Métodos de Arrays en JavaScript

En JavaScript los arrays (`Array`) disponen de numerosos métodos integrados que facilitan su manipulación. A continuación se presenta una guía completa, agrupando los métodos por funcionalidad (transformación, iteración, búsqueda, mutación, etc.). Para cada método se ofrece descripción, sintaxis, ejemplo de uso y observaciones (como si muta el array original). También se destacan diferencias clave entre métodos similares y advertencias comunes.

Transformación e Iteración

`map()`

Descripción: Crea un nuevo arreglo aplicando una función a cada elemento del original[1]. No modifica el array original.
Sintaxis: `nuevoArray = arreglo.map(función(elemento[, índice[, arreglo]])[, thisArg])`
Ejemplo:

```
const numeros = [1, 2, 3];
const dobles = numeros.map(x => x * 2);
console.log(dobles); // [2, 4, 6]
```

Notas: `map()` **no muta** el array original[2] y devuelve siempre un nuevo array de la misma longitud. Es útil para transformar datos. A diferencia de `forEach()`, devuelve el array transformado (por eso es encadenable)[3].

`filter()`

Descripción: Crea un nuevo arreglo con todos los elementos que cumplen una condición dada (la función "predicado" devuelve `true`)[4]. No modifica el array original.
Sintaxis: `nuevoArray = arreglo.filter(función(elemento[, índice[, arreglo]])[, thisArg])`
Ejemplo:

```
const numeros = [1, 2, 3, 4, 5];
const pares = numeros.filter(x => x % 2 === 0);
console.log(pares); // [2, 4]
```

Notas: `filter()` **no muta** el arreglo original[2]. Si ningún elemento cumple la condición, devuelve un array vacío. A diferencia de `find()`, que devuelve el *primer* elemento que cumple la condición (o `undefined`), `filter()` devuelve **todos** los elementos que la cumplen[5][6] (posiblemente ninguno). Por ello `filter()` siempre retorna un array (vacío si no hay coincidencias), mientras que `find()` retorna un único valor o `undefined`[6][4].

`reduce()` y `reduceRight()`

Descripción: Aplica una función reductora acumulativa a los elementos del arreglo, retornando un solo valor. `reduce()` recorre de izquierda a derecha; `reduceRight()` de derecha a izquierda[7][8].
Sintaxis:

```
resultado = arreglo.reduce((acumulador, valorActual[, índice, arreglo]) => {
... }, valorInicial);
resultado = arreglo.reduceRight((acumulador, valorActual[, índice, arreglo])
=> { ... }, valorInicial);
```

Ejemplo:

```
const numeros = [1, 2, 3, 4];
const suma = numeros.reduce((acc, cur) => acc + cur, 0);
console.log(suma); // 10
```

Notas: Si no se proporciona `valorInicial`, `reduce()` toma el primer elemento como acumulador inicial y empieza en el segundo elemento. Llamar a `reduce()` en un array vacío sin valor inicial lanza error[9]. Para evitarlo siempre es recomendable pasar `valorInicial`. A diferencia de `map` o `filter`, devuelve un único valor (que puede ser cualquier tipo). Los huecos (elementos vacíos) se saltan.

`flat()` y `flatMap()`

Descripción:
- `flat(depth)`: Aplana subarreglos hasta la profundidad indicada, devolviendo un nuevo arreglo[10]. Por defecto `depth = 1`.
- `flatMap(fn)`: Aplica primero `map()` con la función `fn` a cada elemento y luego aplanar el resultado un nivel. Es equivalente a `arr.map(fn).flat(1)`.

Sintaxis: `nuevoArray = arreglo.flat([profundidad]);`
Ejemplo:

```
const anidado = [1, 2, [3, 4, [5]]];
console.log(anidado.flat());   // [1, 2, 3, 4, [5]]
console.log(anidado.flat(2));  // [1, 2, 3, 4, 5]
const palabras = ["hola mundo", "buenos dias"];
const letras = palabras.flatMap(p => p.split(""));
console.log(letras);
// ["h","o","l","a"," ","m",..., "s"]
```

Notas: Ambos métodos **no mutan** el arreglo original; retornan un nuevo arreglo[10]. `flat()` elimina ranuras vacías (gaps) en arrays dispersos. Se añade en ES2019 (para navegadores modernos).

`forEach()`

Descripción: Ejecuta una función para cada elemento del arreglo, en orden ascendente[11]. No devuelve nada útil (retorna `undefined`).
Sintaxis: `arreglo.forEach((elemento, índice, arreglo) => { ... }[, thisArg]);`
Ejemplo:

```
const frutas = ["manzana", "banano", "cereza"];
frutas.forEach((fruta, idx) => {
  console.log(`${idx}: ${fruta}`);
});
```

Salida:

```
0: manzana
1: banano
2: cereza
```

Notas: `forEach()` **no muta** el arreglo original[12]. Se usa para efectos secundarios (p.ej. imprimir, modificar otras variables) en cada elemento. No puede detenerse (no admite `break` ni `return` global); para ello es mejor usar un bucle `for` o `for..of`. A diferencia de `map`, no devuelve un nuevo array ni es encadenable[13].

Búsqueda y Filtrado

`find()` y `findIndex()`

Descripción:
- `find(fn)`: Devuelve el **primer elemento** que cumpla la condición (`fn(elemento)` retorna `true`), o `undefined` si ninguno lo cumple[6].
- `findIndex(fn)`: Devuelve el índice del primer elemento que cumpla la condición, o `-1` si ninguno.

Sintaxis: `elemento = arreglo.find(fn(elemento[, índice[, arreglo]]));`
Ejemplo:

```
const edades = [12, 18, 25, 16];
console.log(edades.find(x => x >= 18));      // 18 (primer elemento >=18)
console.log(edades.findIndex(x => x >= 18)); // 1 (índice de 18)
```

Notas: Ambos métodos recorren hasta encontrar coincidencia y dejan de iterar. No modifican el arreglo. Son similares a `filter`, pero devuelven un único resultado (el primero). A diferencia de `filter()`, que puede devolver múltiples coincidencias, `find()`/`findIndex()` solo hallan la primera. Si buscas *todos* los elementos que cumplan, usa `filter()`[14][5].

`findLast()` y `findLastIndex()` (ES2023)

Descripción: Análogos a `find`/`findIndex`, pero recorren de derecha a izquierda.
- `findLast(fn)`: Devuelve el **último elemento** que satisface la condición, o `undefined` si no hay ninguno.
- `findLastIndex(fn)`: Devuelve el índice del último elemento que cumple la condición, o `-1` si no lo encuentra.
Ejemplo:

```
const arr = [5, 12, 50, 130, 44];
console.log(arr.findLast(x => x > 45));      // 130
console.log(arr.findLastIndex(x => x > 45)); // 3
```

Notas: Introducidos en ES2023, operan en orden inverso. No mutan el array. No hay versión en MDN español aún, pero son útiles para obtener la última coincidencia.

`includes()`, `indexOf()` y `lastIndexOf()`

Descripción: Buscan elementos por igualdad.
- `includes(valor[, desde])`: Devuelve `true` si el arreglo contiene `valor` (compara con igualdad estricta *sameValueZero*), o `false` en caso contrario[15]. Soporta `NaN` correctamente. Opcionalmente empieza desde un índice (`desde`, negativo permite contar desde el final).
- `indexOf(valor[, desde])`: Devuelve el índice de la primera aparición de `valor` (comparación `===`), o `-1` si no está[16].
- `lastIndexOf(valor[, desde])`: Devuelve el índice de la última aparición de `valor`, o `-1` si no existe[17].
Sintaxis:

```
arreglo.includes(valorBuscado);
arreglo.indexOf(valorBuscado);
arreglo.lastIndexOf(valorBuscado);
```

Ejemplo:

```
const arr = ["a", "b", "a", "c"];
console.log(arr.includes("a"));      // true
console.log(arr.indexOf("a"));       // 0
console.log(arr.indexOf("z"));       // -1
console.log(arr.lastIndexOf("a"));   // 2
```

Notas: `includes()` retorna booleano, ideal para comprobaciones rápidas. A diferencia de `indexOf`, `includes(NaN)` es `true` si NaN está en el array, pues usa *SameValueZero*. Ninguno modifica el array original. Cuidado: `includes` y `indexOf` distinguen mayúsculas/minúsculas en strings.

`some()` y `every()`

Descripción: Métodos de prueba condicional.
- `some(fn)`: Retorna `true` si *al menos uno* de los elementos cumple la condición (`fn` devuelve verdadero)[18].
- `every(fn)`: Retorna `true` si *todos* los elementos cumplen la condición[19].
Ejemplo:

```
const arr = [2, 5, 8, 1];
console.log(arr.some(x => x > 5));  // true (8 > 5)
console.log(arr.every(x => x > 0)); // true (todos > 0)
console.log(arr.every(x => x < 5)); // false (8 no < 5)
```

Notas: No mutan el array. Se usan para validaciones: por ejemplo, `some` equivale a "¿existe un elemento que…?", `every` equivale a "¿todos los elementos…?". Si buscas un elemento con la condición, pero solo quieres saber si existe o obtenerlo, quizá `find()` o `findIndex()` convienen más.

`at()` (ES2022)

Descripción: Permite acceder por índice, aceptando índices negativos desde el final[20]. Es equivalente a `arr[pos]` pero más legible con negativos.
Sintaxis: `elemento = arreglo.at(índice);`
Ejemplo:

```
const letras = ["a", "b", "c", "d"];
console.log(letras.at(1));  // "b"
console.log(letras.at(-1)); // "d"
```

Notas: `at()` **no muta** el arreglo. Simplifica operaciones como obtener el último elemento con `arr.at(-1)` en lugar de `arr[arr.length-1]`.

Mutadores (Métodos que modifican el arreglo)

Añadir y eliminar en los extremos

- `push(...elementos)`: Añade uno o más elementos al **final** del arreglo y devuelve la nueva longitud[21]. Mutante.
- `pop()`: Elimina el último elemento del arreglo y lo devuelve[22]. Mutante.
- `unshift(...elementos)`: Añade uno o más elementos al **inicio** del arreglo y devuelve la nueva longitud[23]. Mutante.
- `shift()`: Elimina el primer elemento del arreglo y lo devuelve[24]. Mutante.
  Ejemplo:

  ```
  let arr = [1, 2, 3];
  arr.push(4);    // arr -> [1,2,3,4]
  ```

```
arr.shift();    // arr -> [2,3,4], devuelve 1
arr.unshift(0); // arr -> [0,2,3,4]
arr.pop();      // arr -> [0,2,3], devuelve 4
```

Notas: Todos estos métodos **mutan** el array original (cambian su contenido y longitud). `push` y `unshift` devuelven la nueva longitud, mientras que `pop` y `shift` devuelven el elemento eliminado.

`splice()`

Descripción: Añade, elimina o reemplaza elementos en cualquier posición. Es muy versátil[25].
Sintaxis: `arr.splice(inicio, cuantosEliminar[, elem1, elem2, ...])`
- Si `cuantosEliminar > 0`, elimina esa cantidad de elementos a partir de `inicio` (y devuelve un array con lo eliminado).
- Si se proporcionan más argumentos (`elem1, ...`), los inserta en esa posición.
Ejemplo:

```
let arr = [1, 2, 3, 4, 5];
let eliminados = arr.splice(2, 2, 8, 9);
// arr ahora es [1, 2, 8, 9, 5]
// eliminados es [3, 4]
```

Notas: `splice()` **muta** el arreglo original[25] (inserta o borra elementos in situ). Devuelve un nuevo array con los elementos borrados (si los hay). Para extraer sin mutar, use `slice()` o `concat()`.

`fill()` y `copyWithin()`

- `fill(valor[, inicio, fin])`: Rellena (asigna) el mismo valor a todos los elementos del arreglo entre `inicio` (inclusive) y `fin` (exclusive)[26]. Mutante.
- `copyWithin(destino, inicio[, fin])`: Copia parte del arreglo dentro de sí mismo. Por ejemplo, `arr.copyWithin(0, 2, 4)` copiaría el segmento `arr[2..3]` al principio del array. Mutante[27].
  Ejemplo:

  ```
  let arr = [1, 2, 3, 4, 5];
  arr.fill(0, 2, 4);
  console.log(arr); // [1, 2, 0, 0, 5]

  let arr2 = [10, 20, 30, 40, 50];
  arr2.copyWithin(1, 3, 5);
  console.log(arr2); // [10, 40, 50, 40, 50]
  ```

  Notas: Ambos métodos **mutan** el arreglo original. `fill()` sobrescribe valores; `copyWithin()` desplaza valores internos. Útiles para inicialización o reorganización en sitio.

Ordenación y Reorganización

`sort()` y `reverse()`

- `reverse()`: Invierte el orden de los elementos en el arreglo original[28]. Mutante. Ejemplo: `[1,2,3].reverse()` → `[3,2,1]`.
- `sort()`: Ordena los elementos **in-place** (por defecto según orden de cadena Unicode). Mutante[29]. Ejemplo: `[10,2,30].sort()` → `[10,2,30]` (no numérico, sino lexicográfico). Para orden numérico hay que pasar una función comparadora, e.g. `arr.sort((a,b) => a-b)`.
  Notas: Ambas modifican el arreglo original[29][28]. `sort()` puede sorprender: sin función comparadora ordena números como strings. Si se necesita el orden inverso sin mutar, use `toReversed()` (v. abajo).

Métodos no mutantes: `toReversed()`, `toSorted()`, `toSpliced()`, `with()` (ES2023+)

Estos métodos recién introducidos devuelven **nuevos arreglos** sin alterar el original. Funcionan como sus contrapartes mutantes, pero **inmutables**:

- `toReversed()`: Retorna un nuevo array con los elementos en orden inverso[30]. (Equivale a `arr.slice().reverse()`, pero sin mutar).
- `toSorted(comparador)`: Retorna un nuevo array ordenado, sin modificar el original[31]. Usa opcionalmente una función comparadora. (A diferencia de `sort()`, el original queda igual).
- `toSpliced(inicio, cuantosEliminar[, ...])`: Retorna un nuevo array con los elementos eliminados/reemplazados en la posición dada, sin mutar el original[32]. (Es como usar `slice()` y `concat()` de forma conveniente).
- `with(índice, valor)`: Retorna un nuevo array donde se reemplaza el elemento en `índice` por `valor`, dejando el original intacto[33].
  Ejemplo:

```
const orig = [1, 2, 3, 4];
const rev = orig.toReversed();      // [4,3,2,1]
const sorted = orig.toSorted();     // [1,2,3,4] (idéntico porque ya
estaba ordenado)
const spliceado = orig.toSpliced(1, 2, 9); // [1,9,4]
const mod = orig.with(0, 100);      // [100,2,3,4]
console.log(orig); // [1,2,3,4] (original sin cambios)
```

  Notas: Estos métodos **no mutan** el array original (retornan copias modificadas)[34][33]. Requieren entornos compatibles (ES2023). Resultan útiles cuando se quiere seguir un estilo inmutable.

Acceso y Extracción sin mutación

`slice()`

Descripción: Extrae parte del arreglo y devuelve un **nuevo** arreglo con los elementos seleccionados[35]. No modifica el original.
Sintaxis: `subArray = arreglo.slice(inicio, fin)` (como `substring`: `inicio` inclusive, `fin` exclusivo).
Ejemplo:

```
const arr = [10, 20, 30, 40, 50];
const sub = arr.slice(1, 4);
console.log(sub); // [20, 30, 40]
```

Notas: No muta el array. A diferencia de `splice()`, que modifica, `slice()` deja todo igual. Para clonar un array completo se usa `slice(0)` o con operador spread `[...arr]`.

`concat()`

Descripción: Devuelve un **nuevo** arreglo que es la concatenación de uno o varios arreglos (o valores) al original[36]. No modifica el original.
Sintaxis: `nuevo = arreglo.concat(valor1, arreglo2, arreglo3, ...)`
Ejemplo:

```
const a = [1, 2];
const b = [3, 4];
console.log(a.concat(b, 5)); // [1,2,3,4,5]
```

Notas: No muta los arreglos originales. Es análogo a usar el operador spread (`[...a, ...b]`).

`join()`, `toString()`, `toLocaleString()`

- `join(separador)`: Concatena todos los elementos en un string, separados por el separador dado (por defecto “,”)[37]. No muta el array. Ejemplo: `[1,2,3].join("-")` → `"1-2-3"`.
- `toString()`: Llama internamente a `join(",")`, devolviendo un string de los elementos separados por comas[38].
- `toLocaleString()`: Similar a `toString()`, pero formatea cada elemento según la configuración regional (p.ej. números con comas decimales locales)[39].
  Ejemplo:

  ```
  const nums = [1234.5, 6789.01];
  console.log(nums.join(" | "));       // "1234.5 | 6789.01"
  console.log(nums.toString());       // "1234.5,6789.01"
  console.log(nums.toLocaleString('de')); // e.g. "1.234,5,6.789,01"
  (dependiendo de locale)
  ```

Notas: Ninguno de estos muta el arreglo; sólo retorna cadenas. Muy útiles para presentar datos.

Iteradores: `entries()`, `keys()`, `values()` y `[Symbol.iterator]`

- `entries()`: Retorna un **iterador** de pares `[índice, valor]` para cada elemento[40]. Puede usarse con `for..of`.
- `keys()`: Retorna un iterador con todas las **claves** (índices) del array[41].
- `values()`: Retorna un iterador con todos los **valores** del array[42]. (De hecho, `values()` es el iterador por defecto, igual a usar `[Symbol.iterator]` de un array).
  Ejemplo:

```
const arr = ["x", "y", "z"];
for (const [i, val] of arr.entries()) {
  console.log(i, val);
}
// 0 "x"
// 1 "y"
// 2 "z"
for (const key of arr.keys()) { console.log(key); }   // 0,1,2
for (const val of arr.values()) { console.log(val); } // "x","y","z"
```

Notas: Estos métodos **no mutan** el arreglo; proveen objetos iteradores. Son útiles para recorrer índices o pares clave-valor. Por ejemplo, `[...arr.keys()]` da un array con [0,1,2, …].

Métodos Estáticos de Array

- **`Array.from(objetoIterable[, fnMap, thisArg])`**: Crea un nuevo `Array` a partir de un objeto iterable o "similar a arreglo"[43]. Ejemplo: `Array.from("hola")` → `["h","o","l","a"]`. Soporta una función map opcional en el proceso.
- **`Array.of(...elementos)`**: Crea un nuevo `Array` con los argumentos dados como elementos[44]. A diferencia del constructor `Array(7)` (que crea un array vacío de longitud 7), `Array.of(7)` produce `[7]`.
- **`Array.isArray(valor)`**: Devuelve `true` si `valor` es un array, `false` en caso contrario[45]. Es más fiable que `instanceof Array` (que falla entre iframes)[46].
- **`Array.fromAsync()`** (ES2022/ESNext): Crea un `Array` a partir de un iterable asíncrono (retorna una promesa que resuelve en un array). (Ej: `await Array.fromAsync(fetchLines(), x=>x*2)`).
  Ejemplo:

```
console.log(Array.from("abc")); // ["a", "b", "c"]
console.log(Array.of(3));       // [3]
console.log(Array.isArray([])); // true
```

Notas: Estos métodos no pertenecen a instancias de arreglo sino al constructor `Array`. No modifican arrays existentes (devuelven nuevos arrays o booleanos).

Diferencias Clave y Advertencias Comunes

- **`.filter()` vs `.find()`:** `filter` devuelve un array (de posibles varios elementos), `find` devuelve un único elemento o `undefined`. Use `find()` cuando solo le interesa el primer encuentro[5][6].
- **`some()` vs `every()`:** `some` pregunta si **al menos uno** cumple la condición, `every` exige que **todos** la cumplan[47][48].
- **`includes()` vs `indexOf()`:** `includes` es más directo (retorna booleano) y permite detectar `NaN` en el array, mientras que `indexOf` retorna índice (o `-1` si no está). Para solo saber si existe un valor, `includes` es más claro.
- **`slice()` vs `splice()`:** `slice` extrae sin mutar (devuelve copia de segmento), `splice` altera el array (borra/inserta). Si no quiere alterar el original, evite `splice`.
- **Ordenación de números en `sort()`:** Por defecto, ordena como cadenas. Para orden numérico usar comparador, por ejemplo `arr.sort((a,b)=>a-b)`. De lo contrario, `["10","2"]` quedará `["10","2"]`.
- **Uso de `reduce()`:** Siempre provea un valor inicial si el array podría estar vacío. Si no, lanzará `TypeError`[9]. La primera llamada de `callback` tomará `valorInicial` como acumulador si se da. Además, cuidado con retornar `undefined` accidentalmente en la función reductora.
- **Inmutabilidad vs mutabilidad:** Muchos métodos (`map`, `filter`, `slice`, `concat`, `toSorted()`, etc.) **no mutan** el array original (crean uno nuevo). Otros (`push`, `pop`, `shift`, `unshift`, `splice`, `sort`, `reverse`, `fill`, `copyWithin`) sí mutan. Se debe prestar atención a esto para evitar efectos colaterales inesperados.
- **Cadenas vs arreglos:** `join`, `toString`, etc., devuelven cadenas. Al imprimir arreglos directamente en consola (`console.log(arr)`), el navegador generalmente llama `toString()`. Recuerde esto si se confunde sobre por qué ve comas.
- **ForEach y ruptura de bucle:** No se puede detener un `forEach()` con `break` o `return`. Si necesita detener la iteración anticipadamente, use un bucle clásico o los nuevos métodos iteradores (`find`, `some`, etc.).
- **Profundidad de `flat()`:** El valor predeterminado de profundidad es 1. Para aplanar completamente, use `arr.flat(Infinity)`.

Cada método aquí descrito está soportado en navegadores modernos (ES5+). Algunos (como `flatMap`, `findLast`, `toReversed`, etc.) requieren versiones recientes de JavaScript (ES2019+ o ES2023). Verifique compatibilidad si trabaja en entornos antiguos.

Fuentes: Las definiciones y comportamientos descritos provienen de la documentación oficial de MDN Web Docs[5][4][49][11][42], junto con ejemplos de uso estándar en JavaScript. Cada observación clave está soportada por estas referencias.

---

[1] [5] [6] [7] [8] [16] [17] [18] [19] [21] [22] [23] [24] [25] [26] [27] [28] [29] [35] [36] [37] [38] [39] [47] [48] Array - JavaScript | MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array

[2] [4] Array.prototype.filter() - JavaScript | MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

[3] [11] [12] [13] Array.prototype.forEach() - JavaScript | MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach

[9] [49] Array.prototype.reduce() - JavaScript | MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce

[10] Array.prototype.flat() - JavaScript | MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/flat

[14] Array.prototype.find() - JavaScript | MDN

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find

[15] Array.prototype.includes() - JavaScript | MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/includes

[20] Array.prototype.at() - JavaScript | MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/at

[30] [31] [32] [33] [34] Array - JavaScript | MDN

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

[40] Array.prototype.entries()

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/entries

[41] Array.prototype.keys() - JavaScript | MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/keys

[42] Array.prototype.values() - JavaScript | MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/values

[43] Array.from() - JavaScript | MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/from

[44] Array.of() - JavaScript | MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/of

[45] [46] Array.isArray() - JavaScript | MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/isArray