

Métodos de arrays

Los arrays (también llamados arreglos o matrices) cuentan con muchos métodos. Para hacer las cosas más sencillas, en este capítulo se encuentran divididos en dos partes.

Agregar/remove items

Ya conocemos algunos métodos que agregan o extraen elementos del inicio o final de un array:

- `arr.push(...items)` – agrega ítems al final,
- `arr.pop()` – extrae un ítem del final,
- `arr.shift()` – extrae un ítem del inicio,
- `arr.unshift(...items)` – agrega ítems al principio.

Veamos algunos métodos más.

splice

¿Cómo podemos borrar un elemento de un array?

Los arrays son objetos, por lo que podemos intentar con `delete` :

```
1 let arr = ["voy", "a", "casa"];
2
3 delete arr[1]; // remueve "a"
4 alert( arr[1] ); //
5 undefined
6
7 // ahora arr = ["voy", , "casa"];
8 alert( arr.length ); // 3
```

El elemento fue borrado, pero el array todavía tiene 3 elementos; podemos ver que `arr.length == 3` .

Es natural, porque `delete obj.key` borra el valor de `key` , pero es todo lo que hace. Esto está bien en los objetos, pero en general lo que buscamos en los arrays es que el resto de los elementos se desplace y se ocupe el lugar libre. Lo que esperamos es un array más corto.

Por lo tanto, necesitamos utilizar métodos especiales.

El método `arr.splice` funciona como una navaja suiza para arrays. Puede hacer todo: insertar, remover y reemplazar elementos.

La sintaxis es:

```
1 arr.splice(start[, deleteCount, elem1, ..., elemN])
```

Esto modifica `arr` comenzando en el índice `start` : remueve la cantidad `deleteCount` de elementos y luego inserta `elem1, ..., elemN` en su lugar. Lo que devuelve es un array de los elementos removidos.

Este método es más fácil de entender con ejemplos.

Empecemos removiendo elementos:

```

1 let arr = ["Yo", "estudio", "JavaScript"];
2
3
4 arr.splice(1, 1); // desde el índice 1, remover 1 elemento
5 alert( arr ); // ["Yo", "JavaScript"]

```

¿Fácil, no? Empezando desde el índice 1 removi6 1 elemento.

En el pr6ximo ejemplo removemos 3 elementos y los reemplazamos con otros 2:

```

1 let arr = ["Yo", "estudio", "JavaScript"];
2
3 // remueve los primeros 3 elementos y
4 los reemplaza con otros arr.splice(0, 3, "a", "bailar");
5 alert( arr ); // ahora ["a", "bailar", "ahora",
6 "mismo"]

```

Aqu6 podemos ver que splice devuelve un array con los elementos removidos:

```

1 let arr = ["Yo", "estudio"];
2
3 // remueve los 2
4 primeros elementos let removed =
5 arr.splice(0, 2);
6 alert( removed ); // "Yo", "estudio" <-- array de los elementos
removidos

```

El m6todo splice tambi6n es capaz de insertar elementos sin remover ning6n otro. Para eso necesitamos establecer deleteCount en 0 :

```

1 let arr = ["Yo", "estudio", "JavaScript"];
2
3 // desde el index 2
4 // remover 0
5 // despu6s insertar "el", "complejo" y "language"
6
7 arr.splice(2, 0, "el", "complejo", "language");
8 alert( arr ); // "Yo", "estudio", "el", "complejo", "language",
"JavaScript"

```

Los índices negativos están permitidos

En este y en otros métodos de arrays, los índices negativos están permitidos. Estos índices indican la posición comenzando desde el final del array, de la siguiente manera:

```
1 let arr = [1, 2, 5];
2
3 // desde el index -1 (un lugar desde el final)
4 // remover 0 elementos,
5 // después insertar 3 y 4
6 arr.splice(-1, 0, 3, 4);
7
8 alert( arr ); // 1,2,3,4,5
```

slice

El método `arr.slice` es mucho más simple que `arr.splice`.

La sintaxis es:

```
1 arr.slice([principio], [final])
```

Devuelve un nuevo array copiando en el mismo todos los elementos desde `principio` hasta `final` (sin incluir `final`). `principio` y `final` pueden ser negativos, en cuyo caso se asume la posición desde el final del array.

Es similar al método para strings `str.slice`, pero en lugar de substrings genera subarrays.

Por ejemplo:

```
1 let arr = ["t", "e", "s", "t"];
2
3 alert( arr.slice(1, 3) ); // e,s (copia desde 1 hasta 3)
4 alert( arr.slice(-2) ); // s,t (copia desde -2 hasta el
5 final)
```

También podemos invocarlo sin argumentos: `arr.slice()` crea una copia de `arr`. Se utiliza a menudo para obtener una copia que se puede transformar sin afectar el array original.

concat

El método `arr.concat` crea un nuevo array que incluye los valores de otros arrays y elementos adicionales.

La sintaxis es:

```
1 arr.concat(arg1, arg2...)
```

Este acepta cualquier número de argumentos, tanto arrays como valores.

El resultado es un nuevo array conteniendo los elementos de `arr`, después `arg1`, `arg2` etc.

Si un argumento `argN` es un array, entonces todos sus elementos son copiados. De otro modo el argumento en sí es copiado.

Por ejemplo:

```
1 let arr = [1, 2];
2
3 // crea un array a partir de: arr y [3,4]
4 alert( arr.concat([3, 4]) ); // 1,2,3,4
5
6 // crea un array a partir de: arr y [3,4] y [5,6]
7 alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6
8 // crea un array a partir de: arr y [3,4], luego agrega los valores !
9 y 6 alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
10
```

Normalmente, solo copia elementos desde arrays. Otros objetos, incluso si parecen arrays, son agregados como un todo:

```
1 let arr = [1, 2];
2
3 let arrayLike = {
4   0: "something",
5   length: 1
6 }; alert( arr.concat(arrayLike) ); // 1,2,[object
7   Object]
8
```

...Pero si un objeto similar a un array tiene la propiedad especial `Symbol.isConcatSpreadable`, entonces `concat` lo trata como un array y en lugar de añadirlo como un todo, solo añade sus elementos.

```
1 let arr = [1, 2];
2
3 let arrayLike = {   0:
4   "something",
5   1: "else",
6   [Symbol.isConcatSpreadable]: true,
7
8   length: 2
9 }; alert( arr.concat(arrayLike) ); //
10 1,2,something,else
```

Iteración: `forEach`

El método `arr.forEach` permite ejecutar una función a cada elemento del array.

La sintaxis:

```
1 arr.forEach(function(item, index, array) {
2   // ... hacer algo con el elemento
3 });
```

Por ejemplo, el siguiente código muestra cada elemento del array:

```
1 // para cada elemento ejecuta alert
2 ["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
```

Y este caso más detallado da la posición del elemento en el array:

```
1 ["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) =>
2 {   alert(`${item} is at index ${index} in ${array}`); });
3
```

El resultado de la función (si lo hay) se descarta y se ignora.

Buscar dentro de un array

Ahora vamos a ver métodos que buscan elementos dentro de un array.

indexOf/lastIndexOf e includes

Los métodos `arr.indexOf`, `arr.lastIndexOf` y `arr.includes` tienen la misma sintaxis y hacen básicamente lo mismo que sus contrapartes de strings, pero operan sobre elementos en lugar de caracteres:

- `arr.indexOf(item, from)` – busca `item` comenzando desde el `index from`, y devuelve el `index` donde fue encontrado, de otro modo devuelve `-1`.
- `arr.lastIndexOf(item, from)` – igual que el anterior, pero busca de derecha a izquierda.
- `arr.includes(item, from)` – busca `item` comenzando desde el índice `from`, devuelve `true` en caso de ser encontrado.

Por ejemplo:

```
1 let arr = [1, 0, false];
2
3 alert( arr.indexOf(0) ); // 1
4 alert( arr.indexOf(false) ); // 2
5 alert( arr.indexOf(null) ); // -1
6 alert( arr.includes(1) ); //
7 true
```

Tener en cuenta que el método usa comparación estricta (`===`). Por lo tanto, si buscamos `false`, encontrará exactamente `false` y no cero.

Si queremos comprobar si un elemento está incluido y no necesitamos saber su ubicación exacta, es preferible usar `arr.includes`

Además, una pequeña diferencia de `includes` es que puede manejar correctamente `NaN` a diferencia de `indexOf/lastIndexOf` :

```
1 const arr = [NaN]; alert( arr.indexOf(NaN) ); // -1 (debería ser 0, pero no funciona)
2
3 alert( arr.includes(NaN) ); // true (correcto)
```

find y findIndex

Imaginemos que tenemos un array de objetos. ¿Cómo podríamos encontrar un objeto con una condición específica?

Para este tipo de casos es útil el método `arr.find(fn)`

La sintaxis es:

```
1 let result = arr.find(function(item, index, array) {
2   // si true es devuelto aquí, find devuelve el ítem y la iteración se detiene
3   // para el caso en que sea false, devuelve undefined
4 });
```

La función es llamada para cada elemento del array, uno después del otro:

- `item` es el elemento.
- `index` es su índice.
- `array` es el array mismo.

Si devuelve `true` , la búsqueda se detiene y el `item` es devuelto. Si no encuentra nada, entonces devuelve `undefined` .

Por ejemplo, si tenemos un array de usuarios, cada uno con los campos `id` y `name` . Encontremos el elemento con `id == 1` :

```
1 let users = [
2   {id: 1, name: "Celina"},
3   {id: 2, name: "David"},
4   {id: 3, name: "Federico"}
5 ];
6 let user = users.find(item => item.id == 1);
7 alert(user.name); //
8 Celina
9
```

En la vida real los arrays de objetos son bastante comunes por lo que el método `find` resulta muy útil.

Ten en cuenta que en el ejemplo anterior le pasamos a `find` la función `item => item.id == 1` con un argumento. Esto es lo más común, otros argumentos son raramente usados en esta función.

El método `arr.findIndex` es esencialmente lo mismo, pero devuelve el índice donde el elemento fue encontrado en lugar del elemento en sí y devuelve `-1` cuando no encuentra nada. **filter**

El método `find` busca un único elemento (el primero) que haga a la función devolver `true`.

Si existieran varios elementos que cumplen la condición, podemos usar `arr.filter(fn)`.

La sintaxis es similar a `find`, pero `filter` devuelve un array con todos los elementos encontrados:

```
1 let results = arr.filter(function(item, index, array) {  
2   // si devuelve true, el elemento es ingresado al array y la iteración continúa  
3   // si nada es encontrado, devuelve un array vacío  
4 });
```

Por ejemplo:

```
1 let users = [  
2   {id: 1, name: "Celina"},  
3   {id: 2, name: "David"},  
4   {id: 3, name: "Federico"}  
5 ];  
6  
7 // devuelve un array con los dos primeros usuarios  
8 let someUsers = users.filter(item => item.id < 3);  
9 alert(someUsers.length); //  
10 2
```

Transformar un array

Pasamos ahora a los métodos que transforman y reordenan un array.

map

El método `arr.map` es uno de los métodos más comunes y ampliamente usados.

Este método llama a la función para cada elemento del array y devuelve un array con los resultados.

La sintaxis es:

```
1 let result = arr.map(function(item, index, array) {  
2   // devuelve el nuevo valor en lugar de item  
3 });
```

Por ejemplo, acá transformamos cada elemento en el valor de su respectivo largo (`length`):

```
1 let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item =>  
2 item.length); alert(lengths); // 5,7,6
```

sort(fn)

Cuando usamos `arr.sort()`, este ordena el propio array cambiando el orden de los elementos.

También devuelve un nuevo array ordenado pero éste usualmente se descarta ya que `arr` en sí mismo es modificado.

Por ejemplo:

```
1 let arr = [ 1, 2, 15 ];
2
3 // el método reordena el contenido de arr
4 arr.sort();
5 alert( arr ); // 1, 15,
6 2
```

¿Notas algo extraño en los valores de salida?

Los elementos fueron reordenados a 1, 15, 2. Pero ¿por qué pasa esto?

Los elementos son ordenados como strings (cadenas de caracteres) por defecto

Todos los elementos son literalmente convertidos a string para ser comparados. En el caso de strings se aplica el orden lexicográfico, por lo que efectivamente "2" > "15".

Para usar nuestro propio criterio de reordenamiento, necesitamos proporcionar una función como argumento de `arr.sort()`.

La función debe comparar dos valores arbitrarios y devolver el resultado:

```
1 function compare(a, b) {
2   if (a > b) return 1; // si el primer valor es mayor que el segundo
3   if (a == b) return 0; // si ambos valores son iguales
4   if (a < b) return -1; // si el primer valor es menor que el segundo
5 }
```

Por ejemplo, para ordenar como números:

```
1
2
3 function compareNumeric(a, b) {
4   if (a > b) return 1;   if (a == b)
5   return 0;   if (a < b) return -1;
6 }
7 let arr = [ 1, 2, 15 ];
8
9 arr.sort(compareNumeric);
10 alert(arr); // 1, 2, 15
11
```

Ahora sí funciona como esperábamos.

Detengámonos un momento y pensemos qué es lo que está pasando. El array `arr` puede ser un array de cualquier cosa, ¿no? Puede contener números, strings, objetos o lo que sea. Podemos decir que tenemos un

conjunto de *ciertos items*. Para ordenarlos, necesitamos una *función de ordenamiento* que sepa cómo comparar los elementos. El orden por defecto es hacerlo como strings.

El método `arr.sort(fn)` implementa un algoritmo genérico de orden. No necesitamos preocuparnos de cómo funciona internamente (la mayoría de las veces es una forma optimizada del algoritmo [quicksort](#) o [Timsort](#)). Este método va a recorrer el array, comparar sus elementos usando la función dada y, finalmente, reordenarlos. Todo lo que necesitamos hacer es proveer la `fn` que realiza la comparación.

Por cierto, si queremos saber qué elementos son comparados, nada nos impide ejecutar `alert()` en ellos:

```
1 [1, -2, 15, 2, 0, 8].sort(function(a, b)
2 { alert( a + " <> " + b ); return a -
3 b;
4 });
```

El algoritmo puede comparar un elemento con muchos otros en el proceso, pero trata de hacer la menor cantidad de comparaciones posible.

Una función de comparación puede devolver cualquier número

En realidad, una función de comparación solo es requerida para devolver un número positivo para "mayor" y uno negativo para "menor".

Esto nos permite escribir una función más corta:

```
1 let arr = [ 1, 2, 15 ];
2
3 arr.sort(function(a, b) { return a - b; });
4
5 alert(arr); // 1, 2, 15
```

Mejor, con funciones de flecha

¿Recuerdas las [arrow functions](#)? Podemos usarlas en este caso para un ordenamiento más prolijo:

```
1 arr.sort( (a, b) => a - b );
```

Esto funciona exactamente igual que la versión más larga de arriba.

Usa `localeCompare` para strings

¿Recuerdas el algoritmo de comparación [strings](#)? Este compara letras por su código por defecto.

Para muchos alfabetos, es mejor usar el método `str.localeCompare` para ordenar correctamente letras como por ejemplo `Ö`.

Por ejemplo, vamos a ordenar algunos países en alemán:

```
1 let paises = ['Österreich', 'Andorra', 'Vietnam'];
2
3 alert( paises.sort( (a, b) => a > b ? 1 : -1 ) ); // Andorra, Vietnam, Öster
4
5 alert( paises.sort( (a, b) => a.localeCompare(b) ) ); // Andorra,Österreich
```

reverse

El método `arr.reverse` revierte el orden de los elementos en `arr`.

Por ejemplo:

```
1 let arr = [1, 2, 3, 4, 5];
2 arr.reverse();
3 alert( arr ); //
4 5,4,3,2,1
```

También devuelve el array `arr` después de revertir el orden. **split**

y join

Analicemos una situación de la vida real. Estamos programando una app de mensajería y el usuario ingresa una lista de receptores delimitada por comas: `Celina, David, Federico`. Pero para nosotros un array sería mucho más práctico que una simple string. ¿Cómo podemos hacer para obtener un array?

El método `str.split(delim)` hace precisamente eso. Separa la string en elementos según el delimitante `delim` dado y los devuelve como un array.

En el ejemplo de abajo, separamos por "coma seguida de espacio":

```
1 let nombres = 'Bilbo, Gandalf, Nazgul';
2 let arr = nombres.split(', ');
3 for (let name of arr) { alert( `Un mensaje para ${name}.` ); // Un
4 para Bilbo (y los otros n
5 }
6
7
```

El método `split` tiene un segundo argumento numérico opcional: un límite en la extensión del array. Si se provee este argumento, entonces el resto de los elementos son ignorados. Sin embargo en la práctica rara vez se utiliza:

```
1 let arr = 'Bilbo, Gandalf, Nazgul, Saruman'.split(', ', 2);
2 alert(arr); // Bilbo,
3 Gandalf
```

Separar en letras

El llamado a `split(s)` con un `s` vacío separará el string en un array de letras:

```
1 let str = "test";
2
3 alert( str.split('') ); // t,e,s,t
```

`arr.join(glue)` hace lo opuesto a `split`. Crea una string de `arr` elementos unidos con `glue` (pegamento) entre ellos.
Por ejemplo:

```
1 let arr = ['Bilbo', 'Gandalf', 'Nazgul'];
2 let str = arr.join(';'); // une el array en una string
3 usando ;
4 alert( str ); //
5 Bilbo;Gandalf;Nazgul
```

reduce/reduceRight

Cuando necesitamos iterar sobre un array podemos usar `forEach`, `for` o `for..of`.

Cuando necesitamos iterar y devolver un valor por cada elemento podemos usar `map`.

Los métodos `arr.reduce` y `arr.reduceRight` también pertenecen a ese grupo de acciones pero son un poco más complejos. Se los utiliza para calcular un único valor a partir del array.

La sintaxis es la siguiente:

```
1 let value = arr.reduce(function(accumulator, item, index, array) {
2   // ...
3 }, [initial]);
```

La función es aplicada a todos los elementos del array, uno tras de otro, y va arrastrando el resultado parcial al próximo llamado.

Argumentos:

- `accumulator` – es el resultado del llamado previo de la función, equivale a `initial` la primera vez (si `initial` es dado como argumento).
- `item` – es el elemento actual del array.
- `index` – es la posición.
- `array` – es el array.

Mientras la función sea llamada, el resultado del llamado anterior se pasa al siguiente como primer argumento.

Entonces, el primer argumento es el acumulador que almacena el resultado combinado de todas las veces anteriores en que se ejecutó, y al final se convierte en el resultado de `reduce`.

¿Suenan complicado?

La forma más simple de entender algo es con un ejemplo.

Acá tenemos la suma de un array en una línea:

```
let arr = [1, 2, 3, 4, 5];
1 let result = arr.reduce((sum, current) => sum + current,
2 0);
3
4
5
alert(result); // 15
```

La función pasada a `reduce` utiliza solo 2 argumentos, esto generalmente es suficiente.

Veamos los detalles de lo que está pasando.

1. En la primera pasada, `sum` es el valor `initial` (el último argumento de `reduce`), equivale a `0`, y `current` es el primer elemento de array, equivale a `1`. Entonces el resultado de la función es `1`.
2. En la segunda pasada, `sum = 1`, agregamos el segundo elemento del array (`2`) y devolvemos el valor `3`.
En la tercera pasada, `sum = 3` y le agregamos un elemento más, y así sucesivamente... El flujo de cálculos:

1	2	3	4	5
sum 0 current 1	sum 0+1 current 2	sum 0+1+2 current 3	sum 0+1+2+3 current 4	sum 0+1+2+3+4 current 5

→ $0+1+2+3+4+5 = 15$

O en la forma de una tabla, donde cada fila representa un llamado a una función en el próximo elemento del array:

	sum	current	result
primer llamado	0	1	1
segundo llamado	1	2	3
tercer llamado	3	3	6
cuarto llamado	6	4	10
quinto llamado	10	5	15

Acá podemos ver claramente como el resultado del llamado anterior se convierte en el primer argumento del llamado siguiente.

También podemos omitir el valor inicial:

```
1 let arr = [1, 2, 3, 4, 5];
2 // valor inicial removido (no 0) let result =
3 arr.reduce((sum, current) => sum + current);
4 alert( result ); //
5 15
6
```

El resultado es el mismo. Esto es porque en el caso de no haber valor inicial, reduce toma el primer elemento del array como valor inicial y comienza la iteración a partir del segundo elemento.

La tabla de cálculos es igual a la anterior menos la primer fila.

Pero este tipo de uso requiere tener extremo cuidado. Si el array está vacío, entonces el llamado a reduce sin valor inicial devuelve error.

Acá vemos un ejemplo:

```
1 let arr = [];
2
3 // Error: Reduce en un array vacío sin valor inicial
4 // si el valor inicial existe, reduce lo devuelve en el arr vacío.
5 arr.reduce((sum, current) => sum + current);
```

Por lo tanto siempre se recomienda especificar un valor inicial.

El método `arr.reduceRight` realiza lo mismo, pero va de derecha a izquierda.

Array.isArray

Los arrays no conforman un tipo diferente. Están basados en objetos.

Por eso `typeof` no ayuda a distinguir un objeto común de un array:

```
1
```

```
2 alert(typeof {}); // object
  alert(typeof []); // object
```

...Pero los arrays son utilizados tan a menudo que tienen un método especial para eso: `Array.isArray(value)`. Este devuelve `true` si el valor es un array y `false` si no lo es.

```
1 alert(Array.isArray({})); // false
2 alert(Array.isArray([])); //
3 true
```

La mayoría de los métodos aceptan “thisArg”

Casi todos los métodos para arrays que realizan llamados a funciones – como `find`, `filter`, `map`, con la notable excepción de `sort` – aceptan un parámetro opcional adicional `thisArg`.

Ese parámetro no está explicado en la sección anterior porque es raramente usado. Pero para ser exhaustivos necesitamos verlo.

Esta es la sintaxis completa de estos métodos:

```
1 arr.find(func, thisArg);
2 arr.filter(func, thisArg);
3 arr.map(func, thisArg);
4 // ...
5 // thisArg es el último argumento opcional
```

El valor del parámetro `thisArg` se convierte en `this` para `func`.

Por ejemplo, acá usamos un método del objeto `army` como un filtro y `thisArg` da el contexto:

```
1
2
3
4 let army = { minAge: 18, maxAge: 27, canJoin(user) {
5 return user.age >= this.minAge && user.age < this.maxAge;
6 }
7 }; let users =
8 [ {age: 16},
9   {age: 20},
10  {age: 23},
11  {age: 30}
12 ];
13
14 // encuentra usuarios para los cuales army.canJoin devuelve true
15 let soldiers = users.filter(army.canJoin, army);
16
17 alert(soldiers.length); // 2 alert(soldiers[0].age);
18 // 20 alert(soldiers[1].age); // 23
19
20
21
```

Si en el ejemplo anterior usáramos `users.filter(array.canJoin)` , entonces `array.canJoin` sería llamada como una función independiente con `this=undefined` , lo que llevaría a un error inmediato.

La llamada a `users.filter(array.canJoin, array)` puede ser reemplazada con `users.filter(user => array.canJoin(user))` que realiza lo mismo. Esta última se usa más a menudo ya que es un poco más fácil de entender.

Resumen

Veamos el ayudamemoria de métodos para arrays:

- Para agregar/remover elementos:
 - `push(...items)` – agrega ítems al final,
 - `pop()` – extrae un ítem del final,
 - `shift()` – extrae un ítem del inicio,
 - `unshift(...items)` – agrega ítems al inicio.
 - `splice(pos, deleteCount, ...items)` – desde el índice `pos` borra `deleteCount` elementos e inserta `items` .
 - `slice(start, end)` – crea un nuevo array y copia elementos desde la posición `start` hasta `end` (no incluido) en el nuevo array.
 - `concat(...items)` – devuelve un nuevo array: copia todos los elementos del array actual y le agrega `items` . Si alguno de los `items` es un array, se toman sus elementos.
- Para buscar entre elementos:
 - `indexOf/lastIndexOf(item, pos)` – busca por `item` comenzando desde la posición `pos` , devolviendo el índice o `-1` si no se encuentra.
 - `includes(value)` – devuelve `true` si el array tiene `value` , si no `false` .
 - `find/filter(func)` – filtra elementos a través de la función, devuelve el primer/todos los valores que devuelven `true` .
 - `findIndex` es similar a `find` pero devuelve el índice en lugar del valor.
- Para iterar sobre elementos:
 - `forEach(func)` – llama la `func` para cada elemento, no devuelve nada.
- Para transformar el array:
 - `map(func)` – crea un nuevo array a partir de los resultados de llamar a la `func` para cada elemento.
 - `sort(func)` – ordena el array y lo devuelve.
 - `reverse()` – ordena el array de forma inversa y lo devuelve.
 - `split/join` – convierte una cadena en un array y viceversa.
 - `reduce/reduceRight(func, initial)` – calcula un solo valor para todo el array, llamando a la `func` para cada elemento, obteniendo un resultado parcial en cada llamada y pasándolo a la siguiente.
- Adicional:
 - `Array.isArray(arr)` comprueba si `arr` es un array.

Por favor tener en cuenta que `sort` , `reverse` y `splice` modifican el propio array.

Estos métodos son los más utilizados y cubren el 99% de los casos. Pero existen algunos más:

- `arr.some(fn)/arr.every(fn)` comprueba el array.

La función `fn` es llamada para cada elemento del array de manera similar a `map`. Si alguno/todos los resultados son `true`, devuelve `true`, si no, `false`.

Estos métodos se comportan con similitud a los operadores `||` y `&&`: si `fn` devuelve un valor verdadero, `arr.some()` devuelve `true` y detiene la iteración de inmediato; si `fn` devuelve un valor falso, `arr.every()` devuelve `false` y detiene la iteración también.

Podemos usar `every` para comparar arrays:

```
function arraysEqual(arr1, arr2) {
1   return arr1.length === arr2.length && arr1.every((value, index) =>
2 }   alert( arraysEqual([1, 2], [1, 2])); //
3   true
4
5
```

value
===

- `arr.fill(value, start, end)` – llena el array repitiendo `value` desde el índice `start` hasta `end`.
- `arr.copyWithin(target, start, end)` – copia sus elementos desde la posición `start` hasta la posición `end` en *si mismo*, a la posición `target` (reescribe lo existente).
- `arr.flat(depth)/arr.flatMap(fn)` crea un nuevo array plano desde un array multidimensional.

Para la lista completa, ver [manual](#).

A primera vista puede parecer que hay demasiados métodos para aprender y un tanto difíciles de recordar. Pero con el tiempo se vuelve más fácil.

Revisa el ayudamemoria para conocerlos. Después realiza las prácticas de este capítulo para ganar experiencia con los métodos para arrays.

Finalmente si en algún momento necesitas hacer algo con un array y no sabes cómo, vuelve a esta página, mira el ayudamemoria y encuentra el método correcto. Los ejemplos te ayudarán a escribirlos correctamente y pronto los recordarás automáticamente y sin esfuerzo.

✓ Tareas

Transforma `border-left-width` en `borderLeftWidth`

importancia: 5

Escribe la función `camelize(str)` que convierta palabras separadas por guión como “mi-cadena-corta” en palabras con mayúscula “miCadenaCorta”.

Esto sería: remover todos los guiones y que cada palabra después de un guión comience con mayúscula.

Ejemplos:


```
1 camelize("background-color") == 'backgroundColor'; camelize("list-style-
2 image") == 'listStyleImage'; camelize("-webkit-transition") ==
3 'WebkitTransition';
```

P.D. Pista: usa `split` para dividir el string en un array, transfórmalo y vuelve a unirlo (`join`).

Filtrar un rango

importancia: 4

Escribe una función `filterRange(arr, a, b)` que obtenga un array `arr` , busque los elementos con valor mayor o igual a `a` y menor o igual a `b` y devuelva un array con los resultados.

La función no debe modificar el array. Debe devolver un nuevo array.

Por ejemplo:

```
1
2 let arr = [5, 3, 8, 1];
3 let filtered = filterRange(arr, 1,
4 4);
5 alert( filtered ); // 3,1 (valores dentro del rango)
6 alert( arr ); // 5,3,8,1 (array original no
7 modificado)
```

Filtrar rango "en el lugar"

importancia: 4

Escribe una función `filterRangeInPlace(arr, a, b)` que obtenga un array `arr` y remueva del mismo todos los valores excepto aquellos que se encuentran entre `a` y `b` . El test es: $a \leq arr[i] \leq b$.

La función solo debe modificar el array. No debe devolver nada.

Por ejemplo:

```
1 let arr = [5, 3, 8, 1];
2 filterRangeInPlace(arr, 1, 4); // remueve los números excepto aquellos entre
3 1
4 alert( arr ); // [3,
5 1]
```

Ordenar en orden decreciente

importancia: 4

```

1 let arr = [5, 2, 1, -10, 8];
2
3 // ... tu código para ordenar en orden decreciente
4 alert( arr ); // 8, 5, 2, 1, -
5 10

```

Copia y ordena un array

importancia: 5

Supongamos que tenemos un array `arr`. Nos gustaría tener una copia ordenada del mismo, pero mantener `arr` sin modificar.

Crea una función `copySorted(arr)` que devuelva esa copia.

```

1 let arr = ["HTML", "JavaScript", "CSS"];
2
3
4 let sorted = copySorted(arr);
5 alert( sorted ); // CSS, HTML, JavaScript alert(
6 arr ); // HTML, JavaScript, CSS (sin cambios)

```

Crea una calculadora extensible

importancia: 5

Crea una función `Calculator` que cree objetos calculadores "extensibles".

La actividad consiste de dos partes.

1.

Primero, implementar el método `calculate(str)` que toma un string como `"1 + 2"` en el formato "NUMERO operador NUMERO" (delimitado por espacios) y devuelve el resultado. Debe entender más + y menos -.

Ejemplo de uso:

```

1 let calc = new Calculator;
2 alert( calc.calculate("3 + 7") ); //
3 10

```

2.

Luego agrega el método `addMethod(name, func)` que enseñe a la calculadora una nueva operación. Toma el operador `name` y la función con dos argumentos `func(a,b)` que lo implementa.

Por ejemplo, vamos a agregar la multiplicación `*`, division `/` y potencia `**`:

```

1 let powerCalc = new Calculator;
2 powerCalc.addMethod("*", (a, b) => a * b);
3 powerCalc.addMethod("/", (a, b) => a / b);
4 powerCalc.addMethod("**", (a, b) => a ** b);
5
6 let result = powerCalc.calculate("2 ** 3");
7 alert( result ); // 8

```

- Sin paréntesis ni expresiones complejas en esta tarea.
- Los números y el operador deben estar delimitados por exactamente un espacio.
- Puede haber manejo de errores si quisieras agregarlo.

Mapa a nombres

importancia: 5

Tienes un array de objetos `user`, cada uno tiene `user.name`. Escribe el código que lo convierta en un array de nombres.

Por ejemplo:

```

1 let john = { name: "John", age: 25 };
2 let pete = { name: "Pete", age: 30 };
3 let mary = { name: "Mary", age: 28 };
4 let users = [ john, pete, mary
5 ];
6
7 let names = /* ... tu código */
8 alert( names ); // John, Pete,
9 Mary

```

Mapa a objetos

importancia: 5

Tienes un array de objetos `user`, cada uno tiene `name`, `surname` e `id`.

Escribe el código para crear otro array a partir de este, de objetos con `id` y `fullName`, donde `fullName` es generado a partir de `name` y `surname`.

Por ejemplo:

```

1 let john = { name: "John", surname: "Smith", id: 1 };
2 let pete = { name: "Pete", surname: "Hunt", id: 2 };
3 let mary = { name: "Mary", surname: "Key", id: 3 };
4
5 let users = [ john, pete, mary ];
6
7 let usersMapped = /* ... tu código ... */
8
9 /*
10 usersMapped = [
11   { fullName: "John Smith", id: 1 },
12   { fullName: "Pete Hunt", id: 2 },
13   { fullName: "Mary Key", id: 3 }
14 ]
15 */ alert( usersMapped[0].id ) // 1 alert(
16 usersMapped[0].fullName ) // John Smith
17
18

```

Entonces, en realidad lo que necesitas es mapear un array de objetos a otro. Intenta usar `=>` en este caso. Hay un pequeño truco.

Ordena usuarios por edad

importancia: 5

Escribe la función `sortByAge(users)` que cree un array de objetos con la propiedad `age` y los ordene según `age`.

Por ejemplo:

```

1 let john = { name: "John", age: 25 };
2 let pete = { name: "Pete", age: 30 };
3 let mary = { name: "Mary", age: 28 };
4
5 let arr = [ pete, john, mary ];
6
7 sortByAge(arr);
8
9 // ahora: [john, mary, pete]
10 alert(arr[0].name); // John
11 alert(arr[1].name); // Mary
12 alert(arr[2].name); // Pete

```

Barajar un array

importancia: 3

Escribe la función `shuffle(array)` que baraje (reordene de forma aleatoria) los elementos del array.

Múltiples ejecuciones de `shuffle` puede conducir a diferentes órdenes de elementos. Por ejemplo:

```
1 let arr = [1, 2, 3];
2
3 shuffle(arr);
4 // arr = [3, 2, 1]
5
6 shuffle(arr);
7 // arr = [2, 1, 3]
8
9 shuffle(arr);
10 // arr = [3, 1, 2]
11 // ...
```

Todos los reordenamientos de elementos tienen que tener la misma probabilidad. Por ejemplo, `[1,2,3]` puede ser reordenado como `[1,2,3]` o `[1,3,2]` o `[3,1,2]` etc, con igual probabilidad en cada caso.

Obtener edad promedio

importancia: 4

Escribe la función `getAverageAge(users)` que obtenga un array de objetos con la propiedad `age` y devuelva el promedio de `age`.

La fórmula de promedio es $(age1 + age2 + \dots + ageN) / N$.

Por ejemplo:

```
1 let john = { name: "John", age: 25 };
2 let pete = { name: "Pete", age: 30 };
3 let mary = { name: "Mary", age: 29 };
4
5 let arr = [ john, pete, mary ];
6 alert( getAverageAge(arr) ); // (25 + 30 + 29) / 3 =
7 28
```

Filtrar elementos únicos de un array

importancia: 4

Partiendo del array `arr`.

Crea una función `unique(arr)` que devuelva un array con los elementos que se encuentran una sola vez dentro de `arr`.

Por ejemplo:

```

1
2 function unique(arr) {
3   /* tu código */
4 } let strings = ["Hare", "Krishna", "Hare",
5 "Krishna",
6 "Krishna", "Krishna", "Hare", "Hare", ":-0"
7 ]; alert( unique(strings) ); // Hare,
8 Krishna, :-0
9

```

Crea un objeto a partir de un array

importancia: 4

Supongamos que recibimos un array de usuarios con la forma `{id:..., name:..., age:...}`.

Crea una función `groupById(arr)` que cree un objeto, con `id` como clave (key) y los elementos del array como valores.

Por ejemplo:

```

1
2
3 let users = [
4   {id: 'john', name: "John Smith", age: 20},
5   {id: 'ann', name: "Ann Smith", age: 24},
6   {id: 'pete', name: "Pete Peterson", age: 31},
7 ];
8 let usersById = groupById(users);
9 /*
10 // después de llamar a la función deberíamos tener:
11 usersById =
12 {
13   john: {id: 'john', name: "John Smith", age: 20},
14   ann: {id: 'ann', name: "Ann Smith", age: 24},
15   pete: {id: 'pete', name: "Pete Peterson", age: 31},
16 } */
17

```

Dicha función es realmente útil cuando trabajamos con información del servidor.

Para esta actividad asumimos que cada `id` es único. No existen dos elementos del array con el mismo `id`.

Usa el método de array `.reduce` en la solución.