DWC

(Desarrollo Web en entorno cliente)

Índice

1.- Arrays

Los arrays son objetos tipo-lista, cuyo prototipo tiene métodos para efectuar operaciones de recorrido y de mutación. **Ni la longitud de una matriz de JavaScript ni los tipos de sus elementos son fijos**.

2.- Declaración y conceptos básicos

Vamos a ver como se declaran los arrays y una serie de características que poseen en cuanto a su funcionamiento y su recorrido

2.1.- Declaración

Hay dos sintaxis para crear un array vacío:

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

Mayoritariamente se usa la segunda sintaxis. Podemos suministrar elementos iniciales entre paréntesis:

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

Usando new Array() también podemos pasar valores para la creación del array:

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

Raramente se usa, porque los corchetes `[]` son más cortos. También hay una característica especial con esta forma de crear arrays.

Si `new Array` se llama con un solo argumento que es un número, crea una array *sin elementos, pero con la longitud dada*.

Veamos cómo uno puede complicarse la vida innecesariamente:

```
let arr = new Array(2); // will it create an array of [2] ?

alert( arr[0] ); // undefined! no elements.

alert( arr.length ); // length 2
```

En el código anterior, `new Array(number)` tiene todos los elementos `undefined`.

Para evadir tales sorpresas, generalmente usamos corchetes, a menos que realmente sepamos lo que estamos haciendo.

Los elementos del array están numerados, comenzando con cero.

Podemos obtener un elemento por su número entre corchetes:

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

alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum
```

Podemos reemplazar un elemento:

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

O agregar uno nuevo:

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

El total de los elementos en el array es la propiedad `length`:

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

alert( fruits.length ); // 3
```

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

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

alert( fruits ); // Apple,Orange,Plum
```

Un array puede almacenar elementos de cualquier tipo.

Por ejemplo:

```
// mix of values
let arr = [ 'Apple', { name: 'John' }, true, function() { alert('hello'); } ];

// get the object at index 1 and then show its name
alert( arr[1].name ); // John
```

```
// get the function at index 3 and run it
arr[3](); // hello
```

2.2.- Funcionamiento como Colas o Listas

Una **cola** es uno de los usos más comunes de un array. En informática, esto significa una colección ordenada de elementos que admite dos operaciones:

- `Push` agrega un elemento al final.
- `Shift` obtener un elemento desde el principio, avanzando la cola, para que el segundo elemento se convierta en el primero.

Los arrays admiten ambas operaciones.
En la práctica lo necesitamos con mucha frecuencia. Por ejemplo, una cola de mensajes que deben mostrarse en la pantalla.
Hay otro caso de uso para las matrices: la estructura de datos denominada **pila**
Es compatible con dos operaciones:

- `Push` agrega un elemento al final.
- `Pop` toma un elemento del final.

Por lo tanto, los elementos nuevos se agregan o se eliminan siempre del "final".
Una pila generalmente se ilustra como un paquete de cartas: se agregan nuevas cartas en la parte superior o se toman de la parte superior:
Para las pilas, se coge primero el último elemento enviado, también llamado principio LIFO (último en entrar, primero en salir). Para las colas, tenemos FIFO (Primero en entrar, primero en salir).
Los arrays en JavaScript pueden funcionar como una cola y como una pila. Le permiten agregar / eliminar elementos tanto desde el principio o al final.

Métodos que funcionan con el final del array (Cola):

`pop`

Extrae el último elemento del array y lo devuelve:

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

```
alert( fruits.pop() ); // remove "Pear" and alert it

alert( fruits ); // Apple, Orange
```

push

Agrega el elemento al final del array:

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

fruits.push("Pear");

alert( fruits ); // Apple, Orange, Pear
```

Métodos que funcionan con el comienzo del array (Pila):

shift

Extrae el primer elemento del array y lo devuelve:

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

alert( fruits.shift() ); // remove Apple and alert it

alert( fruits ); // Orange, Pear
```

unshift

Agregue el elemento al comienzo del array:

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

fruits.unshift('Apple');

alert( fruits ); // Apple, Orange, Pear
```

Los métodos `push` y `unshift` pueden agregar múltiples elementos a la vez:

```
let fruits = ["Apple"];

fruits.push("Orange", "Peach");
fruits.unshift("Pineapple", "Lemon");

// ["Pineapple", "Lemon", "Apple", "Orange", "Peach"]
```

```
alert( fruits );
```

2.3.-Bucles

Una de las formas más antiguas de iterar sobre los elementos de un array es el bucle `for` sobre los índices:

```
let arr = ["Apple", "Orange", "Pear"];

for (let i = 0; i < arr.length; i++) {
  alert( arr[i] );
}
```

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

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

// iterates over array elements
for (let fruit of fruits) {
  alert( fruit );
}
```

El `for..of` no da acceso al número del elemento actual, solo su valor, pero en la mayoría de los casos es suficiente. Y es más corto.

2.4.- La propiedad length del array

La propiedad `length` se actualiza automáticamente cuando modificamos el array. Para ser precisos, en realidad no es el recuento de valores en el array, sino el mayor índice numérico más uno.

Por ejemplo, un solo elemento con un índice grande da una gran longitud:

```
let fruits = [];
fruits[123] = "Apple";

alert( fruits.length ); // 124
```

Tenga en cuenta que generalmente no usamos arrays como este.

Otra cosa interesante sobre la propiedad `length` es que se puede escribir.

Si lo aumentamos manualmente, no pasa nada interesante. Pero si lo disminuimos, la matriz se trunca. El proceso es irreversible, aquí está el ejemplo:

```
let arr = [1, 2, 3, 4, 5];

arr.length = 2; // truncate to 2 elements
alert( arr ); // [1, 2]

arr.length = 5; // return length back
alert( arr[3] ); // undefined: the values do not return
```

Por lo tanto, la forma más sencilla de vaciar un array es: `arr.length = 0;`.

2.5.- Arrays multidimensionales

Las matrices pueden tener elementos que también son arrays. Podemos usarlo para matrices multidimensionales, por ejemplo para almacenar matrices:

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

alert( matrix[1][1] ); // 5, the central element
```

Los arrays tienen su propia implementación del `toString` método que devuelve una lista de elementos separados por comas.

Por ejemplo:

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

alert( arr ); // 1,2,3
alert( String(arr) === '1,2,3' ); // true
```

2.6.- Resumen conceptos básicos de arrays

Un array es un tipo especial de objeto, adecuado para almacenar y administrar elementos de datos ordenados.

- La declaracion:

```
// square brackets (usual)
let arr = [item1, item2...];


// new Array (exceptionally rare)
let arr = new Array(item1, item2...);
```

  La llamada a `new Array(number)` crea una matriz con la longitud dada, pero sin elementos.

- La propiedad `length` es la longitud del array o, para ser precisos, su último índice numérico más uno. Se ajusta automáticamente mediante métodos de matriz.
- Si acortamos `length` manualmente, el array se trunca.
- Podemos usar un array como Cola o Pila con las siguientes operaciones:
  - `push(...items)` agrega `items` al final.
  - `pop()` elimina el elemento del final y lo devuelve.
  - `shift()` elimina el elemento desde el principio y lo devuelve.
  - `unshift(...items)` añade `ítems` al principio
- Para recorrer los elementos de la matriz:
  - `for (let i=0; i<arr.length; i++)` funciona más rápido, compatible con navegadores antiguos.
  - `for (let item of arr)` la sintaxis moderna solo para elementos,

3.- Métodos de los arrays

Las matrices proporcionan muchos métodos. Para facilitar las cosas, los veremos agrupados por tipos de operacion.

3.1.- Agregar, eliminar y extraer elementos

Vamos a los métodos para poder realizar las operaciones básicas de inserción y borrado. Ademas de la extracción de elementos de un arrya

3.1.1.- Agregar y eliminar al inicio y al fin

Ya conocemos métodos que agregan y eliminan elementos desde el principio o el final:

- `arr.push(...items)` - agrega elementos al final,
- `arr.pop()` - extrae un artículo del final,
- `arr.shift()` - extrae un elemento desde el principio,
- `arr.unshift(...items)` - Agrega elementos al principio.

3.1.2.- splice

¿Cómo eliminar un elemento de un array?

Las matrices son objetos, por lo que podemos intentar usar `delete`:

```
let arr = ["I", "go", "home"];

delete arr[1]; // remove "go"

alert( arr[1] ); // undefined

// now arr = ["I",  , "home"];
alert( arr.length ); // 3
```

El elemento fue eliminado, pero la matriz todavía tiene 3 elementos, podemos ver eso `arr.length == 3`.

Eso es natural, porque `delete obj.key` elimina un valor por el `key`. Es todo lo que hace. Bien para los objetos. Pero para los arrays, generalmente queremos que

el resto de elementos se desplace y ocupe el lugar liberado. Esperamos tener un array más corto ahora.

Por lo tanto, se deben utilizar métodos especiales.

El método arr.splice (inicio) es una navaja suiza para arrays. Puede hacer todo: insertar, eliminar y reemplazar elementos.

La sintaxis es:

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

Comienza desde la posición `index`: elimina `deleteCount` elementos y luego inserta `elem1, ..., elemN` en su lugar.

Splice devuelve los elementos borrados y deja el array original actualizado con las operaciones realizadas.

Este método es fácil de entender con ejemplos.

Comencemos con la eliminación:

```
let arr = ["I", "study", "JavaScript"];

arr.splice(1, 1); // from index 1 remove 1 element

alert( arr ); // ["I", "JavaScript"]
```

A partir del índice `1`, eliminó un elemento.

En el siguiente ejemplo eliminamos 3 elementos y los reemplazamos con los otros dos:

```
let arr = ["I", "study", "JavaScript", "right", "now"];

// remove 3 first elements and replace them with another
arr.splice(0, 3, "Let's", "dance");

alert( arr ) // now ["Let's", "dance", "right", "now"]
```

Aquí podemos ver que `splice` devuelve el array de elementos eliminados:

```
let arr = ["I", "study", "JavaScript", "right", "now"];

// remove 2 first elements
```

```
let removed = arr.splice(0, 2);

alert( removed ); // "I", "study" <-- array of removed elements
```

El método `splice` también puede insertar los elementos sin ninguna eliminación. Para eso tenemos que configurar `deleteCount` a `0`:

```
let arr = ["I", "study", "JavaScript"];

// from index 2
// delete 0
// then insert "complex" and "language"
arr.splice(2, 0, "complex", "language");

alert( arr ); // "I", "study", "complex", "language", "JavaScript"
```

Índices negativos permitidos

Aquí y en otros métodos de array, se permiten índices negativos. Especifican la posición desde el final de la matriz, como aquí:

```
let arr = [1, 2, 5];

// from index -1 (one step from the end)
// delete 0 elements,
// then insert 3 and 4
arr.splice(-1, 0, 3, 4);

alert( arr ); // 1,2,3,4,5
```

3.1.3.- slice

El método arr.slice suele ser confundido muchas veces con `arr.splice` por el nombre aunque su función es más sencilla.

La sintaxis es:

```
arr.slice([start], [end])
```

Devuelve una nueva matriz copiando todos los elementos desde el índice start hasta end(sin incluir end). Ambos start y end pueden ser negativos, en ese caso se supone la posición desde el final de la matriz.

Es similar a un método de cadena str.slice, pero en lugar de subcadenas crea subarrays.

Por ejemplo:

```
let arr = ["t", "e", "s", "t"];

alert( arr.slice(1, 3) ); // e,s (copy from 1 to 3)

alert( arr.slice(-2) ); // s,t (copy from -2 till the end)
```

3.1.4.- concat

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

La sintaxis es:

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

Acepta cualquier número de argumentos, ya sean matrices o valores.

El resultado es una nueva matriz que contiene elementos de arr, y arg1, arg2 etc.

Si un argumento argN es una matriz, todos sus elementos se copian. De lo contrario, el argumento en sí se copia.

Por ejemplo:

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

// create an array from: arr and [3,4]
alert( arr.concat([3, 4]) ); // 1,2,3,4

// create an array from: arr and [3,4] and [5,6]
alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6

// create an array from: arr and [3,4], then add values 5 and 6
```

```
alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

3.2.- Iterar para cada elemento

El método arr.forEach permite ejecutar una función para cada elemento de la matriz.

Esto es muy habitual para métodos modernos sobre los arrays, se implementan los métodos con una función anónima asociada. Se puede utilizar la sintaxis de funciones anónimas aunque desde ES6 esta recomendado la utilización de la sintaxis fat Arrow pues es más simple e intuitiva.

La sintaxis:

```
arr.forEach(function(item, index, array) {
  // ... do something with item
});
```

La función lo que hace es ir pasando por cada elemento del array y permite tratar en el cuerpo de la función el elemento (item), la posición en que se encuentra (index) y la variable array sobre la que actúa (array).

Generalmente en la función se utiliza únicamente un argumento que hace referencia al elemento actual del array.

Por ejemplo, esto muestra cada elemento de la matriz:

```
// for each element call alert
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
```

De otra forma con funcion anónima

```
// for each element call alert
["Bilbo", "Gandalf", "Nazgul"].forEach(function(n){alert(n)});
```

De otra forma con fat Arrow

```
// for each element call alert
```

```
["Bilbo", "Gandalf", "Nazgul"].forEach(n=>alert(n));
```

Y este código es más elaborado y trabaja con los tres argumentos:

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

El resultado de la función (si devuelve alguno) se descarta y se ignora. Generalmente se utiliza para realizar acciones sobre otras variable, como realizar cálculos de los elementos del array

3.3.- Buscar en arrays

Ahora veamos los métodos que buscan en un array.

3.3.1.- indexOf / lastIndexOf e includes

Los métodos arr.indexOf, arr.lastIndexOf y arr.includes tienen la misma sintaxis y hacen esencialmente lo mismo que sus homólogas de cadena, pero operan en elementos en lugar de caracteres:

- **arr.indexOf(item, from)** busca en el array el ítem comenzando desde el índice from, y devuelve el índice donde se encontró, de lo contrario -1.
- **arr.lastIndexOf(item, from)** igual, pero busca de derecha a izquierda.
- **arr.includes(item, from)** busca en el array el ítem comenzando desde el índice from, devuelve truesi se encuentra.

Por ejemplo:

```
let arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1
alert( arr.indexOf(false) ); // 2
alert( arr.indexOf(null) ); // -1

alert( arr.includes(1) ); // true
```

Si queremos verificar la inclusión y no queremos saber el índice exacto, entonces `arr.includes` es preferible.

3.3.2.- find and findIndex

Imagina que tenemos una gran variedad de elementos en el array. ¿Cómo encontramos un objeto con la condición específica?

Aquí el método arr.find (fn) es útil.

La sintaxis es:

```
let result = arr.find(function(item, index, array) {
  // if true is returned, item is returned and iteration is stopped
  // for falsy scenario returns undefined
});
```

La función se llama para elementos del array, uno tras otro:

- `ítem` es el elemento
- `index` es su índice.
- `array` es la matriz en sí misma.

Debemos poner una condición en el cuerpo de la función que determine si el elemento actual es el que estamos buscando. Si devuelve `true`, la búsqueda se detiene e `ítem` se devuelve. Si no se encuentra nada se devuelve `undefined`.

Vamos a ver un ejemplo de un array de elementos que son objetos, cada objeto tiene un id y un nombre.

Vamos a buscar el que tiene `id == 1`:

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];

let user = users.find(item => item.id == 1);

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

En la vida real, las matrices de objetos son algo común, por lo que el método `find` es muy útil.

En el ejemplo proporcionamos a `find` la función `item => item.id == 1` con un argumento. Eso es típico, otros argumentos de esta función rara vez se usan.

El método arr.findIndex es esencialmente el mismo, pero devuelve el índice donde se encontró el elemento en lugar del elemento mismo y se devuelve `-1` cuando no se encuentra nada.

3.3.3.- filter

El método `find` busca un único (primer) elemento que hace que la función devuelva `true`.

Si puede haber muchos, podemos usar arr.filter (fn).

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

```
let results = arr.filter(function(item, index, array) {
  // if true item is pushed to results and the iteration continues
  // returns empty array if nothing found
});
```

Por ejemplo:

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];

// returns array of the first two users
let someUsers = users.filter(item => item.id < 3);

alert(someUsers.length); // 2
```

3.4.- Transformar un array

Pasemos a ver los métodos que transforman y reordenan una matriz.

3.4.1.- map

El método arr.map es uno de los más útiles y de uso frecuente.

Llama a la función para cada elemento de la matriz y devuelve el array de resultados.

La sintaxis es:

```
let result = arr.map(function(item, index, array) {
  // returns the new value instead of item
});
```

Por ejemplo, aquí transformamos cada elemento en su longitud:

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

3.4.2.- sort (fn)

La llamada a arr.sort () ordena el array sobre el propio array cambiando su orden de elementos.

También devuelve el array ordenado, pero el valor devuelto generalmente se ignora, ya que `arr` se modifica a sí mismo.

Por ejemplo:

```
let arr = [ 1, 2, 15 ];

// the method reorders the content of arr
arr.sort();

alert( arr );  // 1, 15, 2
```

¿Notaste algo extraño en el resultado?

El orden se hizo`1, 15, 2`. Incorrecto.¿Pero por qué?

Los elementos se ordenan como cadenas de forma predeterminada.

Literalmente, todos los elementos se convierten en cadenas para realizar comparaciones. Para las cadenas, se aplica el orden lexicográfico y de hecho `"2" > "15"`.

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

La función debe comparar dos valores arbitrarios y devolver:

```
function compare(a, b) {
  if (a > b) return 1; // if the first value is greater than the second
  if (a == b) return 0; // if values are equal
  if (a < b) return -1; // if the first value is less than the second
}
```

Por ejemplo, para ordenar como números:

```
function compareNumeric(a, b) {
  if (a > b) return 1;
  if (a == b) return 0;
  if (a < b) return -1;
}

let arr = [ 1, 2, 15 ];

arr.sort(compareNumeric);

alert(arr);  // 1, 2, 15
```

Ahora funciona según lo previsto.

Pensemos qué está pasando. El `arr` puede ser una variedad de cualquier cosa, ¿verdad?. Puede contener números o cadenas u objetos o lo que sea. Tenemos un conjunto de *algunos artículos*. Para ordenarlo, necesitamos una *función de orden* que sepa cómo comparar sus elementos. El valor predeterminado es un orden de cadena.

El `arr.sort(fn)` método implementa un algoritmo genérico de clasificación. No es necesario que nos importe cómo funciona internamente (una ordenación de tipo quicksort optimizada). Recorrerá la matriz, comparará sus elementos con la función proporcionada y los reordenará, todo lo que necesitamos es proporcionar la `fn` que hace la comparación.

Por cierto, si alguna vez queremos saber qué elementos se comparan, nada impide alertarlos:

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

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

Una función de comparación puede devolver cualquier número

En realidad, solo se requiere una función de comparación para devolver un ***número positivo para decir "mayor"*** y un **número negativo para decir "menor"**.

Eso permite escribir funciones más cortas:

```
let arr = [ 1, 2, 15 ];

arr.sort(function(a, b) { return a - b; });

alert(arr);  // 1, 2, 15
```

Funciones de flecha para lo mejor

Las fat Arrow las podemos usarlos aquí para una sintaxis más clara:

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

Esto funciona exactamente igual que la versión más larga anterior.

Usar `localeCompare` para strings

El algoritmo de comparación de cadenas compara letras por sus códigos por defecto.

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

Por ejemplo, clasifiquemos algunos países en alemán:

```
let countries = ['Österreich', 'Andorra', 'Vietnam'];

alert( countries.sort( (a, b) => a > b ? 1 : -1) ); // Andorra, Vietnam, Österreich (wrong)

alert( countries.sort( (a, b) => a.localeCompare(b) ) ); // Andorra,Österreich,Vietnam (correct!)
```

3.4.3.- reverse

El método arr.reverse invierte el orden de los elementos del arr.

Por ejemplo:

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

alert( arr ); // 5,4,3,2,1
```

También devuelve el array arr después de la inversión.

3.5.- String – Array

Ya comentamos en el objeto String que había una relación entre cadenas y arrays a través de las funciones split y join.

3.5.1.- split

Veamos un ejemplo: estamos escribiendo una aplicación de mensajería, y la los receptores del mensaje se añaden en CC separados por , y " ": John, Pete, Mary. Pero para nosotros una serie de nombres sería mucho más cómoda que una sola cadena. ¿Cómo conseguirlo?

El método str.split (delim) hace exactamente eso. Divide la cadena en un array por el delimitador dado delim.

En el siguiente ejemplo, dividimos la cadena de texto por una coma seguida de espacio:

```
let names = 'Bilbo, Gandalf, Nazgul';

let arr = names.split(', ');

for (let name of arr) {
  alert( `A message to ${name}.` ); // A message to Bilbo  (and other names)
}
```

El método split tiene un segundo argumento numérico opcional: un límite en la longitud de la matriz. Si se proporciona, los elementos adicionales se ignoran. Sin embargo, en la práctica rara vez se usa:

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

alert(arr); // Bilbo, Gandalf
```

Dividir en letras

La llamada a split(s) con una cadena vacía como separador s dividiría la cadena en un array de letras:

```
let str = "test";

alert( str.split('') ); // t,e,s,t
```

3.5.2.- join

La llamada arr.join (separador) hace lo contrario a split. Crea una cadena de elementos a partir de arr unidos por entre ellos por separador.

Por ejemplo:

```
let arr = ['Bilbo', 'Gandalf', 'Nazgul'];

let str = arr.join(';'); // glue the array into a string using ;

alert( str ); // Bilbo;Gandalf;Nazgul
```

3.6.- Sobre el tipo array

Trabajando con arrays (***o que pensamos que son arrays***) pueden surginos un par de dudas:

Hay ocasiones en las que necesitamos saber si un objeto es realmente un array, para ello deberíamos comprobar si el tipo es exactamente un array. En JavaScript existen otro tipo de objetos iterables similares a los arrays (nodeList y HTMLCollection) pero no son realmente arrays, con lo cual no se pueden aplicar los métodos de arrays.

Si nos encontráramos en esa situación sería interesante poder convertir esos objetos en arrays para poder utilizar los métodos vistos.

Para ello disponemos de dos métodos que resuelven estos problemas.

3.6.1.- Array.isArray

Los arrays no forman un tipo del lenguaje aparte. Se basan en objetos.

Por lo tanto typeof no ayuda a distinguir un objeto de un array:

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

Pero los arrays se usan con tanta frecuencia que hay un método especial para eso: Array.isArray (valor) .

Devuelve true si value es una matriz, y de lo contrario false.

```
alert(Array.isArray({})); // false


alert(Array.isArray([])); // true
```

3.6.2.- Array.from

El método array.from(object) permite convertir en array objetct. Esto es muy interesante cuando trabajamos con objetos que aparentemente son Arrays pero realmente no lo son.

Esto lo veremos en el DOM.

Los objetos **NodeList** son colecciones de nodos como los devueltos por propiedades como Node.childNodes y métodos como document.querySelectorAll (), document.getElementsBy…

Aunque NodeList no es un Array, es posible iterar sobre él utilizando forEach(). También puede convertirse a un Array usando Array.from.

Un HTMLCollection es un objeto similar a un array que representa una colección de elementos HTML extraídos del documento
Se puede tener acceso a los elementos de la colección mediante un índice numérico, un nombre o un identificador

```
const p = document.getElementsByTagName('p')[2];
```

A diferencia del NodeList, no es compatible con el método forEach(). Sin embargo, se puede utilizar el método Array.from(HTMLCollection) para convertir la colección a un array normal y, a continuación, utilizar froeach() para iterar sobre él.

```
const elems = document.getElementsByTagName('p');

// itereate using forEach()
Array.from(elems).forEach((p, index) => console.log(p.innerText));
```

4.- Resumen de métodos de arrays

Métodos por tipo de operación:

Para agregar / eliminar elementos:

- push (items) agrega elementos al final,
- pop() extrae un elemento del final,
- shift() extrae un elemento desde el principio,
- unshift (items) agrega elementos al principio.
- Splice (pos, deleteCount, items) en el índice pos eliminar deleteCount elementos e insertar items.
- slice(start, end) crea un nuevo array, copia elementos desde la posición start hasta end (no incluye end).

- concat (items) devuelve un nuevo array: copia todos los miembros del actual y le agrega items. Si alguno de ítems es un array, entonces se añaden sus elementos como si fueran individuales.

Para buscar elementos:

- indexOf/lastIndexOf(item, pos) buscar ítem a partir de la posición pos, devolver el índice o -1 si no se encuentra.
- includes(value) devuelve true si el array tiene value, de lo contrario false.
- find/filter(func) filtrar elementos a través de la función, devolver primero o todos los valores que lo hacen true en la func.
- findIndexes como find, pero devuelve el índice en lugar de un valor.

Para iterar sobre elementos:

- forEach(func)- llama a func para cada elemento, no devuelve nada.

Para transformar array:

- map(func) crea un nuevo array a partir de los resultados de la llamada func para cada elemento.
- sort(func) ordena el array en su lugar, luego la devuelve.
- reverse() invierte el array en su lugar, luego la devuelve.

String - Array

- split/join convierte una cadena en array y viceversa.

Tipo array:

- Array.isArray(arr) comprueba si arr es un array.
- Array.from(obj) convierte obj en un array.

Hay muchos más métodos parra arrays, pero estos son los más utilizados.
Para la lista completa, vea el manual.