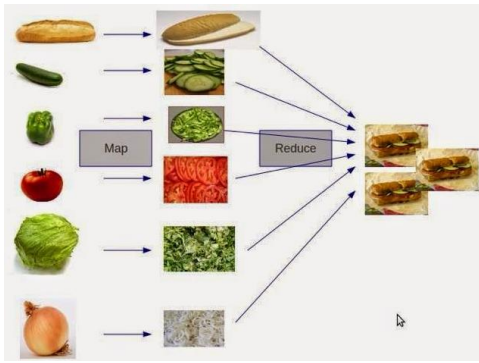
const partesDelCoche = ["asientos", "volante", "puertas", "ruedas", "pintura metalizada"];

const coche = partesDelCoche.reduce(function (valorAnterior, valorActual) {
  return `${valorAnterior} ${valorActual}`;
}, "Mi coche tiene: ");

console.log(coche);
//Mi coche tiene: asientos, volante, puertas, ruedas, pintura metalizada,

```

Una idea del funcionamiento de map y reduce sería: Tenemos un “array” de verduras al que le aplicamos una función *map* para que las corte y al resultado le aplicamos un *reduce* para que obtenga un valor (el sandwich) con todas ellas.



EJERCICIO: Dado el array de notas visto anteriormente, usar un método para que devuelva la nota media.

## forEach

Es el método más general de los que hemos visto. No devuelve nada, sino que permite realizar algo con cada elemento del array.

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
arrayNotas.forEach((nota, indice, arrayCompleto) => {
  console.log('El elemento de la posición ' + indice + ' es: ' + nota)
})
```

Los 3 argumentos de la función son: Valor de cada elemento del array, índice del array, y contenido completo del array.

## includes

Devuelve **true** si el array incluye el elemento pasado como parámetro. Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
arrayNotas.includes(7.5) // true
```

EJERCICIO: Dado un array con los días de la semana indica si algún día es el 'Martes'

## Array.from

Devuelve un array a partir de otro al que se puede aplicar una función de transformación (es similar a *map*). Ejemplo: queremos subir un 10% cada nota:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let arrayNotasSubidas = Array.from(arrayNotas, nota => nota * 1.1)
```

Puede usarse para hacer una copia de un array, igual que *slice*:

```
let arrayA = [5.2, 3.9, 6, 9.75, 7.5, 3];
let arrayB = Array.from(arrayA);
```

También se utiliza mucho para convertir colecciones en arrays y así poder usar los métodos de arrays que hemos visto con las colecciones.

Por ejemplo, si queremos mostrar por consola cada párrafo de la página que comience por la palabra 'Sí' en primer lugar obtenemos todos los párrafos con:

```
let parrafos = document.getElementsByTagName('p')
```

Esto nos devuelve una colección con todos los párrafos de la página (lo veremos más adelante al ver DOM). Podríamos hacer un **for** para recorrer la colección y mirar los que empiecen por lo indicado, pero no podemos aplicarle los métodos vistos aquí porque son sólo para arrays, así que hacemos:

```
let parrafos = document.getElementsByTagName('p');
let arrayParrafos = Array.from(parrafos);
// y ya podemos usar los métodos que queramos:
arrayParrafos.filter(parrafo => parrafo.textContent.startsWith('Si'))
    .forEach(parrafo => alert(parrafo.textContent))
```

**IMPORTANTE:** desde este momento se han acabado los bucles *for* en nuestro código para trabajar con arrays. Usaremos siempre estas funciones!!!

Existen otros métodos para utilizar con arrays, se pueden ver en:

[https://www.w3schools.com/jsref/jsref\\_obj\\_array.asp](https://www.w3schools.com/jsref/jsref_obj_array.asp)

## Referencia vs Copia

Cuando copiamos una variable de tipo *boolean*, *string* o *number* (o se pasa esa variable como parámetro a una función), se hace una copia de la misma. Por lo que, si se modifica, la variable original no es modificada. Ej.:

```
let a = 54
let b = a      // a = 54 b = 54
b = 86        // a = 54 b = 86
```

Sin embargo, al copiar objetos (y los arrays son un tipo de objeto), la nueva variable apunta a la misma posición de memoria que la antigua por lo que los datos de ambas son los mismos:

```
let a = [54, 23, 12]
let b = a      // a = [54, 23, 12] b = [54, 23, 12]
b[0] = 3       // a = [3, 23, 12] b = [3, 23, 12]
let fecha1 = new Date('2022-09-23')
let fecha2 = fecha1      // fecha1 = '2022-09-23' fecha2 = '2022-09-23'
fecha2.setFullYear(1999) // fecha1 = '1999-09-23' fecha2 = '1999-09-23'
```

Si queremos obtener una copia de un array que sea independiente del original podemos obtenerla con `slice` o con `Array.from`:

```
let a = [2, 4, 6]
let copiaDeA = a.slice() // ahora ambos arrays contienen lo mismo, pero son diferentes
let otraCopiaDeA = Array.from(a)
```

En el caso de objetos, es algo más complejo. ES6 incluye **Object.assign**, que hace una copia de un objeto:

```
let a = {id:2, name: 'object 2'}
let copiaDeA = Object.assign({}, a) //ahora ambos objetos contienen lo mismo, pero son diferentes
```

Sin embargo, si el objeto tiene como propiedades otros objetos, éstos se continúan pasando por referencia. En ese caso lo más sencillo sería utilizar `objetoCopia = JSON.stringify(objetoOriginal);`

**JSON** es un es un formato de texto para almacenar y transportar datos.

JavaScript tiene una función integrada para convertir cadenas JSON en objetos JavaScript: `JSON.parse()`

JavaScript tiene una función integrada para convertir un objeto en una cadena JSON: `JSON.stringify()`

```
let a = {id: 2, name: 'object 2', address: {street: 'C/ Ajo', num: 3} }
let copiaDeA = JSON.parse(JSON.stringify(a)) // ahora ambos objetos contienen lo mismo pero
son diferentes
```

```
'use strict';
let a = {id: 2, name: 'Federico', address: {street: 'C/ Ajo', num: 3} }
let b = a;
let copia1= Object.assign({}, a);

//solución:
// ahora ambos objetos contienen lo mismo pero son diferentes
let copiaDeA = JSON.parse(JSON.stringify(a))

console.log((b));
a.name= 'Ana';
a.address.street = 'Avda. Alemania'
console.log(a);
console.log(b);
console.log(copia1);
console.log(copiaDeA);
```

The console output shows the state of the objects after mutations. Red arrows indicate the following:

- `console.log((b));` points to the first log: `{id: 2, name: 'Federico', address: {...}}`
- `a.name= 'Ana';` points to the second log: `{id: 2, name: 'Ana', address: {...}}`
- `a.address.street = 'Avda. Alemania'` points to the third log: `{id: 2, name: 'Ana', address: {...}}`
- `console.log(a);` points to the fourth log (expanded): `{id: 2, name: 'Federico', address: {street: 'Avda. Alemania', num: 3}}`
- `console.log(b);` points to the fifth log (expanded): `{id: 2, name: 'Federico', address: {street: 'C/ Ajo', num: 3}}`
- `console.log(copia1);` points to the sixth log (expanded): `{id: 2, name: 'Federico', address: {street: 'C/ Ajo', num: 3}}`
- `console.log(copiaDeA);` points to the seventh log (expanded): `{id: 2, name: 'Federico', address: {street: 'C/ Ajo', num: 3}}`

**EJERCICIO:** Dado el array `arr1` con los días de la semana haz un array `arr2` que sea igual al `arr1`. Elimina de `arr2` el último día y comprueba qué ha pasado con `arr1`. Repita la operación con un array llamado `arr3` pero que crearás haciendo una copia de `arr1`.

## Rest y Spread

También podemos copiar objetos usando *rest* y *spread*.

Permiten extraer a parámetros los elementos de un array o string (*spread*) o convertir en un array un grupo de parámetros (*rest*). El operador de ambos es `...` (3 puntos).

**Rest** se utiliza para convertir en un array un grupo de parámetros (*rest*). El operador es `...` (3 puntos).

Para usar *rest* como parámetro de una función debe ser siempre el último parámetro:

Ejemplo: queremos hacer una función que calcule la media de las notas que se le pasen como parámetro y que no sabemos cuántas son. Para llamar a la función haremos:

```
console.log(notaMedia(3.6, 6.8))
console.log(notaMedia(5.2, 3.9, 6, 9.75, 7.5, 3))
```

La función `notaMedia` convertirá los parámetros recibidos en un array usando *rest*:

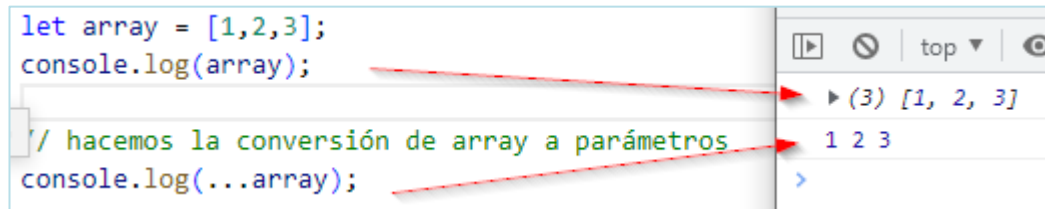
```
function notaMedia(...notas) {
```



```
let total = notas.reduce((total,nota) => total += nota)
return total/notas.length
}
```

**Spread** permite pasar como parámetros independientes los elementos de un array o string. El operador también es ... (3 puntos).

Si lo que queremos es convertir un array en un grupo de elementos haremos *spread*.



```
let array = [1,2,3];
console.log(array);

// hacemos la conversión de array a parámetros
console.log(...array);
```

Por ejemplo, el objeto *Math* proporciona métodos para trabajar con números como *.max* que devuelve el máximo de los números pasados como parámetro. Para saber la nota máxima en vez de utilizar *.reduce* como hicimos en el ejemplo anterior, podemos hacer:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3];
let maximo = Math.max(arrayNotas); //maximo = NaN
// si hacemos Math.max(arrayNotas) devuelve NaN porque arrayNotas es un array y no un número

// hacemos la conversión de array a parámetros
let maximoSpread = Math.max(...arrayNotas); // maxNota = 9.75
```

Estas funcionalidades nos ofrecen otra manera de copiar objetos (pero sólo a partir de ES-2018):

```
let a = {id: 2, name: 'object 2'}
let copiaDeA = { ...a} // ahora ambos objetos contienen lo mismo pero son diferentes
```

```
let b = [2, 8, 4, 6]
let copiaDeB = [ ...b ] // ahora ambos objetos contienen lo mismo pero son diferentes
```

## Desestructuración de arrays

Similar a *rest* y *spread*, permiten extraer los elementos del array directamente a variables y viceversa. Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let [primera, segunda, tercera] = arrayNotas // primera = 5.2, segunda = 3.9, tercera = 6
let [primera, , , cuarta] = arrayNotas // primera = 5.2, cuarta = 9.75
let [primera, ...resto] = arrayNotas // primera = 5.2, resto = [3.9, 6, 9.75, 3]
```

También se pueden asignar valores por defecto:

```
let preferencias = ['Javascript', 'NodeJS']
let [lenguaje, backend = 'Laravel', frontend = 'VueJS'] = preferencias
// lenguaje = 'Javascript', backend = 'NodeJS', frontend = 'VueJS'
```

La desestructuración también funciona con objetos. Es normal pasar un objeto como parámetro para una función, pero si sólo nos interesan algunas propiedades del mismo, podemos desestructurarlo:

```
const miProducto = {
  id: 5,
  name: 'TV Samsung',
  units: 3,
  price: 395.95
};

// Se puede abreviar: function muestraNombre({name, units}) {
function muestraNombre({name: name, units: units}) {
  console.log('Del producto ' + name + ' hay ' + units + ' unidades')
}
muestraNombre(miProducto); //Del producto TV Samsung hay 3 unidades
```

También podemos asignar valores por defecto:

```
function muestraNombre({name, units = 0}) {
  console.log('Del producto ' + name + ' hay ' + units + ' unidades')
}

muestraNombre({name: 'USB Kingston'});
// mostraría: Del producto USB Kingston hay 0 unidades
```

## Los objetos Map y Set

En el ámbito de JavaScript, a menudo los desarrolladores pasan mucho tiempo decidiendo la estructura de datos correcta que se usará. Esto se debe a que elegir la estructura de datos correcta puede facilitar la manipulación de esos datos posteriormente, con lo cual se puede ahorrar tiempo y facilitar la comprensión del código.

Las dos estructuras de datos predominantes para almacenar conjuntos de datos son los Object (Objetos) y Array (un tipo de objeto).

Los desarrolladores utilizan Object para almacenar pares clave-valor y Array para almacenar listas indexadas.

Sin embargo, para dar más flexibilidad a los desarrolladores, en la especificación ECMAScript 2015 se introdujeron dos nuevos tipos de objetos iterables:

- los Map, que son grupos ordenados de pares clave-valor.
- los Set, que son grupos de valores únicos.

### Map

El objeto **Map** es una colección de parejas de [clave,valor].

En cambio, un objeto en Javascript es un tipo particular de *Map* en el que las claves sólo pueden ser texto o números.

Se puede acceder a una propiedad con `.` o **[propiedad]**. Ejemplo:

```
let persona = {
  nombre: 'John',
  apellido: 'Doe',
```

```
    edad: 39
  }
  console.log(persona.nombre)      // John
  console.log(persona['nombre'])    // John
```

Un *Map* permite que la clave sea cualquier cosa (array, objeto, ...).

Más información en [MDN](#) o cualquier otra página.

## Set

El objeto **Set** es como un *Map*, pero que no almacena los valores, almacena solo la clave. Podemos verlo como una colección que no permite duplicados.

Tiene la propiedad **size** que devuelve su tamaño y los métodos **.add** (añade un elemento), **.delete** (lo elimina) o **.has** (indica si el elemento pasado se encuentra o no en la colección) y también podemos recorrerlo con **.forEach**.

Una forma sencilla de eliminar los duplicados de un array es crear con él un *Set*:

```
let ganadores = ['Márquez', 'Rossi', 'Márquez', 'Lorenzo', 'Rossi', 'Márquez',
'Márquez']
let ganadoresNoDuplicados = new Set(ganadores)    // {'Márquez', 'Rossi', 'Lorenzo'}

// o si lo queremos en un array:

ganadoresNoDuplicados = Array.from(new Set(ganadores)) // ['Márquez', 'Rossi', 'Lorenzo']
```