Arrays Avanzados 3

Javascript nos permite un control de las estructura tipo Array mucho más completo que lo que pudimos ver en la primera parte de este manual.

Por ejemplo, al igual que ocurría con las cadenas, vamos a tener a nuestra disposición una serie de funciones ya predefinidas que nos facilitarán el realizar operaciones tanto con Arrays como entidades como con cada uno de sus elementos. Además vamos a poder crear un par de estructuras complejas basadas en Arrays: los arrays multidimensionales y los arrays asociativos.

Para finalizar, Javascript proporciona un sistema de notación propio llamado **JSON** (*Javascript Object Notation*) el cual nos va a permitir, entre otras cosas, definir de manera muy sencilla Arrays y estructuras basadas en arrays tan complejas como queramos.

3.1 Trabajando con Arrays

Javascript posee una buena cantidad de herramientas para el manejo de estructuras tipo Array. A continuación se van a comentar las más relevantes:

3.1.1 Definición

Aunque el uso de new Array es correcto, actualmente es mucho más rápido y sencillo definir los arrays usando la nomenclatura JSON:

```
let formaLarga = new Array('a','e','i','o','u');
let formaCorta = ['a','e','i','o','u'];
```

```
let vacioLargo = new Array();
let vacioCorto = [];
```

Aunque hay un tema completo dedicado a JSON en este manual, de momento vamos a quedarnos con la nueva forma de definir arrays, que es mucho más sencilla.[17]

3.1.2 length

Indica el número de elementos que posee el Array. En realidad `length` no es una función, es una variable asociada a cada array. Por ese motivo no lleva paréntesis ni argumentos.

```
var vocales = new Array('a','e','i','o','u');
var total = vocales.length;
aler(total); //5
```

3.1.3 concat()

Esta función nos permite concatenar elementos a un array. Devuelve un array con los elementos nuevos concatenados, es decir, no modifica el array sobre el que se invoca la función.

```
var vocales = ['a','e','i','o','u'];
var nuevo = vocales.concat(1,2,3,4,5);
//'nuevo' será un array con 10 elementos
// 'a','e','i','o','u',1,2,3,4,5
```

17 En otros lenguajes modernos como Python, la única forma de definir arrays (y otras estructuras) es usando notación JSON.

3.1.4 pop()

Esta función elimina el ultimo elemento del array y devuelve dicho elemento. Es decir, el array sobre el que se invoca esta función se modifica y disminuye su longitud.

```
var vocales = ['a','e','i','o','u'];
var ultima = vocales.pop();

//'ultima' tendrá el valor 'u'
// 'vocales' queda con: 'a','e','i','o'
```

3.1.5 push()

Esta función añade un elemento al final del array. El elemento se le pasa como argumento. El array sobre el que se invoca la función queda afectado y aumenta su longitud.

```
var vocales = ['a','e','i','o','u'];
vocales.push('tururu');

// 'vocales' queda con: 'a','e','i','o','u','tururu'
```

Esta función es muy usada cuando se deben añadir más elementos a un array del que no se conoce su tamaño.

3.1.6 shift()

Esta función es exactamente igual a `pop()` pero elimina y devuelve el primer elemento del array

```
var vocales = ['a','e','i','o','u'];
var primera = vocales.shift();
```

```
//'primera' tendrá el valor 'a'
// 'vocales' queda con: 'e','i','o','u'
```

3.1.7 unshift()

El funcionamiento de esta función es análogo a `push()` con la diferencia de que `unshift()` añade el elemento por el principio del array.

```
var vocales = ['a','e','i','o','u'];
vocales.unshift('tarara');
// 'vocales' queda con: 'tarara','a','e','i','o','u'
```

3.1.8 reverse()

Esta función toma el array sobre el que se invoca y reordena sus elementos de forma inversa. Es decir, el orden del array original queda modificado.

```
var vocales = ['a','e','i','o','u'];
vocales.reverse();
// 'vocales' queda con: 'u','o','i','e','a'
```

3.1.9 join()

Esta función es la contraria a la función de cadenas `split()`. Necesita como argumento un carácter separador y devuelve una cadena con todos los elementos del array unidos por el carácter separador:

```
var vocales = ['a','e','i','o','u'];
var cadena1 = vocales.join(" "); //espacio en blanco
var cadena2 = vocales.join(""); //ningún caracteres
var cadena3 = vocales.join("-"); //espacio en blanco
```

```
// 'cadena1' será la cadena: 'a e i o u'
// 'cadena2' será la cadena: 'aeiou'
// 'cadena3' será la cadena: 'a-e-i-o-u'
```

3.2 Métodos para los elementos de un Arrays

En JavaScript existen funciones que admiten como argumentos otras funciones[18].

Unas de las mas usadas son las que afectan a los elementos de un array. Es decir, tenemos la posibilidad de usar funciones que, automáticamente van a recorrer un array y van a ejecutar la función que pasemos como parámetro en cada uno d ellos elementos del array.

De esta forma en JavaScript se suele evitar programar los típicos bucles que aparecen al recorrer los elementos de un array, ahorrando así mucho código.

Destacar que en este tipo de funciones son bastante usadas la notación de funciones flecha en las funciones definidas en los parámetros, ya que permiten ahorrar mucho código.

3.2.1 some()

Comprueba si ALGÚN elemento del array cumple la condición que se le pasa como parámetro. Devuelve `true` si algún elemento cumple la condición indicada o `false` si no es así.

Supongamos el array:

```
var lista = [2,4,6,8,9,10,15,25,31,46,59,62,75]
```

Y la función:

```
function mayorEdad(edad){
        return edad>=18;
}
```

18 A esas funciones que se pasan como argumentos se les llama callbacks.

Podemos aplicar la función `some` así:

```
//La variable alguno tendrá el valor true
var alguno = lista.some(mayorEdad);
```

Importante destacar que al colocar la función mayorEdad como argumento, no hace falta indicar valor de la edad como parámetro. Esto es porque la función `some` se encarga automáticamente de rellenar ese valor. Además, en este caso, como no hacen falta argumentos, no se colocan paréntesis.

Mismo ejemplo usando funciones anónimas:

```
var alguno = lista.some(function(edad){
        return edad>18;
    });
//La variable alguno tendrá el valor true
console.log(alguno)
```

De esta forma, la función `mayorEdad` se define como función anónima dentro de los argumentos de `some`. Destacar que, salvo el nombre, todo se define igual: argumentos que necesita e instrucciones a realizar.

Mismo ejemplo usando arrow functions:

```
var alguno = lista.some((ele)=>(ele>=18));
//La variable alguno tendrá el valor true
console.log(alguno)
```

En esta forma hemos definido la función de forma anónima usando la notación flecha.

Recuerda que `some` necesita una condición como argumento, es decir, las funciones que usemos como argumento deben devolver un valor booleano.

3.2.2 every()

Comprueba si TODOS los elemento del array cumplen la condición que se le pasa como parámetro. Devuelve `true` si todos los elementos la cumplen o `false` si no es así.

Supongamos el array:

```
var lista = [2,4,6,8,9,10,15,25,31,46,59,62,75]
```

Y la función:

```
function mayorEdad(edad){
        return edad>=18;
}
```

Podemos aplicar la función `every` así:

```
//La variable todos tendrá el valor false
var todos = lista.every(mayorEdad);
```

Mismo ejemplo usando funciones anónimas:

```
var todos = lista.every(function(edad){
        return edad>18;
    });
//La variable todos tendrá el valor false
console.log(alguno)
```

Mismo ejemplo usando arrow functions:

```
var todos = lista.every((ele)=>(ele>=18));
//La variable todos tendrá el valor false
console.log(alguno)
```

Recuerda que `every` necesita una condición como argumento, es decir, las funciones que usemos como argumento deben devolver un valor booleano.

3.2.3 forEach()

Recorre automáticamente el array y ejecuta la función indicada con cada uno de los elementos del array.

Supongamos el array:

```
var lista = [2,4,6,8,9,10,15,25,31,46,59,62,75]
```

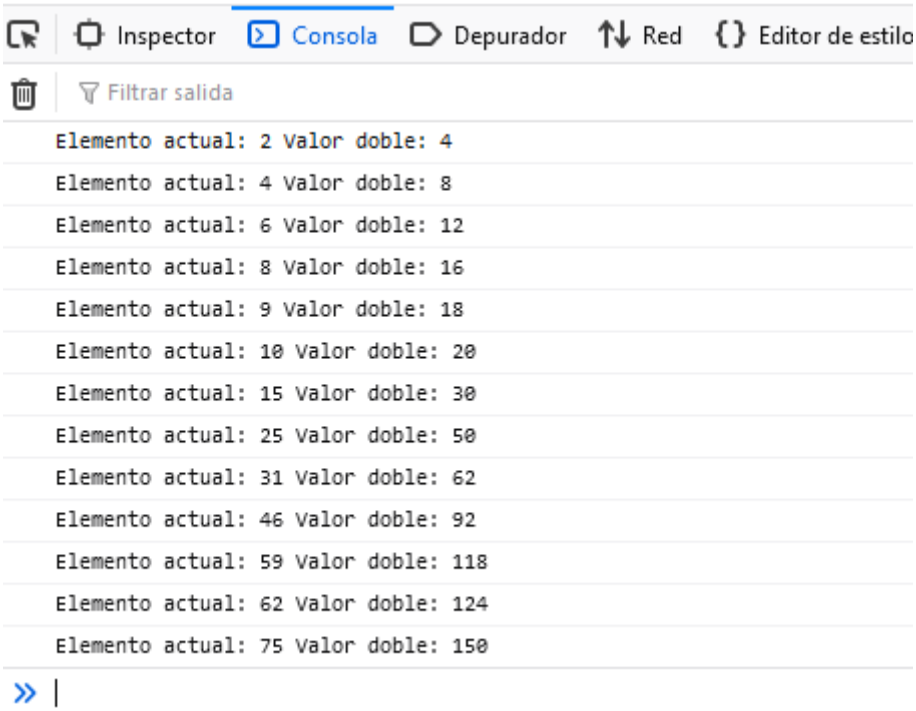
Y la función:

```
function mostrar(ele){
    var res = ele*2;
    console.log("Elemento actual: "+ele+ " Valor doble: "+res)
}
```

Ejecutando un forEach...

```
lista.forEach(mostrar);
```

...obtenemos:



Importante destacar que sale algo por consola porque la función que hemos definido así lo hace, no porque el `forEach` muestre nada.

Fíjate que sólo la linea: `lista.forEach(mostrar);` equivale a hacer un bucle FOR para recorrer el array.

```
for(let i=0; i<lista.length; i++){
 console.log("Elemento actual: "+lista[i]+ " Valor doble: "+
(lista[i]*2));
}
```

Mismo ejemplo usando otras notaciones a la hora de definir la función parámetro:

Usando funciones anónimas:

```
lista.forEach(function(ele){
    var res = ele*2;
    console.log("Elemento actual: "+ele+ " Valor doble: "+res)
    });
```

Mismo ejemplo usando arrow functions:

```
lista.forEach((ele)=>{
    var res = ele*2;
    console.log("Elemento actual: "+ele+ " Valor doble: "+res);
});
```

`forEach` tiene una limitación importante: no puede devolver ningún valor. Es decir, no podemos devolver nada en la función que usemos como parámetro. Aunque usemos un `return` no funcionará:

```
var res = lista.forEach((ele)=>{
        return ele*2;
});
console.log(res); //obtenemos undefined
```

Para solucionar esta carencia JavaScript posee la siguiente función:

3.2.4 map()

Esta función obtiene un nuevo array con los elementos del array original transformados según indique la función que se le pasa como parámetro. Es decir, se aplica una acción sobre el valor correspondiente y el resultado se guarda en el array resultante.

Por tanto, al usar `map` obtenemos siempre un array con la misma longitud que el array al que se recorre.

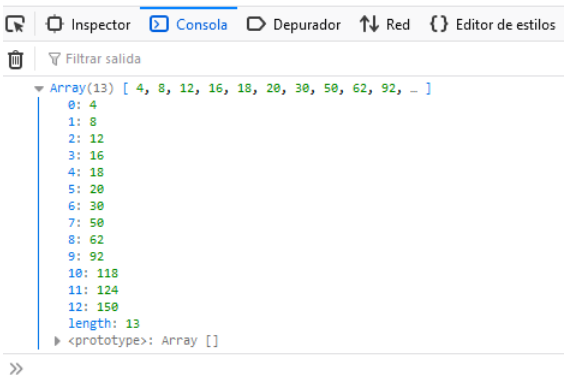
Supongamos el array:

```
var lista = [2,4,6,8,9,10,15,25,31,46,59,62,75]
```

Si hacemos...

```
var res = lista.map((ele)=>{
   return ele*2;
});
console.log(res);
```

...obtenemos

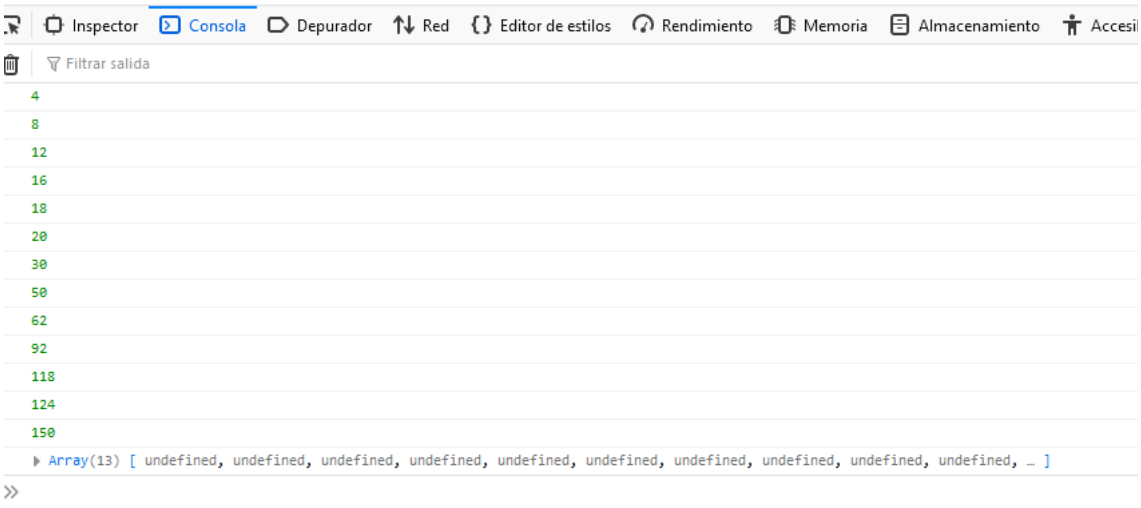


El array que se almacena en la variable `res` tiene todos los valores devueltos por `return` en cada vuelta.

Si usamos una función parámetro sin `return`:

```
var res = lista.map((ele)=>{
   console.log(ele*2);
});
console.log(res);
```

obtenemos:



Se imprimen los valores en cada vuelta tal y como se indican en la función parámetro y, al mostrar el contenido de la variable `res`, obtenemos un array con valores `undefined` porque no hay `return` en la función parámetro.

Importante: `map` y `foreach` se pueden confundir. Ambos funcionan de forma similar salvo que `map` devuelve un array. Es una mala práctica de programación usar `map` si no se va a hacer uso del array resultante (usa `forEach` en su lugar).

3.2.5 filter()

Este método nos permite filtrar los elementos de un array bajo unos criterios y nos devuelve un nuevo array con el resultado de ese "filtro". Es decir, a diferencia de `map` que siempre devuelve un array del mismo tamaño que el array que recorre, `filter` puede devolver un array de menor tamaño que el que recorre.

Supongamos el array:

```
var lista = [1,5,23,4,12,45,78,8,8.9,10,11,3.4,10.1,84,6]
```

Queremos obtener un nuevo array con valores mayores a 10.

El primer razonamiento de alguien que sabe programar pero que no está familiarizado con las funciones que tiene JavaScript es el de usar un bucle para recorrer el array y, en cada vuelta, comprobar si ese elemento es mayor a 10 o no lo es. En caso afirmativo, mete ese valor en el array final.

Lo anterior funciona bien pero, si usamos `filter`, el código se reduce bastante:

```
let listaFiltrada = lista.filter(function(elemento){
      return (elemento > 10)
})
```

o mejor, usando funciones flecha:

```
let listaFiltrada = lista.filter((ele)=> (ele > 10)})
```

Lo que hace este método es recorrer el array y para cada elemento comprobar si la función anónima/flecha retorna true. Hecho esto en todos los elementos, devuelve un nuevo array con todos los items que pasaron la prueba.

Importante: La función anónima/flecha usada debe devolver un valor condicional para hacer el filtro. Si esto no ocurre, filter devolverá un valor vacío:

Ejemplo:

```
let listaFiltrada = lista.filter(function(elemento){
     elemento > 10
})
```

`ListraFiltrada` no tendrá elementos porque esa función anónima no devuelve nada.

3.2.6 Aspectos comunes de los Array Method

Todas los funciones vistas en este apartado tienen una cosa en común: necesitan una función como parámetro.

Además, en todos los casos, esa función usada puede definirse con entre 0 y 3 argumentos que se rellenan de forma automática por orden:

1. **Elemento actual del array**. Este valor cambia en cada vuelta.
2. **Posición del elemento actual** dentro del array. Este valor cambia en cada vuelta.
3. **Array que se recorre**. Este valor no cambia.

Para que funcione todo bien, hay que respetar el orden:

- si hay un parámetro, será el elemento actual.
- si hay dos parámetros, el elemento y la posición.
- Si hay tres parámetros, elemento, posición y array recorrido.

El uso del último parámetro es debido a si queremos tocar el array que se recorre dentro de la función (algo que puede ser peligroso).

3.3 Spread Operator

Este operador (operador de propagación en español) es muy versátil y nos permitirá transformar, copiar y concatenar arrays de una forma rápida y sencilla.

El operador se compone del carácter punto repetido 3 veces y se debe usar siempre como primer elemento del array: `[...`

3.3.1 Copiando arrays

Tal y como ocurre en otros lenguajes de programación, cuando asignamos una variable que contiene un array a otra, no se realiza una copia de esta sino que se crea una nueva referencia. Es decir, hay dos variables apuntando al mismo array. Si una variable cambia el contenido del array, la otra variable ve esos cambios:

```
let vocales = ['a','e','i','o','u'];
let copia = vocales; //asigno una copia
copia[0] = 25; //cambio valores de la copia
console.log(vocales) //[ 25, 'e','i','o','u' ]
```

Esto es algo que podemos solucionar usando el operador de propagación:

```
let vocales = ['a','e','i','o','u'];
let copia = [...vocales]; //asigno una copia
copia[0] = 25; //cambio valores de la copia
console.log(vocales) //[ 'a', 'e','i','o','u' ]
```

3.3.2 Concatenando arrays

Con el operador de propagación podemos concatenar arrays de forma sencilla:

```
let vocales = ['a','e','i','o','u'];
let numeros = new Array(1,2,3,4,5);
let mix = [...vocales,...numeros];

console.log(mix); //[ 'a', 'e', 'i', 'o', 'u', 1, 2, 3, 4, 5 ]
```

3.3.3 Transformando en arrays

Si usamos el operador de propagación sobre una cadena de caracteres, vamos a obtener el mismo resultado que si usáramos el método `.split` sin separador en dicha cadena:

```
let frase = "Martillo va!"
let arrayT = [...frase];
console.log(arrayT);
//[ 'M', 'a', 'r', 't', 'i', 'l', 'l', 'o', ' ', 'v', 'a', '!' ]
```

3.4 Array multidimensionales

Un array multidimensional es una estructura muy en programación compuesta de un arrays donde cada elemento de ese array es a su vez, otro array. Además, esos arrays pueden contener, a su vez, otros arrays como elementos y seguir así hasta que queramos.

Cada vez que hacemos esto (definir arrays cuyos elementos son arrays) se dice que estamos añadiendo una dimensión a la estructura.

Dependiendo de las veces que cumplamos la condición anterior a la hora de construir la estructura, podemos tener por ejemplo:

- Estructuras de 2 dimensiones (bidimensionales): son aquellas donde los elementos del array son a su vez arrays de elementos (números, cadenas, booleanos...). Son estructuras muy comunes que se usan para representar mapas, matrices de elementos, tableros, coordenadas...
- Estructuras de 3 dimensiones (tridimensionales): los elementos del array son arrays que a su vez tienen arrays de elementos. Se suelen usar para representar ubicaciones o instancias en entornos 3D.
- Estructuras de más de 3 dimensiones (multidimensionales): se sigue la regla de colocar arrays de arrays tantas veces como haga falta. Son estructuras complejas que se salen del objetivo de este manual.

3.4.1 Definiendo arrays multidimensionales

La manera más usada para definir arrays multidimensionales en Javascript por su sencillez es la notación JSON. Sin embargo, eso es algo que se enseñará más adelante en este manual, por tanto, en este apartado se va a mostrar otra forma de definir estas estructuras que, si bien es más tediosa (hay que escribir más), es mucho más intuitiva para aquellos lectores que conozcan otros lenguajes de programación.

La idea básica es definir arrays en los que cada uno de sus elementos son a su vez otro array.
Aunque Javascript puede manejar arrays de cualquier dimensión, en este manual vamos a trabajar con arrays de 2 dimensiones ya que son los más comunes dentro del desarrollo de aplicaciones web.

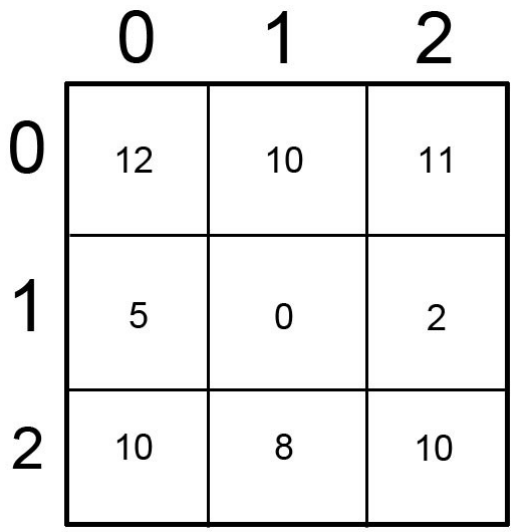
En el siguiente ejemplo vamos a crear un array de dos dimensiones donde tendremos por un lado ciudades y por el otro la temperatura media que hace en cada una durante de los meses de invierno.

```
var temperaturas_medias_ciudad0 = new Array(12,10,11);

var temperaturas_medias_ciudad1 = new Array (5,0,2);

var temperaturas_medias_ciudad2 = new Array (10,8,10);
```

Con las anteriores líneas hemos creado tres arrays de 1 dimensión con tres elementos numéricos cada uno.
Ahora crearemos un nuevo array de tres elementos e introduciremos dentro de cada una de sus casillas los arrays creados anteriormente, con lo que tendremos un array de arrays, es decir, un array de 2 dimensiones.

```
var temperaturas_cuidades = new Array (3);
temperaturas_cuidades[0] = temperaturas_medias_ciudad0;
temperaturas_cuidades[1] = temperaturas_medias_ciudad1;
temperaturas_cuidades[2] = temperaturas_medias_ciudad2;
```

El array `temperaturas_cuidades` sería un array bidimensional:



3.4.2 Recorriendo arrays bidimensionales

Los arrays vistos en la primera parte de este manual (array '*norma*l') almacenan valores en una dimensión, por eso para acceder a las posiciones utilizamos tan solo un índice.

Cuando trabajamos con arrays multidimensionales, para acceder a sus elementos vamos a necesitar tantos indices como dimensiones tenga el array.

Para los arrays bidimensionales vamos a necesitar dos índices. Estos arrays pueden verse como una tabla con filas y columnas. El primer índice serviría para movernos por las filas, y el segundo para las celdas de cada fila (las columnas).

Si quisiéramos mostrar elementos del array que hemos creado en el apartado anterior:

```
alert(temperaturas_cuidades[0][0]); //12
alert(temperaturas_cuidades[0][2]); //11
alert(temperaturas_cuidades[2][1]); //8
```

Como se ve en el ejemplo, debemos hacer uso de dos índices para acceder a un elemento de la estructura.

Si quisiéramos recorrer toda la estructura, debemos recorrer el array original y, a su vez, ir recorriendo el array que corresponde a cada elemento del array original. Es decir, siguiendo la analogía antes comentada, vamos recorriendo las filas y dentro de cada fila, recorremos las celdas.

El método para hacer un recorrido dentro de otro es colocar un bucle dentro de otro, lo que se llama un bucle anidado. Así que necesitamos meter un bucle FOR dentro de otro:

```
//Bucle para recorrer las filas
for (let i=0;i<temperaturas_cuidades.length; i++){
 //bucle para recorrer las celdas de cada fila
  for (j=0;j<temperaturas_cuidades[i].length; j++){
      console.log(temperaturas_cuidades[i][j]);
  }
}
```

Es importante tener claro las vueltas que debe dar cada bucle FOR: Para el bucle más externo la solución es simple: dará tantas vueltas como elementos tenga el array `temperaturas_ciudades`. ¿Pero qué pasa con el resto?

El ejemplo que hemos expuesto es un caso muy sencillo ya que hemos creado una estructura simétrica, es decir tiene el mismo numero de filas que de columnas (en este caso 3x3).

Sin embargo, los arrays que van dentro del array principal no tienen porque tener el mismo tamaño. Pej:

```
var temperaturas_medias_ciudad0 = new Array(12,10,11);
var temperaturas_medias_ciudad1 = new Array (5,0,2,6);
var temperaturas_medias_ciudad2 = new Array (10,3);


var temperaturas_cuidades = new Array (3);
temperaturas_cuidades[0] = temperaturas_medias_ciudad0;
temperaturas_cuidades[1] = temperaturas_medias_ciudad1;
temperaturas_cuidades[2] = temperaturas_medias_ciudad2;
```

El array principal (`temperaturas_ciudades`) tendrá tres elementos, pero cada uno de sus arrays interiores van a tener diferentes longitudes (3, 4 y 2 respectivamente).

Es por eso que, en el FOR anidado es necesario indicar una sentencia que se adapte a cada caso y coloque correctamente la longitud del array que voy a recorrer.

Para el ejemplo hemos usado `temperaturas_cuidades[i].length` la cual nos va a dar en cada momento, la longitud del array por el que vamos a medida que avancemos por el array principal.

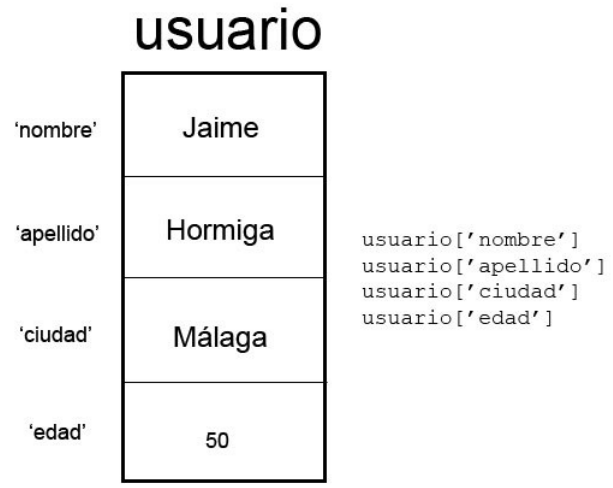
```
for (let i=0;i<temperaturas_cuidades.length; i++){
  for (j=0;j<temperaturas_cuidades[i].length; j++){
      console.log(temperaturas_cuidades[i][j]);
  }
}
```

3.5 Array asociativos (mapas)

Los arrays asociativos son un tipo de array muy útiles y muy comunes en los lenguajes de programación que consisten en arrays cuyos indices no son números, sino cadenas de texto[19]. Es decir, para acceder a una celda de un array asociativo vamos a usar su cadena indice en lugar del número de su posición.

Esto es especialmente útil porque, por un lado, somos nosotros los que decidimos que nombre (cadena) va a tener cada celda del array, y por otro lado, ya no es importante el orden de las celdas puesto que no voy a usar indices numéricos para acceder a ellas.

Ejemplo del típico array asociativo:



19 En otros lenguajes se les llama mapas o diccionarios.

3.5.1 Definiendo arrays asociativos

A diferencia de otros lenguajes, Javascript no posee el elemento 'array asociativo' como tal. Lo que se hace en realidad es definir un objeto cuyo comportamiento imita el funcionamiento de los arrays asociativos.

Ahora mismo, para nosotros, el 'cómo' Javascript gestiona los arrays asociativos no es importante ya que todo se hace de forma transparente para el programador.

```
var usuario = new Array();

//definir elementos
usuario['nombre'] = 'Jaime';
usuario['apellido'] = 'Hormiga';
usuario['ciudad'] = 'Málaga';
usuario['edad'] = 50;

//mostrar elementos
alert(usuario['nombre']);
alert(usuario['apellido']);
alert(usuario['ciudad']);
alert(usuario['edad']);
```

***Nota**: Aunque la forma de definir arrays asociativos indicada en el ejemplo no está mal y es totalmente válida, al igual que ocurría con los arrays multidimensionales, la forma más utilizada con diferencia es la notación JSON*.

Dada la naturaleza especial de estos arrays en Javascript, al crear un array asociativo nos encontramos con una serie de problemas:

- Los arrays asociativos no disponen de un método que nos devuelva su longitud (`length` devuelve 0 o `undefined`).
- Al no poder controlar la longitud del array, no podemos usar bucles FOR normales para recorrerlos[20].
- Muchas de las funciones vistas para arrays normales no funcionan con este tipo de arrays.

20 Se usan bucles FOR-OF y/o FOR-IN como se enseñará más adelante.

Aún así, este tipo de arrays están muy extendidos dentro de la programación web y se usan mucho.

3.6 Spread Operator en arrays bidimensionales

Como hemos visto, tanto os arrays bidimensionales, como los arrays asociativos son arrays con estructuras complejas en su interior en lugar de valores simples.

Es por eso que, si bien se puede usar el operador de propagación en ellos, este no funciona igual que en arrays simples.

En el capítulo 6 de esta parte del manual se profundiza sobre el uso de este operador en estructuras complejas como este tipo de arrays pero podemos comentar aquí que, la principal diferencia de usar este operador en este tipo de estructuras es que no hace una copia de los valores que sean a su vez arrays (de cualquier tipo), lo que hace es una nueva referencia a cada uno de esos nuevos arrays:

```
var tmc0 = new Array(12,10,11);
var tmc1 = new Array (5,0,2);
var tmc2 = new Array (10,8,10);

var original = new Array (3);
original[0] = tmc0;
original[1] = tmc1;
original[2] = tmc2;

let nueva = [...original];
console.log(nueva)
//Obtenemos: [ [ 12, 10, 11 ], [ 5, 0, 2 ], [ 10, 8, 10 ] ]
```

Sin embargo, si tocamos algún valor desde el array bidimensional nuevo, vemos como el original también se altera:

```
nueva[0][0] = 500;
console.log(original)
//[ [ 500, 10, 11 ], [ 5, 0, 2 ], [ 10, 8, 10 ] ]
```