10. ARRAYs (II)

Introducción

Los **arrays** en JavaScript son en realidad un tipo especial de objeto.

- funcionan de manera muy similar a los objetos regulares (naturalmente, solo se puede acceder a las **propiedades numéricas** usando la **sintaxis de corchetes** `[]`)
- tienen una propiedad "mágica" (se va autocalculando) llamada `length`: siempre es uno más que el **índice** más alto del array.

Los arrays generalmente se crean con **literales de array** :

```
const animal = ["perro", "gato", "gallina"];
animal.length;
// 3
```

Los arrays de JavaScript al ser objetos

- puedesasignarles cualquier propiedad, incluidos índices de números arbitrarios.
- la única "magia" es que la `length` se actualizará automáticamente cuando establezca un índice en particular.

```
const animal = ["perro", "gato", "gallina"];
animal[100] = "zorro";
console.log(animal.length);  // 101
console.log(animal);         // ['perro', 'gato', 'gallina', vacío × 97, 'zorro']
```

El array que tenemos arriba

- se denomina **array disperso** (**sparse array**) porque hay ranuras deshabitadas en el medio,
- hará que el **motor** lo desoptimice procesándolo como una tabla hash en vez de un array.
- ¡asegúrate de que tus arrays estén densamente poblados!

La **indexación fuera de los límites** no arroja excepciones:

- si consultas un índice de array inexistente, obtendrás un valor de `undefined` a cambio:

```
const animal = ["perro", "gato", "gallina"];
console.log(typeof a[90]);                    // undefined
```

Los arrays pueden tener cualquier tipo de elemento y pueden crecer o reducirse arbitrariamente.

```
const arr = [1, "foo", verdadero];
arr.push({});
// arr = [1, "foo", verdadero, {}]
```

Los arrays se pueden **iterar** con el bucle `for,` como se puede hacer en otros lenguajes similares a C:

```
for (let i = 0; i < animal.length; i++) {
  // Hacer algo con animal[i]
}
```

O, dado que los arrays son **iterables**, puede usar el bucle `for... of` :

```
for (const valorActual of animal) {
  // Hacer algo con valorActual
}
```

Los arrays vienen con una plétora de **métodos de arrays.**

- muchos de ellos iteraran el array
- por ejemplo, `map()` aplica un **callback** (función de que se pasa como parámetro) a cada elemento del array y

devuelve un nuevo array:

```
const bebes = ["perro", "gato", "gallina"].map( (nombre) => `${nombre} bebé` );
// bebes = ['perro bebé', 'gato bebé', 'gallina bebé']
```

En fondo rosa aparece el callback, que es una **función flecha** (***arrow function***).

Colecciones indexadas

Las colecciones indexadas son colecciones de datos que están **ordenados** por un **valor de índice.**

- esto incluye arrays y construcciones similares a arrays, como objetos `Array` y objetos `TypedArray.`
- un array es una lista ordenada de valores a los que hace referencia con un nombre y un índice.
- por ejemplo, considere un array llamada `emp,` que contiene los nombres de los empleados indexados por su número de empleado numérico.
  - entonces `emp[0]` sería el empleado número cero, `emp[1]` el empleado número uno, y así sucesivamente.
- JavaScript no tiene un tipo de datos de array explícito:
  - sin embargo, puede utilizar el objeto/interfaz predefinido (***built-in***) `Array` y sus métodos para trabajar con arrays en sus aplicaciones.
  - el objeto `Array` tiene métodos para manipular arrays de varias maneras, como **unirlas, invertirlas** y **ordenarlas.**
  - tiene una propiedad para determinar la **longitud del array** y otras propiedades para usar con **expresiones regulares.**

Crear un array

Las siguientes declaraciones crean arrays equivalentes:

```
const arr1 = new Array(elemento0, elemento1, /* …,*/ elementoN);
const arr2 = Array(elemento0, elemento1, /* …,*/ elementoN);
const arr3 = [elemento0, elemento1, /* …,*/ elementoN];
```

- `element0, element1, …, elementN` es una lista de valores para los elementos del array.
- cuando se especifican estos valores, el array se **inicializa** con ellos como **elementos del array.**
- `length` del array se establece en el **número de argumentos.**
- la **sintaxis de corchetes** se denomina "**literal array**" o " **inicializador de array** ".
- es más corto que otras formas de creación de arrays, por lo que generalmente se prefiere.

Para crear un array con una longitud distinta de cero, pero sin ningún elemento, se puede usar cualquiera de los siguientes:

```
// Este...
const arr1 = new Array(longitudDelArray);

// ...resulta en la misma array que esta
const arr2 = Array(longitudDelArray);

// Esto tiene exactamente el mismo efecto
const arr3 = [];
array3.length = longitudDelArray;
```

Nota: En el código anterior, `longitudDelArray` debe ser un `Number.`

- de lo contrario, se creará un array con un solo elemento (el valor proporcionado).
- llamar a `arr.length` devolverá `longitudDelArray,` pero el array no contiene ningún elemento.
- un bucle `for...in` no encontrará ninguna propiedad en el array.

Además de a una variable recién definida como se muestra arriba, los arrays también se pueden asignar como una propiedad de un objeto nuevo o existente:

```
const obj = {};
// …
obj.prop = [elemento0, elemento1, /* …,*/ elementoN];

// o
const obj = { prop: [elemento0, elemento1, /* …,*/ elementoN] };
```

Si desea **inicializar un array** con un solo elemento, y el elemento resulta ser un `Number`, debe usar la sintaxis de corchetes.

- cuando se pasa un solo valor numérico al constructor o función `Array()`, se interpreta como un `longitudDeArray`, no como un solo elemento.

```
// Esto crea un array con un solo elemento: el número 42.
const arr = [42];

// Esto crea un array sin elementos y arr.length establecido en 42.
const array = Array(42);

// Esto es equivalente a:
const arr = [];
arr.length = 42;
```

El acto de llamar a `Array(N)`:

- da como resultado un `RangeError`, si `N` es un **número no entero** cuya parte fraccionaria es distinta de cero.
- el siguiente ejemplo ilustra este comportamiento.

```
const arr = Array(9.3); // RangeError : longitud de array no válida
```

Si tu código necesita crear arrays con elementos individuales de un tipo de datos arbitrario:

- es más seguro usar **literales array.**
- alternativamente, primero crea un array vacío antes de agregarle el elemento único.
- también puedes usar el **método estático** `Array.of` para crear arrays con un solo elemento.

```
const arrayConElemento = Array.of(9.3); // arrayConElemento contiene solo un elemento 9.3
```

Como hacer referencia a los elementos del array

Debido a que los **elementos** también son **propiedades**, puede acceder a ellos mediante accesos de **propiedad.**

- supongamos que define el siguiente array:

```
const miArray = ['Viento', 'Lluvia', 'Fuego'];
```

- puedes referirte al primer elemento del array como `miArray[0]`, al segundo elemento del array como `miArray[1]`, etc...
- el **índice** de los elementos comienza con cero.

Nota: también puedes usar **accesores de propiedad** para acceder a otras propiedades del array, como con un objeto.

```
const arr = ['uno', 'dos', 'tres'];
arr[2]              // tres
arr[ 'length' ]     // 3
```

Llenar un array

Puedes **llenar un array** asignando valores a sus elementos.

- por ejemplo:

```
const vacio = [];
vacio[0] = 'Casey Jones';
vacio[1] = 'Phil Lesh';
vacio[2] = 'Agosto Oeste';
```

Nota: si proporcionas un valor no entero al **operador de array** en el código anterior, se creará una propiedad en el objeto array , en lugar de un elemento de array:

```
const arr = [];
arr[ 3.4 ] = 'Naranjas';
console.log(arr.length);                                    // 0
console.log( Object.hasOwn(arr, 3.4 ));                     // verdadero
```

También puede completar un array cuando la crea:

```
const miArray = new Array('Hola', miVariable, 3.14159);
// o
const miArray = ['Mango', 'Apple', 'Orange'];
```

Comprender la longitud (`length`)

A **nivel de implementación**:

- los arrays de JavaScript en realidad almacenan sus elementos como **propiedades de objeto estándar,** utilizando el índice del array como el **nombre de la propiedad.**
- la propiedad de `length` es especial.
  - su valor es siempre un entero positivo mayor que el índice del último elemento si existe.
  - en el siguiente ejemplo, **`'Polviento'`** está indexado en **30,** por lo que **`gatos.length`** devuelve **30 + 1**).
  - recuerde, los índices de array de JavaScript se basan en 0: comienzan en **0,** no en **1** : esto significa que la propiedad de **`length`** será uno más que el índice más alto almacenado en el array:

```
const gatos = [];
gatos[30] = ['Polviento'];
console.log(gatos.longitud); // 31
```

También puede asignar un valor a la propiedad de `length`

- escribir un valor que sea más corto que el número de elementos almacenados **trunca** el array.
- escribir `0` lo vacía por completo:

```
const gatos = ['Polviento', 'Calimoso', 'Nublado'];
console.log(gatos.length);              // 3

gatos.length = 2;
console.log(gatos);                     // [ 'Dusty', 'Misty' ] - Nublado ha sido eliminado

gatos.length = 0;
console.log(gatos);                     // []; El array de gatos está vacío

gatos.length = 3;
console.log(gatos);                     // [ <3 elementos vacíos> ]
```

Iterar sobre arrays

Dado que los elementos del array de JavaScript se guardan como **propiedades de objeto estándar,** no es recomendable iterar a través de los arrays de JavaScript usando **bucles `for...in`,** porque se enumerarán los elementos normales y todas las propiedades enumerables.

Una operación común es **iterar sobre los valores de un array,** procesando cada uno de alguna manera.

- la forma más sencilla de hacer esto es la siguiente:

```
const colores = ['rojo', 'verde', 'azul'];
for (let i = 0; i < colores.length ; i++) {
   console.log(colores[i]);
}
```

Si tu sabes:

- que ninguno de los elementos en tu array se evalúa como `false` en un contexto booleano
- que tu array consta solo de **nodos DOM,** por ejemplo

puedes usar un **modismo** más eficiente:

```
const divs = document.getElementsByTagName('div');
for (let i = 0, div; div = divs[i] ; i++) {
   /* Procesar div de alguna manera */
}
```

- esto:
  - evita la **sobrecarga** de verificar la longitud del array,
  - asegura que la variable `div` se reasigna al elemento actual cada vez que se completa el ciclo para mayor comodidad.

El método `forEach()` proporciona otra forma de iterar sobre un array:

```
const colores = ['rojo', 'verde', 'azul'];
colores.forEach( (color) => console.log(color) );
// rojo
// verde
// azul
```

- la función pasada a `forEach`
  - se ejecuta una vez para cada elemento del array, y el elemento del array se pasa como argumento a la función.
  - los valores no asignados no se iteran en un bucle `forEach.`
- tenga en cuenta que los elementos de un array que se omiten cuando se define el array no se enumeran cuando `forEach` itera,
- pero *se* enumeran cuando `undefined` se ha asignado manualmente al elemento:

```
const arrayDisperso = ['primero', 'segundo', , 'cuatro'];

arrayDisperso.forEach( (elemento) => {console.log(elemento);} );
// Aparece por consola:
// primero
// segundo
// cuatro


if (arrayDisperso[2] === undefined ) {
   console.log('arrayDisperso[2] no está definido');  // sí sale por consola
}

const arrayNoDisperso = ['primero', 'segundo', undefined, 'cuarto'];

arrayNoDisperso.forEach( (elemento) => {console.log(elemento); } );
// Aparece por consola:
```

```
// primero
// segundo
// undefined
// cuatro
```

Observaciones sobre Arrays dispersos

Los arrays pueden contener "ranuras vacías", que no son lo mismo que las ranuras llenas con el valor **undefined.**

- las **ranuras vacías** se pueden crear de una de las siguientes maneras:

```
// Constructor de array:
const a = Array(5);           // [ <5 elementos vacíos> ]

// Comas consecutivas en el array literal:
const b = [1, 2, , , 5];      // [ 1, 2, <2 elementos vacíos>, 5 ]

// Establecer directamente una ranura con un índice mayor que array.length:
const c = [1, 2];
c[ 4 ] = 5;                   // [ 1, 2, <2 elementos vacíos>, 5 ]

// Alargando un array configurando directamente .length:
const d = [1, 2];
d.length = 5 ;                // [ 1, 2, <3 elementos vacíos> ]

// Eliminando un elemento:
const e = [1, 2, 3, 4, 5];
eliminar e[2];                // [ 1, 2, <1 elemento vacío>, 4, 5 ]
```

- en algunas operaciones, las ranuras vacías se comportan como si estuvieran llenas con **undefined.**

```
const arr = [1, 2, , , 5]; // Crea un array dispersa

// Acceso indexado
console.log(arr[2]); // undefined

// for... of
for (const i of arr) {
   console.log(i);
}
// Registros: 1 2 undefined undefined 5

// Spreading
const otroEjemplo = [ ... arr ];      // "otro" es [ 1, 2, undefined, undefined, 5 ]
```

- pero en otros (sobre todo en los **métodos de iteración de arrays**), se omiten los espacios vacíos.

```
const mapeado = arr.map( (i) => i + 1 );        // [ 2, 3, <2 elementos vacíos>, 6 ]
arr.forEach( (i) => console.log(i) );           // 1 2 5
const filtrado = arr.filter( () => true );      // [ 1, 2, 5 ]
const tieneUnFalsy = arr.some( (k) => !k );     // falso


// enumeración de propiedades
const claves = Object.keys(arr);                // [ '0', '1', '4' ]
for (const clave of arr) {
   console.log(clave);
}
// Sale por consola: '0' '1' '4'


// El Spreding de objeto usa la enumeración de propiedades, no el iterador del array
const objectSpread = { ... arr } ;              // { '0': 1, '1': 2, '4': 5 }
```

Arrays multidimensionales

Los arrays:

- se pueden **anidar,** lo que significa que un array puede contener otro/s array/s como elemento/s.
- utilizando esta característica de los arrays de JavaScript, se pueden crear **arrays multidimensionales.**
- el siguiente código crea un array bidimensional.

```
const a = new Array(4);
for (let i = 0; i < 4; i++) {
    a[i] = new Array(3) ;
    for (let j = 0; j < 3; j++) {
       a[i][j] = `[${i}, ${j}]`;
    }
}
```

Este ejemplo crea un array con las siguientes filas:

```
Fila 0: [0, 0] [0, 1] [0, 2]
Fila 1: [1, 0] [1, 1] [1, 2]
Fila 2: [2, 0] [2, 1] [2, 2]
Fila 3: [3, 0] [3, 1] [3, 2]
```

Uso de arrays para almacenar otras propiedades

Los arrays también se pueden usar como objetos, para almacenar información relacionada.

```
const arr = [1, 2, 3];
arr.propiedad = "valor" ;
console.log( arr.propiedad );          // "valor"
```

- por ejemplo, cuando un array es el resultado de una coincidencia entre una expresión regular y una cadena, el array devuelve propiedades y elementos que brindan información sobre la coincidencia.
- un array es el valor de retorno de
  - `RegExp.prototype.exec()`
  - `String.prototype.match()`
  - `String.prototype.split()`

Trabajo con objetos tipo array

Algunos objetos JavaScript, como

- el `NodeList` devuelto por el `document.getElementsByTagName()`
- o el objeto `arguments` disponible dentro del cuerpo de una función,

se ven y se comportan como arrays en la superficie, pero no comparten todos sus métodos.

- el objeto `arguments` proporciona un atributo `length` pero no implementa métodos de array como `forEach()`.

Los métodos de array no se pueden llamar directamente en **objetos similares a arrays.**

```
function imprimirArgumentos() {
  arguments.forEach( ( item) => {      // TypeError : arguments.forEach no existe
      console.log(elemento);
  });
}
```

Pero puedes llamarlos indirectamente usando `Function.prototype.call()`.

```
function imprimirArgumentos() {
  Array.prototype.forEach.call(arguments, (elemento) => {
```

```
    console.log(elemento);
  });
}
```

Los métodos de prototipo de array también se pueden usar en cadenas, ya que brindan acceso secuencial a sus caracteres de manera similar a los arrays:

```
Array.prototype.forEach.call('una cadena', (chr) => {
   console.log(chr)
})
```

Conversión entre cadenas y arrays

A menudo, se te presentarán algunos **datos sin procesar** contenidos en una cadena grande y larga, y es posible que desee separar los elementos útiles en una forma más útil y luego hacer cosas con ellos, como mostrarlos en una tabla de datos.

- para hacer esto, podemos usar el método `split()`.
- en su forma más simple,
  - esto toma un solo **parámetro,** el carácter en el que desea separar la cadena,
  - devuelve las **subcadenas** entre el **separador** como elementos en un array.

Nota: de acuerdo, este es técnicamente un método de cadena, no un método de array, pero es buen sitio para introducirlo ya que funciona bien aquí.

Juguemos con esto, para ver cómo funciona. Primero, crea una cadena en tu consola:

```
const datos = 'Manchester,Londres,Liverpool,Birmingham,Leeds,Carlisle';
```

Ahora vamos a dividirlo en cada coma:

```
const ciudades = datos.split(',');
ciudades;  // ['Manchester', 'Londres', 'Liverpool', 'Birmingham', 'Leeds', 'Carlisle']
```

Finalmente, intenta encontrar la `length` de su nuevo array y recuperar algunos elementos de él:

```
ciudades.length;
ciudades[0];                          // el primer elemento del array
ciudades[1];                          // el segundo elemento del array
ciudades[ciudades.length - 1];        // el último elemento del array
```

También puedes hacer los contrario usando el método `join()`. Prueba lo siguiente:

```
const ciudadesSeparadasPorComas = ciudades.join(',');
ciudadesSeparadasPorComas
```

Otra forma de convertir un array en una cadena es usar el método `toString()`.

- es posiblemente más simple que `join()` ya que no toma un parámetro, pero es más limitante.
- con `join()` puedes especificar diferentes separadores, mientras que `toString()` siempre usa una coma .

```
const nombresPerro = ['Rocket','Flash','Bella','Slugger'];
nombresPerro.toString(); // Rocket,Flash,Bella,Slugger
```

Revisión de métodos de array

El objeto `Array` tiene los siguientes métodos:

El método `concat()` une dos o mos arrays y devuelve un nuevo array.

```
let miArray = ['1', '2', '3'];
miArray = miArray.concat('a', 'b', 'c');
// miArray ahora es ["1", "2", "3", "a", "b", "c"]
```

El método `join()` une todos los elementos de un array en una cadena.

```
const miArray = ['Viento', 'Lluvia', 'Fuego'];
const lista = miArray.join(' - '); // la lista es "Viento - Lluvia - Fuego"
```

El método `push()` agrega uno o más elementos al final de un array y devuelve la longitud del array resultante.

```
const miArray = ['1', '2'];
miArray.push('3'); // miArray ahora es ["1", "2", "3"]
```

El método `pop()` elimina el último elemento de un array y devuelve ese elemento.

```
const miArray = ['1', '2', '3'];
const ultimo = miArray.pop();
// miArray ahora es ["1", "2"], ultimo = "3"
```

El método `shift()` elimina el primer elemento de un array y devuelve ese elemento.

```
const miArray = ['1', '2', '3'];
const primero = miArray.shift();
// miArray ahora es ["2", "3"], primero es "1"
```

El método `unshift()` agrega uno o más elementos al frente de un array y devuelve la nueva longitud del array.

```
const miArray = ['1', '2', '3'];
miArray.unshift('4', '5');
// miArray se convierte en ["4", "5", "1", "2", "3"]
```

El método `slice()` extrae una sección de un array y devuelve un nuevo array.

```
let miArray = ['a', 'b', 'c', 'd', 'e'];
miArray = miArray.slice(1, 4); // comienza en el índice 1 y extrae todos los elementos
                               // hasta el índice 3, devolviendo [ "b", "c", "d"]
```

El método `at()`

- devuelve el elemento en el índice especificado en el array, o `undefined` si el índice está fuera de rango.
- se usa especialmente con **índices negativos** que acceden a elementos desde el final del array.

```
const miArray = ['a', 'b', 'c', 'd', 'e'];
miArray.at(-2);              // "d", el penúltimo elemento de miArray
```

El método `splice()` elimina elementos de un array y (opcionalmente) los reemplaza. Devuelve los elementos que se eliminaron del array.

```
const miArray = ['1', '2', '3', '4', '5'];
miArray.splice(1, 3, 'a', 'b', 'c', 'd');
// miArray ahora es ["1", "a", "b", "c", "d", "5"]
// Este código comenzó en el índice 1 (o donde estaba el "2"),
```

```
// eliminó 3 elementos allí y luego insertó todos los elementos consecutivos en su lugar.
```

El método `reverse()` transpone los elementos de un array, en su lugar: el primer elemento del array se convierte en el último y el último se convierte en el primero. Devuelve una referencia a el array.

```
const miArray = ['1', '2', '3'];
miArray.reverse();
// transpone El array para que miArray = ["3", "2", "1"]
```

El método `flat()` devuelve un nuevo array con todos los elementos del sub-array concatenados recursivamente hasta la profundidad especificada.

```
let miArray = [1, 2, [3, 4]];
miArray = miArray.flat();
// miArray ahora es [1, 2, 3, 4], ya que el subarray [3, 4] está aplanado
```

El método `sort()` ordena los elementos de un array en su lugar y devuelve una referencia al array.

```
const miArray = ['Viento', 'Lluvia', 'Fuego'];
miArray.sort();
// ordena El array para que miArray = ["Fuego", "Lluvia", "Viento"]
```

- `sort()` también puede tomar una **función de devolución de llamada** (***callback***) para determinar cómo se comparan los elementos del array.
- la función de devolución de llamada se llama con dos argumentos, que son dos valores del array.
- la función compara estos dos valores y devuelve un número positivo, un número negativo o cero, lo que indica el orden de los dos valores.
- por ejemplo, lo siguiente ordenará el array por la última letra de una cadena:

```
const funcionDeOrdenar = (a, b) => {
if (a[a.length - 1] < b[b.length - 1]) {
   return -1;                          // Número negativo => a < b, a viene antes que b
} else if (a[a.length - 1] > b[b.length - 1]) {
   return 1;                           // Número positivo => a > b, a viene después de b
}
   return 0;                           // Cero => a = b, a y b mantienen su orden original
}

miArray.sort(funcionDeOrdenar);
// ordena El array para que miArray = ["Viento","Fuego","Lluvia"]
```

- si **a** es menor que **b** por el sistema de clasificación, devuelve **-1** (o cualquier número negativo)
- si **a** es mayor que **b** por el sistema de clasificación, devuelve **1** (o cualquier número positivo)
- si **a** y **b** se consideran equivalentes, devuelve **0.**

El método `indexOf()` busca en el array *`elementoBuscado`* y devuelve el índice de la primera coincidencia.

```
const a = ['a', 'b', 'a', 'b', 'a'];
console.log(a.indexOf('b'));                    // 1

// Ahora inténtalo de nuevo, comenzando desde después de la última coincidencia
console.log(a.indexOf('b', 2));                 // 3
console.log(a.indexOf('z'));                    // -1, porque no se encontró 'z'
```

El método `lastIndexOf()` funciona como `indexOf`, pero comienza al final y busca hacia atrás.

```
const a = ['a', 'b', 'c', 'd', 'a', 'b'];
console.log(a.lastIndexOf('b')); // 5
```

```
// Ahora inténtalo de nuevo, comenzando desde antes de la última coincidencia
console.log(a.lastIndexOf('b', 4)); // 1
console.log(a.lastIndexOf('z')); // -1
```

El método `forEach()` ejecuta una **función de devolución de llamada** (***callback***) para cada elemento del array y devuelve `undefined`.

```
const a = ['a', 'b', 'c'];
a.forEach( (elemento) => { console.log(elemento); } );
// Registros:
// a
// b
// C
```

- el método `forEach` (y otros a continuación) que reciben una **función de devolución de llamada** se conocen como ***métodos iterativos,*** porque iteran sobre todo el array de alguna manera.
- cada uno toma un segundo argumento opcional llamado `thisArg`.
  - si se proporciona, `thisArg` se convierte en el valor de la **palabra clave** `this` dentro del cuerpo de la **función de devolución de llamada.**
  - si no se proporciona, como en otros casos en los que se invoca una función fuera del **contexto de un objeto explícito,** se referirá al **objeto global** (`window`, `globalThis`, etc.) cuando la función no es estricta, o `undefined` cuando la función es estricta.

Nota: El método `sort()` presentado anteriormente no es un método iterativo, porque su función de devolución de llamada solo se usa para comparar y no se puede llamar en ningún orden en particular según el orden de los elementos. `sort()` tampoco acepta el parámetro `thisArg`.

El método `map()` devuelve un nuevo array del valor devuelto al ejecutar la función de devolución de llamada en cada elemento del array.

```
const a1 = ['a', 'b', 'c'];
const a2 = a1.map( (elemento) => elemento.toUpperCase() );
console.log(a2); // ['A B C']
```

El método `flatMap()` ejecuta `map()` seguido de `flat()` de profundidad 1.

```
const a1 = ['a', 'b', 'c'];
const a2 = a1.flatMap( (elemento) => [elemento.toUpperCase(), elemento.toLowerCase()]);
console.log(a2); // ['A', 'a', 'B', 'b', 'C', 'c']
```

El método `filter()` devuelve un nuevo array que contiene los elementos para los que la devolución de llamada devolvió `true`.

```
const a1 = ['a', 10, 'b', 20, 'c', 30];
const a2 = a1.filter((articulo) => typeof articulo === 'number');
console.log(a2); // [10, 20, 30]
```

El método `find()` devuelve el primer elemento para el que la función de devolución de llamada devolvió `true`.

```
const a1 = ['a', 10, 'b', 20, 'c', 30];
const i = a1.find((elemento) => typeof elemento === 'number');
```

```
console.log(i); // 10
```

El método `findLast()` devuelve el último elemento para el que la función de devolución de llamada devolvió `true`.

```
const a1 = ['a', 10, 'b', 20, 'c', 30];
const i = a1.findLast((elemento) => typeof elemento === 'number');
console.log(i); // 30
```

El método `findIndex()` devuelve el índice del primer elemento para el que la función de devolución de llamada devolvió `true`.

```
const a1 = ['a', 10, 'b', 20, 'c', 30];
const i = a1.findIndex((elemento) => typeof elemento === 'number');
console.log(i); // 1
```

El método `findLastIndex()` devuelve el índice del último elemento para el que la función de devolución de llamada devolvió `true`.

```
const a1 = ['a', 10, 'b', 20, 'c', 30];
const i = a1.findLastIndex((elemento) => typeof elemento === 'number');
console.log(i); // 5
```

El método `every()` devuelve verdadero si la función de devolución de llamada devolvió `true` para cada elemento del array.

```
function esNumero(valor) {
   return typeof valor === 'number';
}

const a1 = [1, 2, 3];
console.log(a1.every (esNumero)); // verdadero

const a2 = [1, '2', 3];
console.log(a2.every (esNumero)); // falso
```

El método `some()` devuelve `true` si la función de devolución de llamada devolvió `true` para al menos un elemento del array.

```
function esNumero(valor) {
   return typeof valor === 'number';
}

const a1 = [1, 2, 3];
console.log(a1.some(esNumero)); // verdadero
const a2 = [1, '2', 3];
console.log(a2.some(esNumero)); // verdadero
const a3 = ['1', '2', '3'];
console.log(a3.some(esNumero)); // falso
```

El método `reduce()`

- aplica la función de devolución de llamada **`(acumulador, valorActual, indiceActual, array)`** para cada valor en el array con el fin de reducir la lista de elementos a un solo valor.
- la función **`reduce`** devuelve el valor final devuelto por la función de devolución de llamada.
  - si se especifica **`valorInicial`**, se llama a la función de devolución de llamada con **`valorInicial`** como primer valor de parámetro y el valor del primer elemento del array como segundo valor de parámetro.
  - si **`valorInicial`** *no* se especifica, entonces los dos primeros valores de parámetro del callback serán el primer y segundo elemento del array.
  - en *cada* llamada posterior, el valor del primer parámetro será cualquier devolución de llamada devuelta en la llamada anterior, y el valor del segundo parámetro será el siguiente valor en el array.
- Si la función de devolución de llamada/callback necesita acceso al índice del elemento que se está procesando, o acceso a todo elarray, están disponibles como parámetros opcionales.

```
const a = [10, 20, 30];
const total = a.reduce((acumulador, valoractual) => acumulador + valoractual, 0);
console.log(total); // 60
```

El método **`reduceRight()`**

- funciona como **`reduce()`**, pero comienza con el último elemento.
- **`reduce`** y **`reduceRight`** son los menos obvios de los métodos de array iterativos.
- deben usarse para algoritmos que combinan dos valores recursivamente para reducir una secuencia a un solo valor.

Typed Arrays (Arrays Tipados)

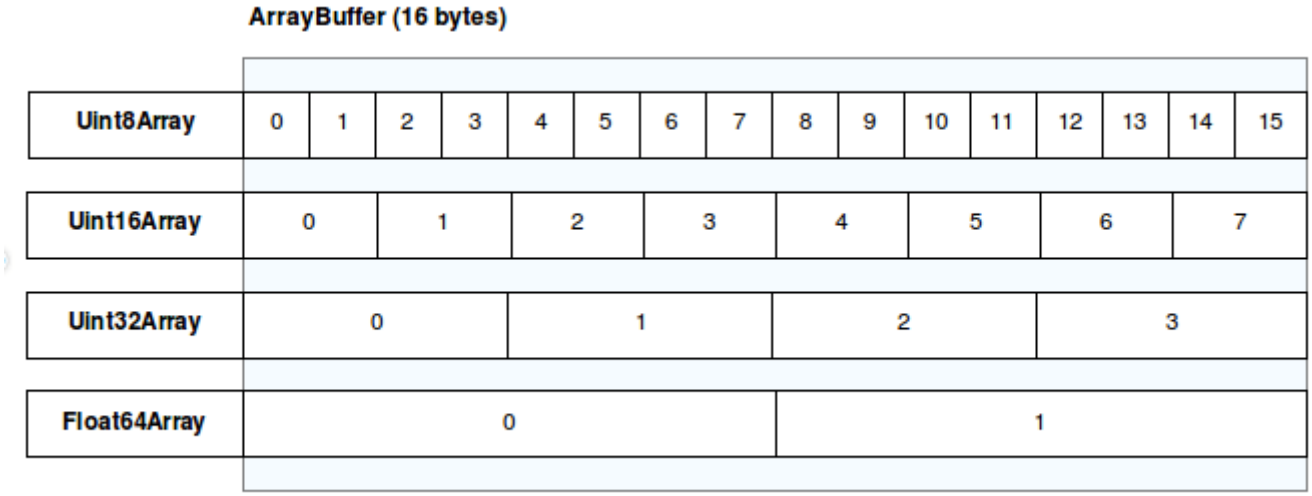
Los **arrays tipados:**

- son objetos similares a arrays y proporcionan un mecanismo para acceder a **datos binarios sin procesar.**
- como ya sabes, los objetos **`Array`** crecen y se reducen dinámicamente y pueden tener cualquier valor de JavaScript.
- los **motores** de JavaScript:
  - llevan a cabo **optimizaciones** para que estos arrays sean rápidos.
  - sin embargo, a medida que las aplicaciones web se vuelven más y más poderosas, agregando funciones como la manipulación de audio y video, el acceso a datos sin procesar mediante WebSockets, etc., ha quedado claro que hay ocasiones en las que sería útil que el código JavaScript pudiera para manipular rápida y fácilmente datos binarios sin procesar en arrays tipeadas.
- Los objetos **typed array**
  - comparten muchos de los mismos métodos que los arrays con semántica similar.
  - sin embargo, conceptualmente no son arrays y no tienen todos los métodos de array.

Búferes y vistas: arquitectura de array tipada

Para lograr la máxima flexibilidad y eficiencia, los **arrays tipados** en JavaScript dividen la implementación en ***búferes*** y ***vistas.***

- un **búfer** (implementado por el objeto **`ArrayBuffer`**)
  - es un objeto que representa un fragmento de datos;
  - no tiene un formato digno de mención y no ofrece ningún mecanismo para acceder a sus contenidos.
  - para acceder a la memoria contenida en un búfer, debe usar una **vista.**
- una **vista** proporciona un *contexto*
  - es decir, un **tipo de datos,** un **desplazamiento inicial** y una **cantidad de elementos**
  - que convierte los datos en bruto en un auténtico array tipado.



array de búfer

El `ArrayBuffer` es un **tipo de datos que se utiliza para representar un búfer de datos binario** genérico de longitud fija.

- no puedes manipular directamente el contenido de un `ArrayBuffer` ;
- en su lugar, tienes que crear una **vista de array con tipo** o un `DataView` que representa el búfer en un formato específico y lo usa para leer y escribir el contenido del búfer.

Vistas de array tipadas

Las **vistas de arrays tipados** tienen nombres autodescriptivos y proporcionan vistas para todos los tipos numéricos habituales.

- como `Int8`, `Uint32`, `Float64` y así sucesivamente.
- hay una vista de array con tipo especial, `Uint8ClampedArray`, que fija los valores entre `0` y `255` . Esto es útil para el **procesamiento de datos de Canvas HTML,** por ejemplo.

Colecciones con clave

Presentamos ahora colecciones de datos que están **indexados** por una **clave**: los objetos `Map` y `Set` contienen elementos que son iterables en el **orden de inserción**.

`Map`s

Objeto `Map`

Un objeto `Map` es una correspondencia o "mapeo" **clave**/**valor** simple y puede iterar sus elementos en el **orden de inserción**.

- el código siguiente muestra algunas operaciones básicas con un `Map`
- puedes usar un bucle `for...of` para devolver un array [clave, valor] para cada iteración.

```
const refranes = new Map();
refranes.set('perro', 'guau');
refranes.set('gato', 'miau');
refranes.set('elefante', 'piuu');
refranes.size;                          // 3
refranes.get('perro');                  // guau
refranes.get('zorro');                  // undefined
refranes.has('pájaro');                 // false
refranes.delete('perro');
refranes.has('perro');                  // false

for (const [clave, valor] of refranes) {
   console.log(`Un ${clave} hace ${valor}`);
}
// "Un gato hace miau"
// "Un elefante hace piuu"

refranes.clear();
refranes.size; // 0
```

Comparación entre `Object` y `Map`

Tradicionalmente, los objetos se han utilizado para asociar cadenas a valores.

- los objetos te permiten:
  - asociar claves a valores,
  - recuperar esos valores,
  - eliminar claves
  - detectar si algo está almacenado en una clave.
- los objetos de `Map`, sin embargo, tienen algunas ventajas más que los hacen mejores para establecer asociaciones o correspondencias entre claves y valores:
  - las claves de un `Object` son `String` o `Symbol`, mientras que un `Map` pueden ser de cualquier tipo.
  - puedes obtener el tamaño `size` de un `Map` fácilmente, mientras que para un `Object` debes buscar tu su tamaño con tu propio código (y recuerda que un `Object` puede tener propiedades própias, propiedades heredadas, enumerables, ...)
  - la iteración de `Map`s es en el orden de inserción de los elementos.
  - un objeto tiene un prototipo, por lo que hay claves predeterminadas en él que tu no creaste (esto se puede cambiar usando `miAsociacion = Object.create(null) )` .

Estos tres consejos pueden ayudarte a decidir si usar un `Map` o un `Object` :

- elige `Map` cuando las claves sean desconocidas hasta el tiempo de ejecución y cuando todas las claves sean del mismo tipo y todos los valores sean del mismo tipo.
- use `Map` si es necesario almacenar valores primitivos como claves porque el objeto trata cada clave como una cadena, ya sea un valor numérico, un valor booleano o cualquier otro valor primitivo.
- use `Object` cuando haya una lógica (código) que opere en elementos individuales.

Objeto `WeakMap`

Un `WeakMap` es una colección de pares clave/valor cuyas claves deben ser objetos, con valores de cualquier tipo de JavaScript, y que no crea **referencias sólidas** como sus claves:

- una clave no contiene una referencia sólida si la presencia de esa referencia al objeto no evita que el objeto sea **recolectado**

como **basura**.
- una vez que un objeto (referenciado por una o más claves de un `WeakMap`) ha sido recolectado o marcado para recolecció por el **recolector de basura** (***garbage collector***) del **runtime**, sus valores correspondientes en cualquier `WeakMap` también se convierten en candidatos para la recolección de elementos no utilizados, siempre que no haya una referencia sólida en otra parte del script/programa.

La API de `WeakMap` es esencialmente la misma que la API de `Map` .

- sin embargo, no permite comprobar la vitalidad de sus claves, por lo que no permite la **enumeración**.
- por lo tanto, no existe un método para obtener una lista de claves en un `WeakMap` .
- si lo hubiera, la lista tendría que depender del estado de recolección de basura, introduciendo el **no determinismo**.

Un ejemplo de uso de los objetos `WeakMap` es almacenar **datos privados** para un objeto u **ocultar detalles de implementación**.

- el siguiente ejemplo procede de la publicación de blog de Nick Fitzgerald "Ocultar detalles de implementación con ECMAScript 6 WeakMaps" .
- los datos y métodos privados pertenecen dentro del objeto y se almacenan en la variable **`privados`** que es un `WeakMap` .
- todo lo expuesto en la instancia y en el prototipo es público; todo lo demás es inaccesible desde el mundo exterior porque los datos privados no se exportan desde el módulo.

```
const privados = new WeakMap();

function Publico() {
  const yo = {
    // meter aqui los datos privados
  };
  privados.set(this, uo);
}

Publico.prototype.metodo = function () {
  const yo = privados.get(this);
  // código que modifica los datos privados en `yo`
  // …
};

module.exports = Publico;
```

`Set`s (conjuntos)

Objeto `Set`

Los objetos `Set` son colecciones de valores.

- puedes iterar sus elementos en orden de inserción .
- un valor en un `Set` solo puede aparecer una vez ; es único en la colección del `Set` .

El siguiente código muestra algunas operaciones básicas con un Set:

```
const miConjunto = new Set();
miConjunto.add(1);
miConjunto.add('algún texto');
miConjunto.add('foo');

miConjunto.has(1);          // true
miConjunto.delete('foo');
miConjunto.size;            // 2

for (const elemento of miConjunto) {
  console.log(elemento);
}
// 1
// "algún texto"
```

Como pasar de `Array` a `Set` y viceversa

Puedes:

- crear un **Array** a partir de un **Set** utilizando `Array.from` o la **sintaxis spread** .

- además, el constructor `Set` acepta un `Array` y lo convierte en un `Set`.

Nota: los `Set` almacenan *valores únicos, por lo que* cualquier elemento duplicado del Array se elimina en nla conversión.

```
Array.from(miConjunto);
[...miConjunto2];

miConjunto2 = new Set([1, 2, 3, 4]);
```

Array y Set comparados

Tradicionalmente, un conjunto de elementos se almacenaba en un array en JavaScript en muchas situaciones.

- el objeto `Set` , sin embargo, tiene algunas ventajas:
  - eliminar elementos de `Array` por valor (`arr.splice(arr.indexOf(val), 1)`) es muy lento.
    - un Set te permite eliminar elementos por su valor.
    - si los almacenases en un array, tendrías que hacer `splice` basándote en el índice del elemento.
  - el valor `NaN` no se puede encontrar con el método `indexOf` en un array.
  - los objetos `Set` almacenan valores únicos: no es necesario que realices un seguimiento manual de los duplicados.

Objeto `WeakSet`

Los `WeakSet` son colecciones de objetos:

- un objeto en un `WeakSet` solo puede ocurrir una vez.
- es único en la colección de `WeakSet` y los objetos no son **enumerables**.

Las principales diferencias con `Set` son:

- los `WeakSet`s son colecciones de objetos *solamente*, y no de valores arbitrarios de cualquier tipo.
- el `WeakSet` es *débil* : guarda **referencias débiles** a los objetos de la colección.
  - si no hay otra referencia a un objeto, fuera del `WeakSet`, almacenado en el `WeakSet`, el recolector de basura del runtime lo puede recolectar.
  - esto también significa que no se puede disponer de una lista de objetos actuales almacenados en la colección.
- los `WeakSet`s no son enumerables.

Los casos de uso de los objetos WeakSet son limitados. No perderán memoria, por lo que puede ser seguro usar elementos DOM como clave y marcarlos con fines de seguimiento, por ejemplo.

Igualdad de clave y valor de Map y Set

Tanto

- la igualdad de clave de los objetos `Map`
- la igualdad de valores de los objetos `Set` se basan en el algoritmo SameValueZero:
  - la igualdad funciona como el operador de comparación de identidad `===` .
  - -0 y +0 se consideran iguales.
  - NaN se considera igual a sí mismo (al contrario de `===`).

Igualdad Same-Value-Zero

Similar a la igualdad del mismo valor, pero +0 y -0 se consideran iguales.

- la igualdad del mismo valor cero no se expone como una API de JavaScript, pero se puede implementar con código personalizado
- solo difiere de la **igualdad estricta** al tratar a `NaN` como equivalente,
- solo se diferencia de la **igualdad del mismo valor** al tratar -0 como equivalente a 0 .
- esto hace que normalmente tenga el comportamiento más sensato durante una búsqueda, especialmente cuando se trabaja con `NaN` .
- lo utilizan `Array.prototype.includes()` , `TypedArray.prototype.includes()` , así como los métodos `Map` y `Set` para comparar la igualdad de claves.

```
function sameValueZero(x, y) {
  if (typeof x === "number" && typeof y === "number") {
    // x e y son iguales(pueden ser -0 y 0) o ambos son NaN
    return x === y || (x !== x && y !== y);
  }
  return x === y;
}
```