

INTRODUCCION A ARRAYS EN JAVASCRIPT

1

2º DAW - Desarrollo Web en Entorno Cliente

¿Qué son los arrays?

- A menudo es necesario una colección ordenada, donde tenemos un 1ro, un 2do, un 3er elemento y así sucesivamente. Por ejemplo, necesitamos almacenar una lista de algo: usuarios, bienes, elementos HTML, etc.
- No es conveniente usar objetos aquí, porque no proveen métodos para manejar el orden de los elementos. No podemos insertar una nueva propiedad “entre” los existentes. Los objetos no están hechos para eso. Existe una estructura llamada Array (llamada en español arreglo o matriz/vector) para almacenar colecciones ordenadas.
- Hay dos sintaxis para crear un array vacío:

```
1 let arr = new Array();  
2 let arr = [];
```

```
1 let fruits = ["Apple", "Orange", "Plum"];
```

- Casi siempre se usa la segunda. Podemos suministrar elementos iniciales entre los corchetes
- Un array puede almacenar elementos de cualquier tipo. Por ejemplo:

```
1 // mezcla de valores  
2 let arr = [ 'Apple', { name: 'John' }, true, function() { alert('hello'); } ];  
3  
4 // obtener el objeto del índice 1 y mostrar su nombre  
5 alert( arr[1].name ); // John  
6  
7 // obtener la función del índice 3 y ejecutarla  
8 arr[3](); // hello
```

Los elementos del array están numerados comenzando desde cero. Podemos obtener un elemento por su número entre corchetes:

```
1 let fruits = ["Apple", "Orange", "Plum"];
2
3 alert( fruits[0] ); // Apple
4 alert( fruits[1] ); // Orange
5 alert( fruits[2] ); // Plum
```

Podemos reemplazar un elemento:

```
1 fruits[2] = 'Pear'; // ahora ["Apple", "Orange", "Pear"]
```

...o agregar uno nuevo al array:

```
1 fruits[3] = 'Lemon'; // ahora ["Apple", "Orange", "Pear", "Lemon"]
```

La cuenta total de elementos en el array es su longitud `length`:

```
1 let fruits = ["Apple", "Orange", "Plum"];
2
3 alert( fruits.length ); // 3
```

También podemos usar `alert` para mostrar el array completo.

```
1 let fruits = ["Apple", "Orange", "Plum"];
2
3 alert( fruits ); // Apple,Orange,Plum
```

Obtener los últimos elementos con “at”

Digamos que queremos el último elemento de un array. Algunos lenguajes de programación permiten el uso de índices negativos para este propósito, como `fruits[-1]`.

Aunque en JavaScript esto no funcionará. El resultado será `undefined`, porque el índice de los corchetes es tratado literalmente. Podemos calcular explícitamente el último índice y luego acceder al elemento: `fruits[fruits.length - 1]`.

```
1 let fruits = ["Apple", "Orange", "Plum"];
2
3 alert( fruits[fruits.length-1] ); // Plum
```

Un poco engorroso, ¿no es cierto? Necesitamos escribir el nombre de la variable dos veces. Afortunadamente, hay una sintaxis más corta: `fruits.at(-1)`:

```
1 let fruits = ["Apple", "Orange", "Plum"];
2
3 // es lo mismo que fruits[fruits.length-1]
4 alert( fruits.at(-1) ); // Plum
```

En otras palabras, `arr.at(i)`:

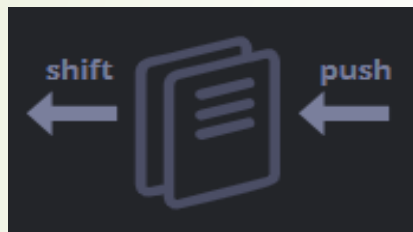
- es exactamente lo mismo que `arr[i]`, si $i \geq 0$.

para valores negativos de i , salta hacia atrás desde el final del array.

Métodos pop/push, shift/unshift

Una cola es uno de los usos más comunes de un array. En ciencias de la computación, significa una colección ordenada de elementos que soportan dos operaciones:

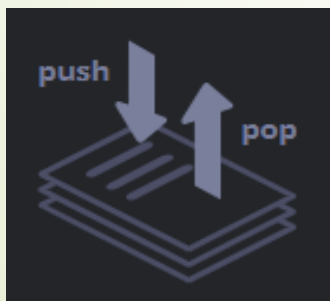
- push inserta un elemento al final.
- shift obtiene el elemento del principio, avanzando la cola, y así el segundo elemento se vuelve primero.



Los arrays soportan ambas operaciones. Por ejemplo, una cola de mensajes que necesitamos mostrar en pantalla. Hay otro caso de uso para los arrays – la estructura de datos llamada pila. Ella soporta dos operaciones:

- push agrega un elemento al final.
- pop toma un elemento desde el final.

Entonces los elementos nuevos son agregados o tomados siempre desde el “final”. Una pila es usualmente mostrada como un mazo de cartas, donde las nuevas cartas son agregadas al tope o tomadas desde el tope:



Para las pilas, la última introducida es la primera en ser recibida, en inglés esto es llamado principio LIFO (Last-In-First-Out, última en entrar primera en salir). Para las colas, tenemos FIFO (First-In-First-Out primera en entrar, primera en salir).

Los arrays en JavaScript pueden trabajar como colas o pilas. Ellos permiten agregar/quitar elementos al/del principio o al/del final. Métodos que trabajan sobre el final del array:

Pop → Extrae el último elemento del array y lo devuelve:

```
1 let fruits = ["Apple", "Orange", "Pear"];
2
3 alert( fruits.pop() ); // quita "Pear" y lo muestra en un alert
4
5 alert( fruits ); // Apple, Orange
```

Tanto `fruits.pop()` como `fruits.at(-1)` devuelven el último elemento del array, pero `fruits.pop()` también modifica el array eliminando tal elemento.

Push → Agrega el elemento al final del array:

```
1 let fruits = ["Apple", "Orange"];
2
3 fruits.push("Pear");
4
5 alert( fruits ); // Apple, Orange, Pear
```


El llamado a `fruits.push(...)` es igual a `fruits[fruits.length] =` Métodos que trabajan con el principio del array:

Shift → Extrae el primer elemento del array y lo devuelve:

```
1 let fruits = ["Apple", "Orange", "Pear"];
2
3 alert( fruits.shift() ); // quita Apple y lo muestra en un alert
4
5 alert( fruits ); // Orange, Pear
```

Unshift → Agrega el elemento al principio del array:

```
1 let fruits = ["Orange", "Pear"];
2
3 fruits.unshift('Apple');
4
5 alert( fruits ); // Apple, Orange, Pear
```

Los métodos `push` y `unshift` pueden agregar múltiples elementos de una vez:

```
1 let fruits = ["Apple"];
2
3 fruits.push("Orange", "Peach");
4 fruits.unshift("Pineapple", "Lemon");
5
6 // ["Pineapple", "Lemon", "Apple", "Orange", "Peach"]
7 alert( fruits );
```

Interiores

Un array es una clase especial de objeto. Los corchetes usados para acceder a una propiedad `arr[0]` vienen de la sintaxis de objeto. Son esencialmente lo mismo que `obj[key]`, donde `arr` es el objeto mientras los números son usados como claves. Ellos extienden los objetos proveyendo métodos especiales para trabajar con colecciones ordenadas de datos y también la propiedad `length`. Pero en el corazón es aún un objeto.

Recuerde, solo hay ocho tipos de datos básicos en JavaScript (consulte el capítulo Tipos de datos para obtener más información). Array es un objeto y, por tanto, se comporta como un objeto. Por ejemplo, es copiado por referencia:

```
1 let fruits = ["Banana"]
2
3 let arr = fruits; // copiado por referencia (dos variables referencian al mismo array)
4
5 alert( arr === fruits ); // true
6
7 arr.push("Pear"); // modifica el array por referencia
8
9 alert( fruits ); // Banana, Pear - ahora con 2 items
```

...Pero lo que hace a los array realmente especiales es su representación interna. El motor trata de almacenarlos en áreas de memoria contigua, uno tras otro, justo como muestra la ilustración en este capítulo. Hay otras optimizaciones también para hacer que los arrays trabajen verdaderamente rápido.

Pero todo esto se puede malograr si dejamos de trabajarlos como arrays de colecciones ordenadas y comenzamos a usarlos como si fueran objetos comunes. Por ejemplo, técnicamente podemos hacer esto:

```
1 let fruits = []; // crea un array
2
3 fruits[99999] = 5; // asigna una propiedad con un índice mucho mayor que su longitud
4
5 fruits.age = 25; // crea una propiedad con un nombre arbitrario
```

Esto es posible porque los arrays son objetos en su base. Podemos agregar cualquier propiedad en ellos. Pero el motor verá que estamos tratándolo como un objeto común. Las optimizaciones específicas no son aptas para tales casos y serán desechadas, y sus beneficios desaparecerán.

Las formas de malograr un array:

- Agregar una propiedad no numérica como `arr.test = 5`.
- Generar agujeros como: agregar `arr[0]` y luego `arr[1000]` (y nada entre ellos).
- Llenar el array en orden inverso, como `arr[1000]`, `arr[999]` y así.

Piensa en los arrays como estructuras especiales para trabajar con datos ordenados. Ellos proveen métodos especiales para ello. Los arrays están cuidadosamente afinados dentro de los motores JavaScript para funcionar con datos ordenados contiguos, por favor úsalos de esa manera. Y si necesitas claves arbitrarias, hay altas chances de que en realidad necesites objetos comunes `{}`.

Performance

Los métodos push/pop son rápidos, mientras que shift/unshift son lentos.

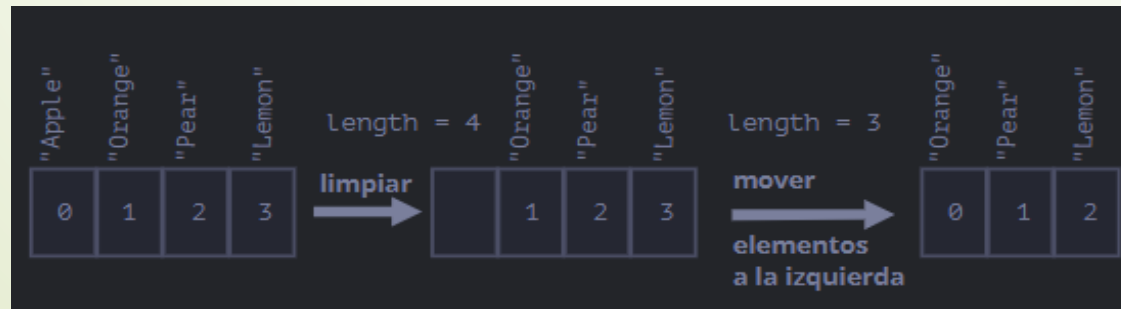


¿Por qué es más rápido trabajar con el final del array que con el principio? Veamos qué pasa durante la ejecución:

```
1 fruits.shift(); // toma 1 elemento del principio
```

No es suficiente tomar y eliminar el elemento con el índice 0. Los demás elementos necesitan ser reenumerados también. La operación shift debe hacer 3 cosas:

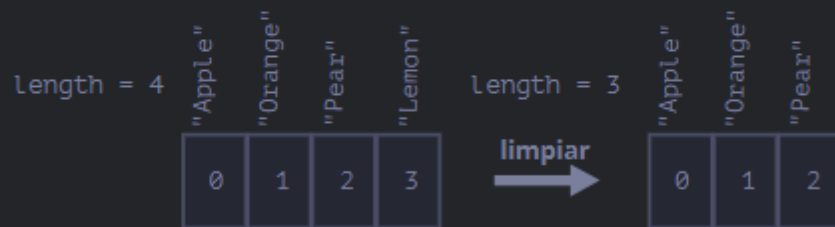
1. Remove el elemento con índice 0.
2. Mover todos los elementos hacia la izquierda y reenumerarlos: desde el índice 1 a 0, de 2 a 1 y así sucesivamente.
3. Actualizar la longitud: la propiedad length.



Cuanto más elemento haya en el array, más tiempo tomará moverlos, más operaciones en memoria.

Algo similar ocurre con unshift: para agregar un elemento al principio del array, necesitamos primero mover todos los elementos hacia la derecha, incrementando sus índices. ¿Y qué pasa con push/pop? Ellos no necesitan mover nada. Para extraer un elemento del final, el método pop limpia el índice y acorta length. Las acciones para la operación pop:

```
1 fruits.pop(); // toma 1 elemento del final
```



El método pop no necesita mover nada, porque los demás elementos mantienen sus índices. Es por ello que es muy rápido. Algo similar ocurre con el método push.

Bucles

Una de las formas más viejas de iterar los items de un array es el bucle for sobre sus índices:

```
1 let arr = ["Apple", "Orange", "Pear"];
2
3 for (let i = 0; i < arr.length; i++) {
4   alert( arr[i] );
5 }
```

Pero para los arrays también hay otra forma de bucle, for..of:

```
1 let fruits = ["Apple", "Orange", "Plum"];
2
3 // itera sobre los elementos del array
4 for (let fruit of fruits) {
5   alert( fruit );
6 }
```

for..of no da acceso al número del elemento en curso, solamente a su valor, pero en la mayoría de los casos eso es suficiente. Y es más corto. Técnicamente, y porque los arrays son objetos, es también posible usar for..in:

Pero es una mala idea. Existen problemas potenciales con esto:

1. El bucle `for..in` itera sobre todas las propiedades, no solo las numéricas.
2. Existen objetos “simil-array” en el navegador y otros ambientes que parecen arrays. Esto es, tienen `length` y propiedades indexadas, pero pueden también tener propiedades no numéricas y métodos que usualmente no necesitamos. Y el bucle `for..in` los listará. Entonces si necesitamos trabajar con objetos simil-array, estas propiedades “extras” pueden volverse un problema.
3. El bucle `for..in` está optimizado para objetos genéricos, no para arrays, y es de 10 a 100 veces más lento. Por supuesto es aún muy rápido. Una optimización puede que solo sea importante en cuellos de botella, pero necesitamos ser concientes de la diferencia.

En general, no deberíamos usar `for..in` en arrays.

Acerca de “length”

La propiedad `length` automáticamente se actualiza cuando se modifica el array. Para ser precisos, no es la cuenta de valores del array sino el mayor índice más uno. Por ejemplo, un elemento simple con un índice grande da una longitud grande:

```
1 let fruits = [];  
2 fruits[123] = "Apple";  
3  
4 alert( fruits.length ); // 124
```

Una cosa interesante acerca de la propiedad `length` es que se puede sobrescribir.

Si la incrementamos manualmente, nada interesante ocurre. Pero si la decrementamos, el array se trunca. El proceso es irreversible, aquí el ejemplo:

```
1 let arr = [1, 2, 3, 4, 5];
2
3 arr.length = 2; // truncamos a 2 elementos
4 alert( arr ); // [1, 2]
5
6 arr.length = 5; // reponemos la longitud length
7 alert( arr[3] ); // undefined: el valor no se recupera
```

Entonces la forma más simple de limpiar un array es: `arr.length = 0;`

new Array()

Hay una sintaxis más para crear un array:

```
1 let arr = new Array("Apple", "Pear", "etc");
```

Es raramente usada porque con corchetes [] es más corto. También hay una característica peculiar con ella. Si new Array es llamado con un único argumento numérico, se crea un array sin items, pero con la longitud "length" dada. Veamos cómo uno puede dispararse en el pie:

```
1 let arr = new Array(2); // ¿Crearé un array de [2]?  
2  
3 alert( arr[0] ); // undefined! sin elementos.  
4  
5 alert( arr.length ); // longitud 2
```

Para evitar sorpresas solemos usar corchetes, salvo que sepamos lo que estamos haciendo.

Arrays multidimensionales

Los arrays pueden tener items que a su vez sean arrays. Podemos usarlos como arrays multidimensionales, por ejemplo para almacenar matrices:

```
1 let matrix = [  
2   [1, 2, 3],  
3   [4, 5, 6],  
4   [7, 8, 9]  
5 ];  
6  
7 alert( matrix[1][1] ); // 5, el elemento central
```

toString

Los arrays tienen su propia implementación del método toString que devuelve una lista de elementos separados por coma. Por ejemplo:

```
1 let arr = [1, 2, 3];  
2  
3 alert( arr ); // 1,2,3  
4 alert( String(arr) === '1,2,3' ); // true
```

Probemos esto también:

```
1 alert( [] + 1 ); // "1"  
2 alert( [1] + 1 ); // "11"  
3 alert( [1,2] + 1 ); // "1,21"
```

Los arrays no tienen Symbol.toPrimitive ni un valueOf viable, ellos implementan la conversión toString solamente, así [] se vuelve una cadena vacía, [1] se vuelve "1" y [1,2] se vuelve "1,2".

Cuando el operador binario más "+" suma algo a una cadena, lo convierte a cadena también, entonces lo siguiente se ve así:

```
1 alert( "" + 1 ); // "1"  
2 alert( "1" + 1 ); // "11"  
3 alert( "1,2" + 1 ); // "1,21"
```

No compares arrays con ==

Las arrays en JavaScript, a diferencia de otros lenguajes de programación, no deben ser comparadas con el operador `==`. Este operador no tiene un tratamiento especial para arrays, trabaja con ellas como con cualquier objeto. Recordemos las reglas:

- Dos objetos son iguales `==` solo si hacen referencia al mismo objeto.
- Si uno de los argumentos de `==` es un objeto y el otro es un primitivo, entonces el objeto se convierte en primitivo, como se explica en el capítulo Conversión de objeto a valor primitivo.
- ...Con la excepción de `null` y `undefined` que son iguales `==` entre sí y nada más.

La comparación estricta `===` es aún más simple, ya que no convierte tipos. Entonces, si comparamos arrays con `==`, nunca son iguales, a no ser que comparemos dos variables que hacen referencia exactamente a la misma array. Por ejemplo:

```
1 alert( [] == [] ); // falso
2 alert( [0] == [0] ); // falso
```

Estas arrays son técnicamente objetos diferentes. Así que no son iguales. El operador `==` no hace comparaciones de elemento a elemento. Comparaciones con primitivos también pueden dar resultados aparentemente extraños:

```
1 alert( 0 == [] ); // verdadero
2
3 alert('0' == [] ); // falso
```

Aquí, en ambos casos, comparamos un primitivo con un objeto array. Entonces la array [] se convierte a primitivo para el propósito de comparar y se convierte en una string vacía "". Luego el proceso de comparación continúa con los primitivos, como se describe en el capítulo Conversiones de Tipos:

```
1 // después de que [] se convierta en ''
2 alert( 0 == '' ); // verdadero, ya que '' se convierte en el número 0
3
4 alert('0' == '' ); // falso, sin conversión de tipos, strings diferentes
```

Entonces, ¿cómo comparamos arrays? Simple: no utilices el operador ==. En lugar, compáralas elemento a elemento en un bucle o utilizando métodos de iteración explicados en el siguiente capítulo.

RESUMEN

- Los arrays son una clase especial de objeto, adecuados para almacenar y manejar items de datos ordenados. La declaración:

```
1 // corchetes (lo usual)
2 let arr = [item1, item2...];
3
4 // new Array (excepcionalmente raro)
5 let arr = new Array(item1, item2...);
```

- El llamado a `new Array(number)` crea un array con la longitud dada, pero sin elementos.
 - La propiedad `length` es la longitud del array o, para ser preciso, el último índice numérico más uno. Se autoajusta al usar los métodos de array.
 - Si acortamos `length` manualmente, el array se trunca.
- Obtener los elementos:
 - Podemos obtener un elemento por su índice, como `arr[0]`
 - También podemos usar el método `at(i)`, que permite índices negativos. Para valores negativos de `i`, cuenta hacia atrás desde el final del array. Cuando `i >= 0`, funciona igual que `arr[i]`.
- Podemos usar un array como una pila “deque” o “bicola” con las siguientes operaciones:
 - `push(...items)` agrega items al final.
 - `pop()` remueve el elemento del final y lo devuelve.
 - `shift()` remueve el elemento del principio y lo devuelve.
 - `unshift(...items)` agrega items al principio.
- Para iterar sobre los elementos de un array:
 - `for (let i=0; i<arr.length; i++)` – lo más rápido, compatible con viejos navegadores.
 - `for (let item of arr)` – la sintaxis moderna para items solamente.
 - `for (let i in arr)` – nunca lo uses.
- Para comparar arrays, no uses el operador `==` (como tampoco `>`, `<` y otros), ya que no tienen un tratamiento especial para arrays. Lo manejan como cualquier objeto y no es lo que normalmente queremos. En su lugar puedes utilizar el bucle `for..of` para comparar arrays elemento a elemento.

TAREAS

¿El array es copiado?

¿Qué va a mostrar este código?

```
let fruits = ["Apples", "Pear", "Orange"];

// introduce un valor nuevo dentro de una copia
let shoppingCart = fruits;
shoppingCart.push("Banana");

// ¿Qué hay en "fruits"?
alert( fruits.length ); // ¿?
```

LLamados en un contexto de array

¿Cuál es el resultado y por qué?

```
let arr = ["a", "b"];

arr.push(function() {
  alert( this );
});

arr[2](); // ?
```


Operaciones en arrays.

Tratemos 5 operaciones de array.

Crear un array `styles` con los items "Jazz" y "Blues".

Agregar "Rock-n-Roll" al final.

Reemplazar el valor en el medio por "Classics". Tu código para encontrar el valor medio debe funcionar con cualquier array de longitud impar.

Quitar el primer valor del array y mostrarlo.

Anteponer Rap y Reggae al array.

El array durante el proceso:

Jazz, Blues

Jazz, Blues, Rock-n-Roll

Jazz, Classics, Rock-n-Roll

Classics, Rock-n-Roll

Rap, Reggae, Classics, Rock-n-Roll

Suma de números ingresados

Escribe una función `sumInput()` que:

Pida al usuario valores usando `prompt` y los almacene en el array.

Termine de pedirlos cuando el usuario ingrese un valor no numérico, una cadena vacía, o presione "Escape".

Calcule y devuelva la suma de los items del array.

P.D. Un cero `0` es un número válido, por favor no detengas los ingresos con el cero.

[Ejecutar el demo](#)

Subarray máximo

La entrada es un array de números, por ejemplo `arr = [1, -2, 3, 4, -9, 6]`.

La tarea es: encuentra el subarray contiguo de items de `arr` con la suma máxima.

Escribe la función `getMaxSubSum(arr)` que devuelva tal sumo.

Por ejemplo:

```
getMaxSubSum([-1, 2, 3, -9]) == 5 (la suma de items resaltados)
```

```
getMaxSubSum([2, -1, 2, 3, -9]) == 6
```

```
getMaxSubSum([-1, 2, 3, -9, 11]) == 11
```

```
getMaxSubSum([-2, -1, 1, 2]) == 3
```

```
getMaxSubSum([100, -9, 2, -3, 5]) == 100
```

```
getMaxSubSum([1, 2, 3]) == 6 (toma todo)
```

Si todos los elementos son negativos, significa que que no tomamos ninguno (el subarray está vacío), entonces la suma es cero:

```
getMaxSubSum([-1, -2, -3]) = 0
```

Trata de pensar un solución rápida: $O(n^2)$ o incluso $O(n)$ si puedes.

[Abrir en entorno controlado con pruebas.](#)

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/remover ítems**
- 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.

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
5 alert( arr[1] ); // 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 remplazar 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 arr.splice(1, 1); // desde el índice 1, remover 1 elemento
4
5 alert( arr ); // ["Yo", "JavaScript"]
```

¿Fácil, no? Empezando desde el índice 1 removió 1 elemento. En el próximo ejemplo removemos 3 elementos y los reemplazamos con otros 2:

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

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

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

El método splice también es capaz de insertar elementos sin remover ningún otro. Para eso necesitamos establecer deleteCount en 0:

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

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  
5 alert( arr.slice(-2) ); // s,t (copia desde -2 hasta el 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  
9 // crea un array a partir de: arr y [3,4], luego agrega los valores 5 y 6  
10 alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

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 };
7
8 alert( arr.concat(arrayLike) ); // 1,2,[object Object]
```

...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 = {
4   0: "something",
5   1: "else",
6   [Symbol.isConcatSpreadable]: true,
7   length: 2
8 };
9
10 alert( arr.concat(arrayLike) ); // 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` y `arr.includes` tienen una sintaxis similar 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.includes(item, from)` – busca `item` comenzando desde el índice `from`, devuelve `true` en caso de ser encontrado.

Usualmente estos métodos se usan con un solo argumento: el item a buscar. De manera predeterminada, la búsqueda es desde el principio. 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
7 alert( arr.includes(1) ); // true
```

Tener en cuenta que el método usa la comparación estricta (`===`). Por lo tanto, si buscamos `false`, encontrará exactamente `false` y no cero. Si queremos comprobar si un elemento existe en el array, pero no necesitamos saber su ubicación exacta, es preferible usar `arr.includes`

El método `arr.lastIndexOf` es lo mismo que `indexOf`, pero busca de derecha a izquierda.

```
1 let fruits = ['Apple', 'Orange', 'Apple']
2
3 alert( fruits.indexOf('Apple') ); // 0 (primera "Apple")
4 alert( fruits.lastIndexOf('Apple') ); // 2 (última "Apple")
```

find y findIndex/findLastIndex

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  
7 let user = users.find(item => item.id == 1);  
8  
9 alert(user.name); // Celina
```

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

1. El método `arr.findIndex` tiene la misma sintaxis, pero devuelve el índice donde el elemento fue encontrado en lugar del elemento en sí. Devuelve `-1` cuando no lo encuentra.
2. El método `arr.findLastIndex` es como `findIndex`, pero busca de derecha a izquierda, similar a `lastIndexOf`.

Un ejemplo:

```
1 let users = [  
2   {id: 1, name: "John"},  
3   {id: 2, name: "Pete"},  
4   {id: 3, name: "Mary"},  
5   {id: 4, name: "John"}  
6 ];  
7  
8 // Encontrar el índice del primer John  
9 alert(users.findIndex(user => user.name == 'John')); // 0  
10  
11 // Encontrar el índice del último John  
12 alert(users.findLastIndex(user => user.name == 'John')); // 3
```


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 continua  
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  
10 alert(someUsers.length); // 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 => item.length);  
2 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 este 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
6 alert( arr ); // 1, 15, 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:

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

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 );  
3   return a - b;  
4 });
```

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

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

`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
3 let str = arr.join(';'); // une el array en una string usando ;
4
5 alert( str ); // 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`.

La forma más simple de entender algo es con un ejemplo. Acá tenemos la suma de un array en una línea:

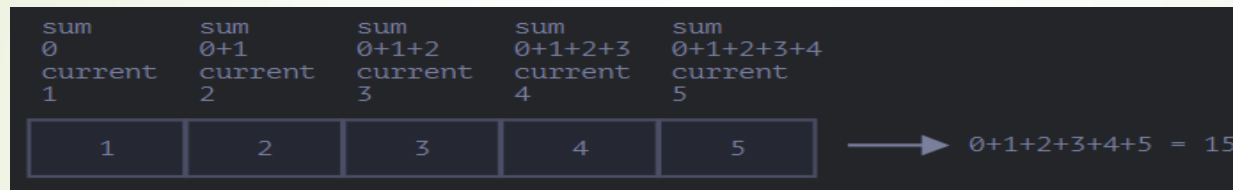
```
1 let arr = [1, 2, 3, 4, 5];  
2  
3 let result = arr.reduce((sum, current) => sum + current, 0);  
4  
5 alert(result); // 15
```

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 inicial (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:



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

```

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

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

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

La llamada a users.filter(army.canJoin, army) puede ser reemplazada con users.filter(user => army.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(value)` comprueba si value 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) {  
    return arr1.length === arr2.length && arr1.every((value, index) => value === arr2[index]);  
}  
  
alert( arraysEqual([1, 2], [1, 2])); // true
```
- `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*

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:

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

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

[Abrir en entorno controlado con pruebas.](#)

Filtrar un rango

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:

```
let arr = [5, 3, 8, 1];

let filtered = filterRange(arr, 1, 4);

alert( filtered ); // 3,1 (valores dentro del rango)

alert( arr ); // 5,3,8,1 (array original no modificado)
```

[Abrir en entorno controlado con pruebas.](#)

Filtrar rango "en el lugar"

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: $a \leq \text{arr}[i] \leq b$.

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

Por ejemplo:

```
let arr = [5, 3, 8, 1];

filterRangeInPlace(arr, 1, 4); // remueve los números excepto
aquellos entre 1 y 4

alert( arr ); // [3, 1]
```

Abrir en entorno controlado con pruebas.

Ordenar en orden decreciente

```
let arr = [5, 2, 1, -10, 8];

// ... tu código para ordenar en orden decreciente

alert( arr ); // 8, 5, 2, 1, -10
```

Copia y ordena un array

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.

```
let arr = ["HTML", "JavaScript", "CSS"];

let sorted = copySorted(arr);

alert( sorted ); // CSS, HTML, JavaScript
alert( arr ); // HTML, JavaScript, CSS (sin cambios)
```

Mapa a nombres

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

Por ejemplo:

```
let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let users = [ john, pete, mary ];

let names = /* ... tu código */

alert( names ); // John, Pete, Mary
```

Crea una calculadora extensible

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:

```
let calc = new Calculator;  
  
alert( calc.calculate("3 + 7") ); // 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 **:

```
let powerCalc = new Calculator;  
powerCalc.addMethod("*", (a, b) => a * b);  
powerCalc.addMethod("/", (a, b) => a / b);  
powerCalc.addMethod("**", (a, b) => a ** b);  
  
let result = powerCalc.calculate("2 ** 3");  
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.

[Abrir en entorno controlado con pruebas.](#)

Mapa a objetos

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:

```
let john = { name: "John", surname: "Smith", id: 1 };
let pete = { name: "Pete", surname: "Hunt", id: 2 };
let mary = { name: "Mary", surname: "Key", id: 3 };
```

```
let users = [ john, pete, mary ];
```

```
let usersMapped = /* ... tu código ... */
```

```
/*
usersMapped = [
  { fullName: "John Smith", id: 1 },
  { fullName: "Pete Hunt", id: 2 },
  { fullName: "Mary Key", id: 3 }
]
*/
```

```
alert( usersMapped[0].id ) // 1
```

```
alert( usersMapped[0].fullName ) // John Smith
```

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

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

Por ejemplo:

```
let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let arr = [ pete, john, mary ];

sortByAge(arr);

// ahora: [john, mary, pete]
alert(arr[0].name); // John
alert(arr[1].name); // Mary
alert(arr[2].name); // Pete
```

Obtener edad promedio

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:

```
let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 29 };

let arr = [ john, pete, mary ];

alert( getAverageAge(arr) ); // (25 + 30 + 29) / 3 = 28
```

Barajar un array

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:

```
let arr = [1, 2, 3];
```

```
shuffle(arr);
```

```
// arr = [3, 2, 1]
```

```
shuffle(arr);
```

```
// arr = [2, 1, 3]
```

```
shuffle(arr);
```

```
// arr = [3, 1, 2]
```

```
// ...
```

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.

Filtrar elementos únicos de un array

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:

```
function unique(arr) {  
  /* tu código */  
}  
  
let strings = ["Hare", "Krishna", "Hare", "Krishna",  
  "Krishna", "Krishna", "Hare", "Hare", ":-0"]  
];  
  
alert( unique(strings) ); // Hare, Krishna, :-0
```

[Abrir en entorno controlado con pruebas.](#)

Crea un objeto a partir de un array

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:

```
let users = [
  {id: 'john', name: "John Smith", age: 20},
  {id: 'ann', name: "Ann Smith", age: 24},
  {id: 'pete', name: "Pete Peterson", age: 31},
];

let usersById = groupById(users);

/*
// después de llamar a la función deberíamos tener:

usersById = {
  john: {id: 'john', name: "John Smith", age: 20},
  ann: {id: 'ann', name: "Ann Smith", age: 24},
  pete: {id: 'pete', name: "Pete Peterson", age: 31},
}
*/
```

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.

[Abrir en entorno controlado con pruebas.](#)

Iterables

- Los objetos iterables son una generalización de arrays. Es un concepto que permite que cualquier objeto pueda ser utilizado en un bucle `for..of`.
- Por supuesto, las matrices o arrays son iterables. Pero hay muchos otros objetos integrados que también lo son. Por ejemplo, las cadenas o strings son iterables también. Como veremos, muchos operadores y métodos se basan en la iterabilidad.
- Si un objeto no es técnicamente una matriz, pero representa una colección (lista, conjunto) de algo, entonces el uso de la sintaxis `for..of` es una gran forma de recorrerlo. Veamos cómo funciona.

Symbol.iterator

Por ejemplo: tenemos un objeto que no es un array, pero parece adecuado para `for..of`. Como un objeto `range` que representa un intervalo de números:

```
1  let range = {  
2    from: 1,  
3    to: 5  
4  };  
5  
6  // Queremos que el for..of funcione de la siguiente manera:  
7  // for(let num of range) ... num=1,2,3,4,5
```

Para hacer que el objeto `range` sea iterable (y así permitir que `for..of` funcione) necesitamos agregarle un método llamado `Symbol.iterator` (un símbolo incorporado especial usado solo para realizar esa función).

1. Cuando se inicia `for..of`, éste llama al método `Symbol.iterator` una vez (o genera un error si no lo encuentra). El método debe devolver un iterador : un objeto con el método `next()`.
2. En adelante, `for..of` trabaja solamente con ese objeto devuelto.
3. Cuando `for..of` quiere el siguiente valor, llama a `next()` en ese objeto.
4. El resultado de `next()` debe tener la forma `{done: Boolean, value: any}`, donde `done=true` significa que el bucle ha finalizado; de lo contrario, el nuevo valor es `value`.

Aquí está la implementación completa de range:

```
1  let range = {
2    from: 1,
3    to: 5
4  };
5
6  // 1. Una llamada a for..of inicializa una llamada a esto:
7  range[Symbol.iterator] = function() {
8
9    // ... devuelve el objeto iterador:
10   // 2. En adelante, for..of trabaja solo con el objeto iterador debajo, pidiéndole l
11   return {
12     current: this.from,
13     last: this.to,
14
15     // 3. next() es llamado en cada iteración por el bucle for..of
16     next() {
17       // 4. debe devolver el valor como un objeto {done:.., value :...}
18       if (this.current <= this.last) {
19         return { done: false, value: this.current++ };
20       } else {
21         return { done: true };
22       }
23     }
24   };
25 };
26
27 // ¡Ahora funciona!
28 for (let num of range) {
29   alert(num); // 1, luego 2, 3, 4, 5
30 }
```

Note una característica fundamental de los iterables: separación de conceptos.

- El range en sí mismo no tiene el método next().
- En cambio, la llamada a range[Symbol.iterator]() crea un otro objeto llamado “iterador”, y su next() genera valores para la iteración.

Por lo tanto, el objeto iterador está separado del objeto sobre el que itera. Técnicamente, podríamos fusionarlos y usar el range mismo como iterador para simplificar el código. De esta manera:

```
1  let range = {
2    from: 1,
3    to: 5,
4
5    [Symbol.iterator]() {
6      this.current = this.from;
7      return this;
8    },
9
10   next() {
11     if (this.current <= this.to) {
12       return { done: false, value: this.current++ };
13     } else {
14       return { done: true };
15     }
16   }
17 };
18
19 for (let num of range) {
20   alert(num); // 1, luego 2, 3, 4, 5
21 }
```

Ahora `range[Symbol.iterator]()` devuelve el objeto range en sí: tiene el método `next()` necesario y recuerda el progreso de iteración actual en `this.current`. ¿Más corto? Sí. Y a veces eso también está bien.

La desventaja es que ahora es imposible tener dos bucles `for..of` corriendo sobre el objeto simultáneamente: compartirán el estado de iteración, porque solo hay un iterador: el objeto en sí. Pero dos `for-of` paralelos es algo raro, incluso en escenarios asíncronos.

String es iterable

Las matrices y cadenas son los iterables integrados más utilizados. En una cadena o string, el bucle `for..of` recorre sus caracteres:

```
1 for (let char of "test") {  
2   // Se dispara 4 veces: una vez por cada carácter  
3   alert( char ); // t, luego e, luego s, luego t  
4 }
```

¡Y trabaja correctamente con valores de pares sustitutos (codificación UTF-16)!

```
1 let str = 'X😄';  
2 for (let char of str) {  
3   alert( char ); // X, y luego 😄  
4 }
```

Llamar a un iterador explícitamente

Para una comprensión más profunda, veamos cómo usar un iterador explícitamente. Vamos a iterar sobre una cadena exactamente de la misma manera que `for..of`, pero con llamadas directas. Este código crea un iterador de cadena y obtiene valores de él “manualmente”:

```
1 let str = "Hola";
2
3 // hace lo mismo que
4 // for (let char of str) alert(char);
5
6 let iterator = str[Symbol.iterator]();
7
8 while (true) {
9   let result = iterator.next();
10  if (result.done) break;
11  alert(result.value); // retorna los caracteres uno por uno
12 }
```

Rara vez se necesita esto, pero nos da más control sobre el proceso que `for..of`. Por ejemplo, podemos dividir el proceso de iteración: iterar un poco, luego parar, hacer otra cosa y luego continuar.

Iterables y simil-array (array-like)

Los dos son términos oficiales que se parecen, pero son muy diferentes. Asegúrese de comprenderlos bien para evitar confusiones.

- Iterables son objetos que implementan el método `Symbol.iterator`, como se describió anteriormente.
- `simil-array` son objetos que tienen índices y longitud o `length`, por lo que se “ven” como arrays.

Cuando usamos JavaScript para tareas prácticas en el navegador u otros entornos, podemos encontrar objetos que son iterables o `array-like`, o ambos. Por ejemplo, las cadenas son iterables (`for..of` funciona en ellas) y `array-like` (tienen índices numéricos y `length`).

Pero un iterable puede que no sea `array-like`. Y viceversa, un `array-like` puede no ser iterable. Por ejemplo, `range` en el ejemplo anterior es iterable, pero no es `array-like` porque no tiene propiedades indexadas ni `length`.

Y aquí el objeto tiene forma de matriz, pero no es iterable:

```
1 let arrayLike = { // tiene índices y longitud => array-like
2   0: "Hola",
3   1: "Mundo",
4   length: 2
5 };
6
7 // Error (sin Symbol.iterator)
8 for (let item of arrayLike) {}
```

Tanto los iterables como los `array-like` generalmente no son arrays, no tienen “push”, “pop”, etc. Eso es bastante inconveniente si tenemos un objeto de este tipo y queremos trabajar con él como con una matriz. P.ej. nos gustaría trabajar con `range` utilizando métodos de matriz.

Array.from

Existe un método universal `Array.from` que toma un valor iterable o simil-array y crea un Array “real” a partir de él. De esta manera podemos llamar y usar métodos que pertenecen a una matriz. Por ejemplo:

```
1 let arrayLike = {
2   0: "Hola",
3   1: "Mundo",
4   length: 2
5 };
6
7 let arr = Array.from(arrayLike); // (*)
8 alert(arr.pop()); // Mundo (el método pop funciona)
```

`Array.from` en la línea (*) toma el objeto, y si es iterable o simil-array crea un nuevo array y copia allí todos los elementos. Lo mismo sucede para un iterable:

```
1 // suponiendo que range se toma del ejemplo anterior
2 let arr = Array.from(range);
3 alert(arr); // 1,2,3,4,5 (la conversión de matriz a cadena funciona)
```

La sintaxis completa para `Array.from` también nos permite proporcionar una función opcional de “mapeo”:

```
1 Array.from(obj[, mapFn, thisArg])
```

El segundo argumento opcional `mapFn` puede ser una función que se aplicará a cada elemento antes de agregarlo a la matriz, y `thisArg` permite establecer el `this` para ello. Por ejemplo:

```
1 // suponiendo que range se toma del ejemplo anterior
2
3 // el cuadrado de cada número
4 let arr = Array.from(range, num => num * num);
5
6 alert(arr); // 1,4,9,16,25
```

Aquí usamos `Array.from` para convertir una cadena en una matriz de caracteres:

```
1 let str = 'X😄';
2
3 // separa str en un array de caracteres
4 let chars = Array.from(str);
5
6 alert(chars[0]); // X
7 alert(chars[1]); // 😄
8 alert(chars.length); // 2
```

A diferencia de `str.split`, `Array.from` se basa en la naturaleza iterable de la cadena y, por lo tanto, al igual que `for..of`, funciona correctamente con pares sustitutos. Técnicamente aquí hace lo mismo que:

```
1 let str = 'X😄';
2
3 let chars = []; // Array.from internamente hace el mismo bucle
4 for (let char of str) {
5   chars.push(char);
6 }
7
8 alert(chars);
```

... Pero es más corto. Incluso podemos construir un segmento o slice compatible con sustitutos en él:

```
1 function slice(str, start, end) {
2   return Array.from(str).slice(start, end).join('');
3 }
4
5 let str = 'X😄𐄀';
6
7 alert( slice(str, 1, 3) ); // 😄𐄀
8
9 // el método nativo no admite pares sustitutos
10 alert( str.slice(1, 3) ); // garbage (dos piezas de diferentes pares sustitutos)
```

RESUMEN

- Los objetos que se pueden usar en `for..of` se denominan iterables.
 - Técnicamente, los iterables deben implementar el método llamado `Symbol.iterator`.
 - El resultado de `obj[Symbol.iterator]()` se llama iterador. Maneja el proceso de iteración adicional.
 - Un iterador debe tener el método llamado `next()` que devuelve un objeto `{done: Boolean, value: any}`, donde `done: true` marca el fin de la iteración; de lo contrario, `value` es el siguiente valor.
 - El método `Symbol.iterator` se llama automáticamente por `for..of`, pero también podemos llamarlo directamente.
 - Los iterables integrados, como cadenas o matrices, también implementan `Symbol.iterator`.
 - El iterador de cadena es capaz de manejar los pares sustitutos.
- Los objetos que tienen propiedades indexadas y longitud o `length` se llaman array-like. Dichos objetos también pueden tener otras propiedades y métodos, pero carecen de los métodos integrados de las matrices.
- Si miramos dentro de la especificación, veremos que la mayoría de los métodos incorporados suponen que funcionan con iterables o array-likes en lugar de matrices "reales", porque eso es más abstracto.
- `Array.from(obj[, mapFn, thisArg])` crea un verdadero Array de un obj iterable o array-like, y luego podemos usar métodos de matriz en él. Los argumentos opcionales `mapFn` y `thisArg` nos permiten aplicar una función a cada elemento.

Asignación desestructurante

- Las dos estructuras de datos más usadas en JavaScript son Object y Array.
 - Los objetos nos permiten crear una simple entidad que almacena items con una clave cada uno.
 - Los arrays nos permiten reunir items en una lista ordenada.
- Pero cuando los pasamos a una función, tal vez no necesitemos un objeto o array como un conjunto sino en piezas individuales.
- La asignación desestructurante es una sintaxis especial que nos permite “desempaquetar” arrays u objetos en varias variables, porque a veces es más conveniente.
- La desestructuración también funciona bien con funciones complejas que tienen muchos argumentos, valores por defecto, etcétera.

Desestructuración de Arrays

Un ejemplo de cómo el array es desestructurado en variables:

```
1 // tenemos un array con el nombre y apellido
2 let arr = ["John", "Smith"]
3
4 // asignación desestructurante
5 // fija firstName = arr[0]
6 // y surname = arr[1]
7 let [firstName, surname] = arr;
8
9 alert(firstName); // John
10 alert(surname); // Smith
```

Ahora podemos trabajar con variables en lugar de miembros de array. Se ve genial cuando se combina con split u otro método que devuelva un array:

```
1 let [firstName, surname] = "John Smith".split(' ');
2 alert(firstName); // John
3 alert(surname); // Smith
```

Como puedes ver, la sintaxis es simple. Aunque hay varios detalles peculiares. Veamos más ejemplos para entenderlo mejor.

El resto '...'

En general, si el array es mayor que la lista de la izquierda, los ítems extras son omitidos. Por ejemplo, aquí solo dos ítems son tomados, el resto simplemente es ignorado:

```
1 let [name1, name2] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];
2
3 alert(name1); // Julius
4 alert(name2); // Caesar
5 // ítems posteriores no serán asignados a ningún lugar
```

si queremos también obtener todo lo que sigue, podemos agregarle un parámetro que obtiene “el resto” usando puntos suspensivos “...”:

```
1 let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];
2
3 // `rest` es un array de ítems, comenzando en este caso por el tercero.
4 alert(rest[0]); // Consul
5 alert(rest[1]); // of the Roman Republic
6 alert(rest.length); // 2
```

El valor de rest es un array con los elementos restantes del array original. Podemos usar cualquier otro nombre de variable en lugar de rest, sólo hay que asegurar que tenga tres puntos que lo antecedan y que esté último en la asignación desestructurante.

```
1 let [name1, name2, ...titles] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];
2 // ahora titles = ["Consul", "of the Roman Republic"]
```


Valores predeterminados

Si el array es más corto que la lista de variables a la izquierda, no habrá errores. Los valores ausentes son considerados undefined:

```
1 let [firstName, surname] = [];  
2  
3 alert(firstName); // undefined  
4 alert(surname); // undefined
```

Si queremos un valor “predeterminado” para reemplazar el valor faltante, podemos proporcionarlo utilizando =:

```
1 // valores predeterminados  
2 let [name = "Guest", surname = "Anonymous"] = ["Julius"];  
3  
4 alert(name); // Julius (desde array)  
5 alert(surname); // Anonymous (predeterminado utilizado)
```

Los valores predeterminados pueden ser expresiones más complejas e incluso llamadas a función, que serán evaluadas sólo si el valor no ha sido proporcionado. Por ejemplo, aquí utilizamos la función prompt para dos valores predeterminados.

```
1 // sólo ejecuta la captura para surname  
2 let [name = prompt('nombre?'), surname = prompt('apellido?')] = ["Julius"];  
3  
4 alert(name); // Julius (desde array)  
5 alert(surname); // lo que reciba la captura
```

Observa que el prompt se ejecuta solamente para el valor faltante (surname).

Desestructuración de objetos

La asignación desestructurante también funciona con objetos. La sintaxis básica es:

```
1 let {var1, var2} = {var1:..., var2:...}
```

Debemos tener un símil-objeto en el lado derecho, el que queremos separar en variables. El lado izquierdo contiene un símil-objeto “pattern” para sus propiedades correspondientes. En el caso más simple, es la lista de nombres de variables en {...}. Por ejemplo:

```
1 let options = {  
2   title: "Menu",  
3   width: 100,  
4   height: 200  
5 };  
6  
7 let {title, width, height} = options;  
8  
9 alert(title); // Menu  
10 alert(width); // 100  
11 alert(height); // 200
```

Las propiedades options.title, options.width y options.height son asignadas a las variables correspondientes. No importa el orden sino los nombres. Esto también funciona:

```
1 // cambiado el orden en let {...}  
2 let {height, width, title} = { title: "Menu", height: 200, width: 100 }
```

El patrón de la izquierda puede ser más complejo y especificar el mapeo entre propiedades y variables. Si queremos asignar una propiedad a una variable con otro nombre, por ejemplo que `options.width` vaya en la variable llamada `w`, lo podemos establecer usando dos puntos:

```
1  let options = {
2    title: "Menu",
3    width: 100,
4    height: 200
5  };
6
7  // { propiedadOrigen: variableObjetivo }
8  let {width: w, height: h, title} = options;
9
10 // width -> w
11 // height -> h
12 // title -> title
13
14 alert(title); // Menu
15 alert(w);    // 100
16 alert(h);    // 200
```

Los dos puntos muestran “qué : va dónde”. En el ejemplo de arriba la propiedad `width` va a `w`, `height` va a `h`, y `title` es asignado al mismo nombre. Para propiedades potencialmente faltantes podemos establecer valores predeterminados utilizando `=`, de esta manera:

```
1  let options = {
2    title: "Menu"
3  };
4
5  let {width = 100, height = 200, title} = options;
6
7  alert(title); // Menu
8  alert(width); // 100
9  alert(height); // 200
```

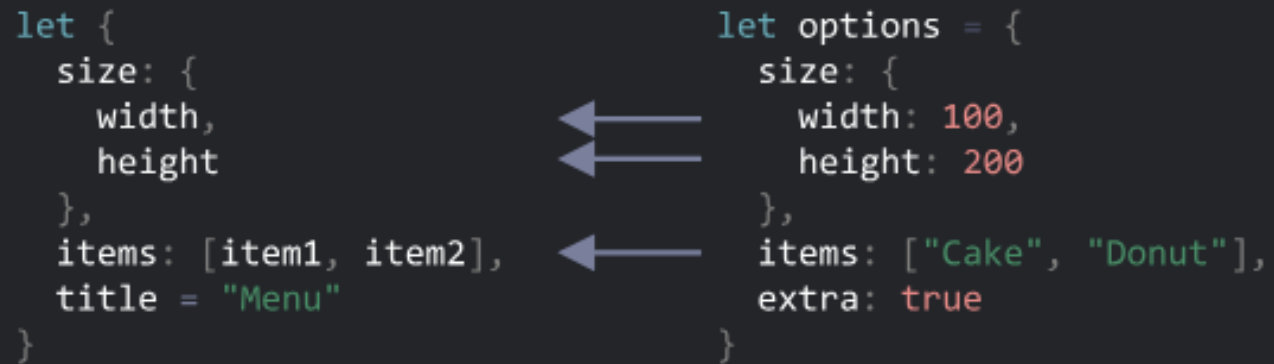
Desestructuración anidada

Si un objeto o array contiene objetos y arrays anidados, podemos utilizar patrones del lado izquierdo más complejos para extraer porciones más profundas.

En el código de abajo options tiene otro objeto en la propiedad size y un array en la propiedad items. El patrón en el lado izquierdo de la asignación tiene la misma estructura para extraer valores de ellos:

```
1  let options = {
2    size: {
3      width: 100,
4      height: 200
5    },
6    items: ["Cake", "Donut"],
7    extra: true
8  };
9
10 // la asignación desestructurante fue dividida en varias líneas para mayor claridad
11 let {
12   size: { // colocar tamaño aquí
13     width,
14     height
15   },
16   items: [item1, item2], // asignar ítems aquí
17   title = "Menu" // no se encuentra en el objeto (se utiliza valor predeterminado)
18 } = options;
19
20 alert(title); // Menu
21 alert(width); // 100
22 alert(height); // 200
23 alert(item1); // Cake
24 alert(item2); // Donut
```

Todas las propiedades del objeto options con excepción de extra que no está en el lado izquierda, son asignadas a las variables correspondientes:



```
let {  
  size: {  
    width,  
    height  
  },  
  items: [item1, item2],  
  title = "Menu"  
}  
  
let options = {  
  size: {  
    width: 100,  
    height: 200  
  },  
  items: ["Cake", "Donut"],  
  extra: true  
}
```

The diagram illustrates the mapping of properties from the `options` object to the destructured object. Blue arrows point from the `width` and `height` properties of the `size` object in `options` to the corresponding variables in the destructured object. Another blue arrow points from the `items` array in `options` to the `items` array in the destructured object. The `extra` property in `options` has no corresponding variable in the destructured object, as it is not destructured.

Por último tenemos `width`, `height`, `item1`, `item2` y `title` desde el valor predeterminado. Tenga en cuenta que no hay variables para `size` e `items`, ya que tomamos su contenido en su lugar.

Argumentos de función inteligentes

Hay momentos en que una función tiene muchos argumentos, la mayoría de los cuales son opcionales. Eso es especialmente cierto para las interfaces de usuario. Imagine una función que crea un menú. Puede tener ancho, altura, título, elementos de lista, etcétera. Aquí hay una forma errónea de escribir tal función:

```
1 function showMenu(title = "Untitled", width = 200, height = 100, items = []) {  
2     // ...  
3 }
```

En la vida real, el problema es cómo recordar el orden de los argumentos. Normalmente los IDEs (Entorno de desarrollo integrado) intentan ayudarnos, especialmente si el código está bien documentado, pero aún así... Otro problema es cómo llamar a una función si queremos que use sus valores predeterminados en la mayoría de los argumentos. ¿Así?

```
1 // undefined para que use los valores predeterminados  
2 showMenu("My Menu", undefined, undefined, ["Item1", "Item2"])
```

Esto no es nada grato. Y se torna ilegible cuando tratamos con muchos argumentos.

¡La desestructuración llega al rescate! Podemos pasar los argumentos como un objeto, y la función inmediatamente los desestructura en variables:

```
1 // pasamos un objeto a la función
2 let options = {
3   title: "My menu",
4   items: ["Item1", "Item2"]
5 };
6
7 // ...y los expande inmediatamente a variables
8 function showMenu({title = "Untitled", width = 200, height = 100, items = []}) {
9   // title, items - desde options
10  // width, height - usan los predeterminados
11  alert( `${title} ${width} ${height}` ); // My Menu 200 100
12  alert( items ); // Item1, Item2
13 }
14
15 showMenu(options);
```

También podemos usar desestructuración más compleja con objetos anidados y mapeo de dos puntos:

```
1  let options = {
2    title: "My menu",
3    items: ["Item1", "Item2"]
4  };
5
6  function showMenu({
7    title = "Untitled",
8    width: w = 100, // width va a w
9    height: h = 200, // height va a h
10   items: [item1, item2] // el primer elemento de items va a item1, el segundo a item2
11 }) {
12   alert( `${title} ${w} ${h}` ); // My Menu 100 200
13   alert( item1 ); // Item1
14   alert( item2 ); // Item2
15 }
16
17 showMenu(options);
```

La sintaxis completa es la misma que para una asignación desestructurante:

```
1  function({
2    incomingProperty: varName = defaultValue // propiedadEntrante: nombreVariable = valor
3    ...
4  })
```

Entonces, para un objeto de parámetros, habrá una variable `varName` para la propiedad `incomingProperty`, con `defaultValue` por defecto. Por favor observe que tal desestructuración supone que `showMenu()` tiene un argumento. Si queremos todos los valores predeterminados, debemos especificar un objeto vacío:

```
1  showMenu({}); // ok, todos los valores son predeterminados
2
3  showMenu(); // esto daría un error
```

Podemos solucionar esto, poniendo `{}` como valor predeterminado para todo el objeto de argumentos:

```
1  function showMenu({ title = "Menu", width = 100, height = 200 } = {}) {
2    alert( `${title} ${width} ${height}` );
3  }
4
5  showMenu(); // Menu 100 200
```

En el código de arriba, todo el objeto de argumentos es `{}` por defecto, por lo tanto siempre hay algo para desestructurar.

RESUMEN

- La asignación desestructurante permite mapear instantáneamente un objeto o array en varias variables.
- La sintaxis completa para objeto:

```
1 let {prop : varName = default, ...rest} = object
```

- Esto significa que la propiedad prop se asigna a la variable varName; pero si no existe tal propiedad, se usa el valor default. Las propiedades de objeto que no fueron mapeadas son copiadas al objeto rest.
- La sintaxis completa para array:

```
1 let [item1 = default, item2, ...resto] = array
```

- El primer item va a item1, el segundo a item2, todos los ítems restantes crean el array resto.
- Es posible extraer información desde arrays/objetos anidados, para esto el lado izquierdo debe tener la misma estructura que el lado derecho.

TAREAS

Asignación desestructurante

Tenemos un objeto:

```
let user = {  
  name: "John",  
  years: 30  
};
```

Escriba la asignación desestructurante que asigne las propiedades:

name en la variable name.

years en la variable age.

isAdmin en la variable isAdmin (false, si no existe tal propiedad)

Este es un ejemplo de los valores después de su asignación:

```
let user = { name: "John", years: 30 };
```

```
// tu código al lado izquierdo:
```

```
// ... = user
```

```
alert( name ); // John
```

```
alert( age ); // 30
```

```
alert( isAdmin ); // false
```

El salario máximo

Hay un objeto salaries:

```
let salaries = {  
  "John": 100,  
  "Pete": 300,  
  "Mary": 250  
};
```

Crear la función topSalary(salaries) que devuelva el nombre de la persona mejor pagada.

Si salaries es vacío, debe devolver null.

Si hay varias personas con la mejor paga, devolver cualquiera de ellas.

PD: Utilice Object.entries y desestructuración para iterar sobre pares de claves/valores.

[Abrir en entorno controlado con pruebas.](#)