

CONTENIDOS

3.1 STRINGS

- 3.1.1 USO DE STRINGS
- 3.1.2 COMPARACIÓN DE STRINGS
- 3.1.3 MÉTODOS DE LOS STRINGS

3.2 ARRAYS

- 3.2.1 ¿QUÉ ES UN ARRAY?
- 3.2.2 CREACIÓN DE ARRAYS

3.3 RECORRER ARRAYS

- 3.3.1 USO DEL BUCLE FOR PARA RECORRER ARRAYS
- 3.3.2 BUCLE FOR...IN
- 3.3.3 BUCLE FOR..OF

3.4 MÉTODOS DE LOS ARRAYS

- 3.4.1 TAMAÑO DEL ARRAY
- 3.4.2 SABER SI UN ELEMENTO ES UN ARRAY
- 3.4.3 MÉTODOS PARA AÑADIR ELEMENTOS
- 3.4.4 MEZCLAR ARRAYS
- 3.4.5 OBTENER Y AÑADIR SUBARRAYS
- 3.4.6 CONVERTIR ARRAY EN STRING
- 3.4.7 BÚSQUEDA DE ELEMENTOS EN UN ARRAY
- 3.4.8 MODIFICAR EL ORDEN DE LOS ELEMENTOS DE UN ARRAY
- 3.4.9 DESESTRUCTURACIÓN DE ARRAYS

3.5 ESTRUCTURAS DE TIPO SET

- 3.5.1 INTRODUCCIÓN
- 3.5.2 DECLARACIÓN Y CREACIÓN DE CONJUNTOS
- 3.5.3 MÉTODOS DE LOS CONJUNTOS
- 3.5.4 CONVERTIR CONJUNTOS EN ARRAYS
- 3.5.5 RECORRER CONJUNTOS

3.6 MAPAS

- 3.6.1 ¿QUÉ SON LOS MAPAS?
- 3.6.2 DECLARAR MAPAS
- 3.6.3 ASIGNAR VALORES A MAPAS
- 3.6.4 OPERACIONES SOBRE MAPAS
- 3.6.5 CONVERTIR MAPAS EN ARRAYS
- 3.6.6 RECORRER MAPAS

3.1 STRINGS

3.1.1 USO DE STRINGS

Los valores de tipo string tienen que delimitarse con comillas, bien sean dobles ("), simples (') o invertidas (`). Es posible que los strings contengan comillas como carácter literal, siempre que no coincidan con el delimitador del string. Ejemplos:

```
let s1="Miguel dijo 'venid' en voz baja";  
let s2='Miguel dijo \'venid\' en voz baja';
```

Se pueden escapar caracteres para representar caracteres no visibles, que no están en el teclado o que sean problemáticos:

```
let s3="Primera 1 línea\nSegunda línea";
```

Dentro de valores de string delimitados por comillas invertidas (` `) se pueden colocar directamente expresiones del lenguaje siempre que se coloquen dentro de los símbolos \${}. Ejemplo:

```
let x=8, y=9;  
console.log(`La suma de x e y es: ${x+y}`);  
//Escribe: La suma de x e y es: 17
```

3.1.2 COMPARACIÓN DE STRINGS

Los textos se pueden comparar igual que los números.

```
let texto1="ana";  
let texto2="Ana";  
if(texto1==texto2)  
    console.log("son iguales");  
else  
    console.log("son distintos");
```

Por pantalla aparece el texto son distintos porque se distingue entre mayúsculas y minúsculas.

Al comparar qué texto es mayor, se usa el orden que tienen los caracteres en la tabla Unicode.

Así **casa** es menor que **Caso**. Pero, hay que tener en cuenta que las minúsculas, en la tabla Unicode, son mayores que las mayúsculas y por eso JavaScript considera que casa es mayor que Caso.

3.1.2.1 MÉTODO LOCALECOMPARE

Es un método que permite comparar sin discriminar entre mayúsculas y minúsculas, y considerando la forma de ordenar de cada lengua. Este método recibe como parámetro el texto con el que deseamos comparar. Podemos indicar un segundo parámetro que es el código oficial de idioma. Sin ese segundo parámetro se utiliza la configuración local del equipo que ejecuta el código.

Este método *devuelve un número negativo si el segundo texto es mayor que el primero, un número positivo si es el primer texto el que es mayor en orden alfabético, y cero si los dos textos son iguales.*

```
let texto3="Oso";  
let texto4="Ñu";  
console.log(texto3.localeCompare(texto4));  
console.log(texto3.localeCompare(texto4,"es"));
```

3.1.3 MÉTODOS DE LOS STRINGS

3.1.3.1 TAMAÑO DEL STRING

La propiedad ***length*** devuelve el tamaño de un texto:

```
let texto="Nabucodonosor";  
console.log(texto.length);
```

3.1.3.2 OBTENER UN CARÁCTER CONCRETO

El método ***charAt*** permite extraer un carácter concreto del texto. Basta con indicar la posición del carácter deseado, teniendo en cuenta que el primer carácter es el número cero, el siguiente el uno y así sucesivamente. Ejemplo:

```
let texto="Nabucodonosor";  
console.log(texto.charAt(5));
```

Hay otro método llamado ***charCodeAt*** que funciona igual, pero devuelve el código del carácter de esa posición:

```
let texto="Nabucodonosor";  
console.log(texto.charCodeAt(5));
```

Escribe el valor **99** que es el código Unicode de la letra *o*.

3.1.3.3 CONVERTIR A MAYÚSCULAS Y MINÚSCULAS

Lo realizan los métodos (sin argumentos) ***toUpperCase***, para pasar un texto a mayúsculas, y ***toLowerCase***, para minúsculas.

```
let texto="En el año 2019 saqué el carnet de Camión";  
console.log(texto.toUpperCase());
```

Para evitar problemas debido a la forma específica de pasar a mayúsculas de algunas lenguas, disponemos de las funciones ***toLocaleUpperCase*** (pasar a mayúsculas) y ***toLocaleLowerCase*** (pasar a minúsculas). Se puede indicar un segundo parámetro con el código específico de país (es, fr, en, etc.) pero, hoy en día, las funciones habituales (***toUpperCase*** y ***toLowerCase***) funcionan correctamente en casi cualquier lengua.

3.1.3.4 MÉTODOS DE BÚSQUEDA DE CARACTERES

indexOf

Sintaxis:

```
texto.indexOf(texto [, inicio])
```

Devuelve la posición del texto indicado en la variable, empezando a buscar desde el lado derecho. Si aparece varias veces, se devuelve la primera posición en la que aparece. El segundo parámetro (inicio) es opcional y nos permite empezar a buscar desde una posición concreta. Ejemplo:

```
var var1="Dónde esta la x, busca, busca";  
document.write(var1.indexOf("x"));  
//Escribe 14, posición del carácter "x"
```

En el caso de que el texto no se encuentre, devuelve -1

lastIndexOf

Igual que el método anterior, pero, en este caso, se devuelve la última posición en la que aparece el texto (o menos uno, si no se encuentra).

```
Itexto. lastIndexOf (texto f. iniciol )
```

endsWith

```
itexto.endsWith(textoABuscar[, tamaño])
```

Este método devuelve verdadero si el texto finaliza con el texto que indiquemos en el parámetro **textoABuscar**.

```
var texto="Esto es una estructura estática";  
console.log( texto. endsWith ("estática "  ));
```

Escribe true, porque efectivamente el texto termina con la palabra *estática*.

El parámetro tamaño solo mira si termina el string con el texto indicado considerando que el tamaño del texto es el indicado:

```
var texto="Esto es una estructura estática";  
console.log(texto.endsWith("estructura",22));
```

El nuevo código vuelve a escribir true, porque los primeros 22 caracteres del texto terminan con el texto estructura.

startsWith

Es muy parecido al anterior, solo que este método busca si el string comienza con el texto indicado. Si es así, devuelve true. El parámetro posición mira si el texto comienza por el indicado, pero empezando a mirar desde la posición indicada:

```
var texto="Esto es una estructura estática";
console.log(texto.startsWith("Esto")); //Escribe true
console.log(texto.startsWith( "es", 5)); //También escribe true
```

3.1.3.5 EXTRAER Y MODIFICAR SUBCADENAS

replace

texto.replace(textoBuscado,textoReemplazo)

Busca en el string el texto indicado en el primer parámetro (textoBuscado) y lo cambia por el segundo texto (textoReemplazo). Ejemplo:

```
var texto="Esto es una estructura estática";
console . log( texto . replace( "st", "xxt"));
```

Escribe:

```
Exxtto es una estructura estática
```

trim

Es un método que no necesita argumentos. Se encarga de quitar los espacios en blanco a derecha y a izquierda del texto. Ejemplo:

```
let texto="          texto con muchos espacios
console. log( "-'+texto+'-");
console.log("-"+texto.trim()+"-");
```

Es un método muy útil para ser usado cuando se realizan lecturas de datos sobre los usuarios. Es habitual necesitar que se eliminen los espacios a izquierda y derecha que, inadvertidamente, escriben los usuarios.

Slice

```
texto.slice(inicio [,fin])
```

Extrae del texto los caracteres desde la posición indicada por inicio, hasta la posición fin (sin incluir esta posición). Si no se indica fin, se toma desde el inicio hasta el final. Ejemplo:

```
let texto="Esto es una estructura estática";
console.log(texto.slice(3,10));
```

El resultado obtiene desde el carácter de la posición número 3 (en realidad el cuarto) hasta el índice anterior al 10 (coge hasta el décimo carácter). En definitiva, escribe: **o es un**

El parámetro fin puede usar números negativos, en ese caso, los números indican la posición contando desde el final del texto. Ejemplo:

```
let texto="Esto es una estructura estática";
console.log(texto.slice(3,-5));
```

Escribe por consola: **o es una estructura est**

Substring

texto.substring(inicio[, fin])

Funciona exactamente igual que slice anterior, pero no admite usar números negativos

substr

texto.substr(inicio [, tamaño])

Extrae del texto el número de caracteres indicados por el parámetro tamaño, desde la posición indicada por el número asociado al carácter inicio. Si no se indica tamaño alguno, se extraen los caracteres desde la posición indicada por inicio hasta el final. Ejemplo:

```
let texto="Esto es una estructura estática";  
console.log( texto. substr(3,10));
```

split

Es un método capaz de dividir el texto en un array de textos

texto.split ([textoDelim, límite])

Sin indicar parámetro alguno, se genera un array donde cada elemento del array será cada carácter del string. Si se indica el parámetro textoDelim, este se usa como texto delimitador; es decir, se divide el texto en trozos separados por ese delimitador. Ejemplo:

```
let texto="Esto es una estructura estática";  
console.log(texto.split(" "));
```

Escribe: ['Esto', 'es', 'una', 'estructura', 'estática']

El segundo parámetro pone un límite tope de divisiones. Por ejemplo:

```
var texto="Esto es una estructura estática";  
console.log(texto.split(" ",3));
```

Escribe: [' Esto', 'es', 'una']

El texto delimitador puede ser una expresión regular en lugar de un texto, eso permite dividir el texto en base a criterios más complejos.

3.1.3.6 CONVERTIR CÓDIGO A TEXTO

El método **fromCharCode** permite que indiquemos una serie de números de código de la tabla Unicode y nos devolverá un string formado por los caracteres correspondientes a estos códigos.

```
console.log(String.fromCharCode(65,66,67));
```

El código anterior escribe el texto ABC, porque 65 es el código de la letra A, 66 el de la B y 67 el de la C.

3.2 ARRAYS

3.2.1 ¿QUÉ ES UN ARRAY?

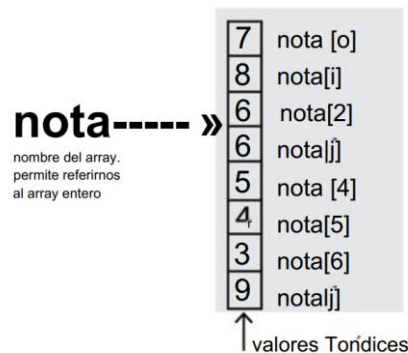
Todos los lenguajes de programación disponen de un tipo de variable que es capaz de manejar conjuntos de datos. A este tipo de estructuras se las llama arrays de datos. También se las llama listas, vectores o arreglos, pero todos estos nombres tienen connotaciones que hacen que se puedan confundir con otras estructuras de datos. Por ello, es más popular el nombre sin traducir al castellano: array.

Los arrays son variables que permiten almacenar, usando una misma estructura, una serie de valores.

En JavaScript los arrays son objetos.

Los arrays de JavaScript son totalmente dinámicos, su tamaño se puede modificar después de la declaración a voluntad.

En JavaScript los arrays son heterogéneos. Admite, por ejemplo, que un elemento sea un número y otro un string. Cada elemento del array puede ser de un tipo distinto.



3.2.2 CREACIÓN DE ARRAYS

DECLARACIÓN Y ASIGNACIÓN DE VALORES

Hay muchas formas de declarar un array. Por ejemplo, si deseamos declarar un array vacío, la forma habitual es:

```
let a= [];
```

Los corchetes vacíos tras la asignación, significan array vacío. Un array que se llama simplemente **a** y que está listo para indicar valores.

Equivalente a ese código, podemos hacer:

```
let a=new Array();
```

Para colocar valores en el array se usa el índice que indica la posición del array en la que colocamos el valor. El primer índice es el cero, por ejemplo:

```
a[0]="Pedro";
a[1]="Luis";
a[2]="Marta";
a[3]="Sofía";
```

Si quisiéramos mostrar un elemento concreto por consola haríamos:

```
console.log(a[2]); //Muestra: Marta
```

Podemos asignar valores en la propia declaración del array:

```
let nota=[7,8,6,6,5,4,3,9];
```

Disponemos también de esta otra forma de declarar:

```
let nota=new Array(7,8,6,6,5,4,3,9);
```

En este caso, la variable nota es un array de ocho elementos, todos números. Para mostrar el primer elemento:

```
console.log(nota[0]); //Muestra: 7
```

El método console.log tiene capacidad para mostrar todo un array:

```
console.log(nota); //Muestra: [7,8,6,6,5,4,3,9]
```

Sin embargo, salvo para tareas de depuración, no es conveniente este uso. Lo lógico es utilizar bucles de recorrido.

USO DE CONST Y LET CON ARRAYS

Es muy habitual declarar arrays con la palabra clave **const** en lugar de con **let**. Por ejemplo:

```
const datos=[4.5,6.78,7.12,9.123];
```

Hemos declarado un array de cuatro valores decimales. Lo hemos declarado con const, lo que, en principio, indica que el valor de la variable datos no puede variar. Sin embargo, este código es válido:

```
datos[0]=4.671;
```

Hemos añadido un nuevo elemento al array. Luego, aun declarando las variables con la palabra clave const, se ha modificado el contenido del array. La razón es que, realmente, los arrays son objetos. la variable a la que se asigna el array realmente es lo que nos sirve para llegar al valor del array, es una referencia al array, pero no es el array en sí. Digamos que lo que realmente contiene esa variable es cómo llegar al objeto.

Este código sí provoca un error:

```
const datos=[4.5, 6.78, 7.12, 9.123];  
datos=[9.18, 4.95]
```

En la primera línea declaramos datos como la forma de acceder al array que tiene valores: [4.5, 6.78, 7.12, 9.123]. Esta variable es un enlace o una referencia ese array. Pero en la segunda línea decimos que la variable datos es un enlace a otro array. Por lo tanto, ahora sí estamos modificando la referencia y eso no se permite porque la variable datos se ha declarado con la palabra **const**.

Cuando modificamos los elementos del array, o eliminamos elementos o incluso cuando los añadimos, el array sigue siendo el mismo. La referencia no cambia.

OPERACIÓN DE ASIGNACIÓN CON ARRAYS

En este ejemplo:

```
const datos = [4.5, 6.78, 7.12, 9.123];  
const datos2 = datos;  
datos2[0] = 400;  
console.log(datos[0]); // escribe 400
```

Cuando asignamos la variable *datos2* al array *datos*, no se hace una copia del array. Tanto *datos* como *datos2* son dos variables que hacen referencia al mismo array. Por eso, cuando cambiamos el primer elemento del array *datos2*, también cambiamos el del array *datos*, ya que en realidad es el mismo array.

VALORES INDEFINIDOS

Veamos este código:

```
let a=["Saúl","Rocío"];  
a[3]="María";  
console.log(a[2]); //Escribe undefined
```

Estamos creando el array llamado *a* y dando valor a los dos primeros elementos (*a*[0], *a*[1]). Luego, damos valor al cuarto elemento del array (*a*[3] = "María"). En ningún momento hemos dado valor al elemento *a*[2] y por eso, el código provoca la escritura el texto *undefined*. El resumen es que podemos dejar elementos sin definir en un array. Es más, incluso en la propia declaración se pueden dejar elementos vacíos.

ELIMINAR ELEMENTOS DE UN ARRAY

Para borrar un elemento se utiliza la palabra **delete** tras la cual se indica el elemento a eliminar:

```
const dias=["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"];  
delete dias[2];  
console.log(dias);
```

Se borra el miércoles (aparece el texto "<empty ítem>" en su lugar). Este elemento pasa a ser indefinido.

ARRAYS HETEROGÉNEOS

Los arrays de JavaScript son heterogéneos. Admiten mezclar en el mismo array valores de diferente tipo:

```
const a=[3,4,"Hola",true, Math.random()];
```

Incluso podemos definir arrays de este tipo:

```
const b= [3,4, "Hola" , [99,55,33]];
```

El cuarto elemento (b[3]) es un array. Es decir, se pueden colocar arrays dentro de otros arrays. Es más, los elementos de un array pueden ser de cualquier tipo, hay arrays de todo tipo de objetos. De modo que esta instrucción es perfectamente posible:

```
console.log(b[3][1]); //Escribe 55
```

En el ejemplo anterior, b[3][1] hace referencia al segundo elemento del cuarto elemento de b. En definitiva, las posibilidades de uso de arrays en JavaScript para almacenar datos son espectaculares.

3.3 RECORRER ARRAYS

3.3.1 USO DEL BUCLE FOR PARA RECORRER

Una de las tareas fundamentales, en la manipulación de arrays, es recorrer cada elemento de un array para poderlo examinar. Esto es fácil de hacer mediante el bucle for:

```
const notas = [5,6,7,4,9,8,9,9,7,8];
for (let i=0; i <notas.length; i++) {
  console.log('La nota ${i} es ${notas[i]}');
}
```

En el bucle se utiliza el método **length** que nos permite saber el tamaño de un array.

El problema es cuando hay elementos indefinidos en el array:

```
const notas=[5,6, ,,,9,,,8,,9,,7,8];
for (let i=0;i<notas.length;i++){
  console.log('La nota ${i} es ${notas[i]}');
}
```

Ahora el array de notas contiene muchos elementos indefinidos. Si se deja así, el código mostrará el texto **undefined** para el elemento 2,el 3,.. Lo lógico es evitar los elementos indefinidos. Por lo que el código debería modificarse de esta forma:

```
const notas=[5,6, ,,,9,,,8,,9,,7,8];
for (let i=0;i<notas.length;i++) {
  if(notas[i]!==undefined){
    console.log('La nota ${i} es ${notas[i]}');
  }
}
```

El bloque if permite comprobar primero que el elemento está definido y, solo si es así se muestra.

3.3.2 BUCLE FOR...IN

JavaScript posee un bucle mucho más cómodo para recorrer arrays. Se trata de un bucle for especial llamado **for..in**. La ventaja es que no necesita tanto código, basta con indicar el nombre del contador y el array que se recorre. Su sintaxis general es la siguiente:

```
for(let índice in nombreArray){
  instrucciones
}
```

El **índice** es el nombre de una variable que se declara en el propio **for**, y que irá tomando todos los valores del índice del array que recorre. Esa variable *se salta los elementos indefinidos, solo recorre los definidos*. Por lo que el bucle **for..in** que nos permite recorrer las notas, saltándonos las indefinidas, es:

```
const notas=[5,6,,,,9,,,8,,9,,7,8];  
for(let i in notas){  
  console.log('La nota ${i} es ${notas[i]}');  
} // se saltaría los indefinidos
```

3.3.3 BUCLE FOR..OF

En este tipo de bucle, la variable que se crea en el bucle va almacenando los diferentes valores del array (en lugar de los índices como hacía **for..in**). Ejemplo:

```
const notas=[5,6,,,,9,,,8,,9,,7,8];  
for (let nota of notas){  
  console.log(nota);  
}
```

for..of no se salta los elementos indefinidos. Nuevamente, necesitamos usar la instrucción **if** para poder saltar los elementos indefinidos **for..of** no se salta los elementos indefinidos.

```
for(let nota of notas){  
  if(nota!==undefined){  
    console.log(nota);  
  }  
}
```

3.4 MÉTODOS DE LOS ARRAYS

3.4.1 TAMAÑO DEL ARRAY

Propiedad **length** permite conocer el tamaño del array.

```
const dias=["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"];  
console.log(dias.length); //Escribe 7
```

Es fácil, gracias a este método, añadir un elemento al final de un array:

```
nota[nota.length]=8;
```

De esta forma, añadimos el número 18 al final del array **nota**.

3.4.2 SABER SI UN ELEMENTO ES UN ARRAY

JavaScript aporta un operador llamado **instanceof**. Este método, en realidad, sirve para poder comprobar si un objeto pertenece a una determinada clase. Los arrays pertenecen a la clase **Array**. Saber si una variable concreta es de clase **Array** es, simplemente, valorar con **instanceof** si pertenece a la clase **Array**. Ejemplo:

```
let a= [1,2,3,4,5,6,7,8,9];  
let b="Hola";  
console.log(a instanceof Array); //Escribe true  
console.log(b instanceof Array); //Escribe false
```

3.4.3 MÉTODOS PARA AÑADIR ELEMENTOS

push: inserta un elemento al final del array. Hay que indicar el valor del elemento. Ejemplo:

```
const cantantes=["Simón"];  
cantantes.push("Garfunkel"); // ahora cantantes tendrá 2 elementos.
```

pop: retira un elemento del array que se puede asignar a una variable. Ejemplo:

```
const cantantes2=["Peter","Paúl","Mary"];  
let x=cantantes2.pop();  
console.log(`El array ha quedado así: ${cantantes2}`);  
console.log(`La variable x vale ${x}`);
```

shift: quita un elemento por delante del array. Ejemplo:

```
const cantantes3=["Crosby","Stilis","Nash"];  
let x=cantantes3.shift(); // elimina a Crosby del array y se almacena en x  
console.log(`El array ha quedado: ${cantantes3}`);  
console.log(`La variable x vale ${x}`);
```

unshift: inserta un elemento por delante del array. Hay que indicar como parámetro el valor a añadir por el lado izquierdo:

```
const cantantes4=["Crosby","Stilis","Nash"];  
cantantes4.unshift("Young"); //inserta "Young" como primer elemento del array  
console.log(`El array ha quedado: ${cantantes4}`);
```

3.4.4 MEZCLAR ARRAYS

concat : para añadir los elementos de un array a otro. El resultado de este método es un nuevo array, no se modifica ninguno de los dos arrays originales, que contiene los elementos unidos de ambos arrays. Por eso, el resultado de concat debe de ser asignado a un nuevo array. Ejemplo:

```
const planetas1=["Venus","Mercurio","La Tierra","Marte"];  
const planetas2=["Júpiter","Saturno","Urano","Neptuno"];  
const planetas=planetas1.concat(planetas2);  
console.log(planetas); // El array planetas contiene a todos los planetas, tanto los del primer array como los del segundo.
```

3.4.5 OBTENER Y AÑADIR SUBARRAYS

slice : permite copiar una serie de elementos de un array indicando el índice del primer elemento que deseamos copiar y el índice final. El resultado es un nuevo array.

```
const dias=["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"];  
const tresDias=dias.slice(3,6); // se copian los elementos 3, 4 y 5 (6 no incluido) en nuevo array  
console.log(tresDias);
```

Si no se pone el segundo parámetro en slice se extrae una copia hasta el final.

El propio método **slice** nos permite hacer una copia completa de un array. Para ello, basta que se use sin parámetros.

let array2=array1.slice(); //array2 es una copia de array1

splice: para eliminar definitivamente los elementos de un array, también lo podemos utilizar para añadir elementos. Para ello, se indica como primer parámetro, desde dónde iniciamos la eliminación y cuántos elementos eliminamos. Este método modifica el array original y devuelve otro array que contiene los elementos eliminados.

```
const castillaLaMancha= ["Guadalajara", "Sevilla", "Granada", "Ciudad Real", "Albacete"];  
const borrados=castillaLaMancha.splice(1,2);  
console.log(`Se han borrado los elementos: ${borrados}`); //contendrá Sevilla y Granada  
console.log(`Ahora quedan: ${castillaLaMancha}`);
```

Además de eliminar elementos, los podemos cambiar por la serie de elementos que indiquemos:

```
const castillaLaMancha= ["Guadalajara", "Sevilla", "Granada", "Ciudad Real", "Albacete"];  
castillaLaMancha.splice(1,2,"Toledo", "Cuenca");  
console.log(castillaLaMancha);
```

El número de elementos que se añaden no tiene por qué coincidir con los que se quitan.

3.4.6 CONVERTIR ARRAY EN STRING

join: permite convertir un array en un string. El string resultante contiene todos los elementos del array.

```
const dias=["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"];  
console.log(dias.join());
```

Cada elemento se separa con una coma. Es posible indicar otro separador ya que se puede indicar un texto como parámetro de join para que sea utilizado como separador:

```
const dias=["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"];  
console.log(dias.join(""));  
console.log(dias.join(";");  
console.log(dias.join("<->"));
```

3.4.7 BÚSQUEDA DE ELEMENTOS EN UN ARRAY

indexOf: permite buscar el valor elemento en un array. Si lo encuentra devuelve su posición y si no, devuelve el valor -1. Ejemplo:

```
const dias=["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"];  
console.log(dias.indexOf('Jueves')); // escribe valor 3
```

Si hubiera dos elementos con ese valor, devuelve el primero empezando con la izquierda. También podemos indicar el inicio de la búsqueda, por ejemplo:

dias.indexOf("Jueves",4);

Busca la palabra Jueves a partir del elemento con índice 4 (en este caso no lo encontrará y devolverá -1).

lastIndexOf: método idéntico al anterior, solo que empieza a buscar desde el último elemento. Ejemplo:

const numeros=[2,3,5,6,4,3,7,8,4,3,2,8,3,2];
console.log(numeros.lastIndexOf(3));

Este código escribe el número 12, que es la última posición en la que aparece el valor 3 en el array. Al igual que ocurría con lastIndexOf, podemos empezar a buscar desde una posición concreta, pero, en este caso, se busca hacia la izquierda desde esa posición. Ejemplo:

const números=[2,3,5,6,4,3,7,8,4,3,2,8,3,2];
console.log(numeros.lastIndexOf (3,6));

Escribe el número 5, porque empezando a buscar desde la posición 6 hacia la izquierda, el primer valor que tiene un tres es el elemento con índice 5.

includes: permite buscar un elemento y devuelve **true** si ese valor está en el array y **false** si no lo está.

const dias=["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"];
console.log(dias.includes('Jueves')); //escribe true

3.4.8 MODIFICAR EL ORDEN DE LOS ELEMENTOS DE UN ARRAY

reverse() : El método reverse es capaz de dar la vuelta a los elementos de un array. No devuelve un nuevo array, modifica el original.

const dias=["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"];
dias.reverse();
console.log(dias);

sort() : Permite ordenar los elementos de un array. Ejemplo:

const dias=["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"];
dias.sort();
console.log(dias);

Es una función de fácil uso pero tiene el problema habitual de no saber ordenar con las características nacionales y de considerar que las minúsculas son mayores que las mayúsculas. Tiene en cuenta para ordenar la posición de los caracteres en la tabla Unicode. La manera de solventar este problema es gracias a que el método sort admite indicar una función de tipo callback que nos permita personalizar la ordenación. En el tema de funciones veremos cómo usarla.

3.4.9 DESESTRUCTURACIÓN DE ARRAYS

No afecta solo a los arrays, pero en este apartado nos centraremos en la sintaxis de desestructuración relacionada con los arrays. *Una primera capacidad que aporta es la de*

asignar valores a varias variables a la vez. En este ejemplo se asignan de golpe valores para tres variables:

```
let [saludo, despedida, cierre]=["Hola", "Adiós", "Hasta nunca"];  
console.log(saludo); //Escribe Hola  
console.log(despedida); //Escribe Adiós  
console.log(cierre); //Escribe Hasta nunca
```

Otra capacidad es facilitar la operación clásica de **swap de variables**. Es decir, intercambiar los valores de dos variables:

```
[a,b]=[b,a]; // La variable a valdrá lo que valía b y viceversa.
```

Hay también un operador llamado **operador de propagación (spread en inglés)** que permite convertir un array en variables y viceversa. Este operador se usa con tres puntos seguidos (...) Ejemplo:

```
let array=[1,2,3];  
let [x,y,z]=[...array]; //La variable x valdrá uno, y valdrá dos y z valdrá tres.
```

Incluso es válido este código:

```
let array=[1,2,3];  
let [x,y]=[...array]; // Ahora x vale uno e y vale dos.
```

También es válida esta sintaxis:

```
let a,b,array;  
[a,b,...array] = [1,2,3,4,5]; // la variable a valdrá 1, b valdrá 2 y array será efectivamente un array con tres elementos que valdrán 3, 4 y 5 respectivamente.
```

3.5 ESTRUCTURAS DE TIPO SET

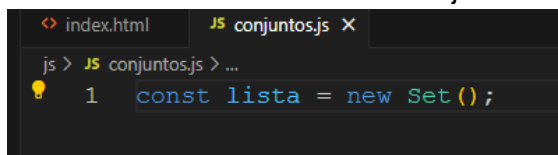
3.5.1 INTRODUCCIÓN

Los Sets (que se pueden traducir con conjuntos) son una estructura de datos (son objetos realmente) que permiten, de forma similar a los arrays, almacenar datos.

A diferencia de los arrays, no admiten valores duplicados y esa es su virtud. Es muy habitual recoger datos, pero eliminando aquellos que ya están repetidos. En los arrays esta tarea es pesada de realizar, pero con los conjuntos es muy sencilla ya que no hay que implementar nada.

3.5.2 DECLARACIÓN Y CREACIÓN DE CONJUNTOS

Podemos declarar e iniciar un conjunto vacío de valores:



```
index.html JS conjuntos.js X  
js > JS conjuntos.js > ...  
1 const lista = new Set();
```

Para añadir nuevos valores al conjunto hay que hacer uso del *método add*:

```
lista.add(5);  
lista.add(10);  
lista.add(10);  
lista.add(15);  
console.log(lista);
```

```
PS C:\Users\Jose Antonio\Desktop  
Set(3) { 5, 10, 15 }
```

Observando la salida podemos ver que los duplicados no están

Podemos también iniciar la lista a partir de un array:

```
1 const lista = new Set([1,2,5,5,2,10]);  
2 console.log(lista);
```

```
PS C:\Users\Jose Antonio\Desktop  
Set(4) { 1, 2, 5, 10 }
```

Es posible también iniciar el conjunto con un texto:

```
1 const lista = new Set("Conjunto");  
2 console.log(lista);
```

Cada elemento del conjunto es una letra y, como siempre, no se repiten los mismos caracteres.

```
PS C:\Users\Jose Antonio\Desktop\pruebas\js  
Set(6) { 'C', 'o', 'n', 'j', 'u', 't' }
```

Para añadir strings completos hay que utilizar el método add:

```
const lista = new Set();  
lista.add("Conjunto")  
console.log(lista);
```

```
PS C:\Users\Jose Antonio\Desktop  
Set(1) { 'Conjunto' }
```

3.5.3 MÉTODOS DE LOS CONJUNTOS

3.5.3.1 TAMAÑO DEL CONJUNTO

La *propiedad* **size** permite saber el tamaño de un conjunto:

```
const lista = new Set([2,6,8,10]);  
console.log(lista.size);
```

```
PS C:\Users\Jose Antonio\Desktop  
4
```


3.5.3.2 ELIMINAR VALORES

El método ***delete*** elimina el valor indicado de un conjunto.

El método ***clear*** elimina todo el conjunto.

```
const lista = new Set([2,6,8,10]);  
  
console.log(lista);  
lista.delete(8);  
console.log(lista);  
lista.clear();  
console.log(lista);
```

```
PS C:\Users\Jose Antonio\De  
Set(4) { 2, 6, 8, 10 }  
Set(3) { 2, 6, 10 }  
Set(0) {}  
PS C:\Users\Jose Antonio\De
```

A la salida se observa que el número 8 ha sido eliminado y después con clear se elimina todo el conjunto.

3.5.3.3 BUSCAR VALORES

El método ***has*** permite que se le indique un valor y devuelve ***true*** si dicho valor forma parte del conjunto:

```
const lista = new Set([2,6,8,10]);  
  
console.log(lista.has(6));  
console.log(lista.has(9));
```

```
PS C:\Users\Jose Antonio\De  
true  
false  
PS C:\Users\Jose Antonio\De
```

El método has devuelve true porque encuentra el número 6 en el conjunto y false al no encontrar el número 9.

3.5.4 CONVERTIR CONJUNTOS EN ARRAYS

JavaScript posee un operador muy interesante conocido como operador de propagación (en inglés se llama *operador spread*). Este operador usa tres puntos seguidos detrás de los cuales podemos indicar el conjunto a convertir.

```
const lista = new Set([2,6,8,10]);  
const arrayNum = [...lista];  
  
console.log(lista);  
console.log(arrayNum);  
const nuevaLista = new Set([...arrayNum]);  
console.log(nuevaLista);
```

```
PS C:\Users\Jose Antonio\De  
Set(4) { 2, 6, 8, 10 }  
[ 2, 6, 8, 10 ]  
Set(4) { 2, 6, 8, 10 }  
PS C:\Users\Jose Antonio\De
```

A la salida vemos el conjunto lista inicial, después se muestra el arrayNum construido a partir del conjunto. La última línea podemos ver que un array lo podemos volver a convertir en conjunto teniendo en cuenta que si hubiese elementos repetidos en el array no pasarían a formar parte del conjunto.

3.5.5 RECORRER CONJUNTOS

Al igual que ocurre con los arrays, es habitual necesitar recorrer cada elemento de un conjunto. Para ello, disponemos de los bucles `for..of` que recorren cada elemento del conjunto:

```
const lista = new Set([2,6,8,10]);  
⚡  
console.log(lista);  
for (let numero of lista){  
  console.log(numero);  
}
```

```
PS C:\Users\Jose Antonio\De  
Set(4) { 2, 6, 8, 10 }  
2  
6  
8  
10
```

La variable *numero* que se declara antes de la palabra **of** se va actualizando en cada iteración con el elemento correspondiente de la lista o conjunto.

3.6 MAPAS

3.6.1 ¿QUÉ SON LOS MAPAS?

Los mapas permiten, en JavaScript, crear estructuras de tipo clave-valor, en las cuales las claves no se pueden repetir y tienen asociado el valor. Tanto las claves como los valores pueden ser de cualquier tipo. En un mismo mapa no puede haber dos elementos con la misma clave, pero sí pueden repetir valor.

3.6.2 DECLARAR MAPAS

Como los arrays y los conjuntos, realmente los mapas son objetos. Para declarar un mapa e iniciarlo sin contenido lo hacemos como en el siguiente código:

```
const provincias = new Map();
```

3.6.3 ASIGNAR VALORES A MAPAS

3.6.3.1 MÉTODO SET

El método `set` es el que permite asignar nuevos elementos. Este método requiere la clave del nuevo elemento y después el valor con el que asociamos dicha clave:

```
const provincias = new Map();

provincias.set(1, "Alava");
provincias.set(28, "Madrid");
provincias.set(30, "Murcia");
provincias.set(41, "Sevilla");
console.log(provincias);
```

```
PS C:\Users\Jose Antonio\Desktop> node conjuntos.js
Map(4) {
  1 => 'Alava',
  28 => 'Madrid',
  30 => 'Murcia',
  41 => 'Sevilla'
}
```

Si volvemos a añadir un elemento con la misma clave, este sustituye al anterior ya que no puede haber claves repetidas.

3.6.3.2 USO DE ARRAYS PARA CREAR MAPAS

También podemos utilizar un array para crear un mapa, donde cada elemento del array es otro array, en el que el primer elemento es la clave y el segundo el valor de esa clave.

```
const alumnos = new Map([[1, "Jose"], [2, "Alvaro"], [3, "Joaquín"], [4, "María"]]);

console.log(alumnos);
```

```
PS C:\Users\Jose Antonio\Desktop\pruebas\js> node conjuntos.js
Map(4) { 1 => 'Jose', 2 => 'Alvaro', 3 => 'Joaquin', 4 => 'Maria' }
```

3.6.4 OPERACIONES SOBRE MAPAS

3.6.4.1 OBTENER VALORES DE UN MAPA

En los mapas es posible obtener el valor de una clave con el *método* **get** al que se le indica la clave del elemento que queremos obtener.

```
const alumnos = new Map([[1, "Jose"], [2, "Alvaro"], [3, "Joaquín"], [4, "María"]]);

let alumno = alumnos.get(3);
console.log(alumnos);
console.log(alumno);
```

```
PS C:\Users\Jose Antonio\Desktop\pruebas\js> node conjuntos.js
Map(4) { 1 => 'Jose', 2 => 'Alvaro', 3 => 'Joaquin', 4 => 'Maria' }
Joaquín
```

3.6.4.2 BUSCAR UNA CLAVE EN UN MAPA

El método **has** permite buscar una clave en un mapa. Si la encuentra, devuelve true y si no, devuelve false.

```
const alumnos = new Map([[1, "Jose"], [2, "Alvaro"], [3, "Joaquín"], [4, "María"]]);

console.log(alumnos.has(4));
console.log(alumnos.has("Jose"));
```

```
PS C:\Users\Jose Antonio\I
true
false
```

A la salida tenemos **true** ya que existe un elemento con clave 4 cuyo valor es María. Al buscar “Jose” devuelve false porque no existe en el mapa una clave que sea “Jose”, aunque si como valor.

3.6.4.3 BORRAR VALORES

El método delete permite eliminar un elemento del mapa. Para ello, hay que indicar la clave de dicho elemento:

```
const alumnos = new Map([[1, "Jose"], [2, "Alvaro"], [3, "Joaquín"], [4, "María"]]);

console.log(alumnos);
alumnos.delete(1);
console.log(alumnos);
```

```
PS C:\Users\Jose Antonio\Desktop\pruebas\js> node conjuntos.js
Map(4) { 1 => 'Jose', 2 => 'Alvaro', 3 => 'Joaquín', 4 => 'María' }
Map(3) { 2 => 'Alvaro', 3 => 'Joaquín', 4 => 'María' }
```

En la salida observamos que el elemento con clave 1 y valor “Jose” ha sido eliminado.

3.6.4.4 OBTENER OBJETOS ITERABLES

Un objeto iterable es un tipo de objeto semejante a un array, ya que es una colección de valores que se pueden recorrer mediante bucles de tipo for...of. Sin embargo, los objetos iterables no son realmente arrays y, por lo tanto, se manipulan de forma distinta.

Los mapas permite crear objetos iterables que contienen solo las claves y objetos iterables solo con los valores. El método **keys** obtiene las claves y el método **values** obtiene los valores.

```
const alumnos = new Map([[1, "Jose"], [2, "Alvaro"], [3, "Joaquín"], [4, "María"]]);

console.log(alumnos);
let claves = alumnos.keys();
let valores = alumnos.values();
console.log("--CLAVES----");
for (let clave of claves){
  console.log(clave);
}
console.log("--VALORES----");
for (let valor of valores){
  console.log(valor);
}
```

```
PS C:\Users\Jose Antonio\Desktop\pruebas\js> node conjuntos.js
Map(4) { 1 => 'Jose', 2 => 'Alvaro', 3 => 'Joaquín', 4 => 'María' }
--CLAVES----
1
2
3
4
--VALORES----
Jose
Alvaro
Joaquín
María
```

3.6.5 CONVERTIR MAPAS EN ARRAYS

Al igual que ocurre con los conjuntos, el operador de propagación de JavaScript permite convertir mapas en array.

```
const alumnos = new Map([[1,"Jose"],[2,"Alvaro"],[3,"Joaquín"],[4,"María"]]);

console.log(alumnos);
const arrayAlum = [...alumnos];
console.log(arrayAlum);
```

```
PS C:\Users\Jose Antonio\Desktop\pruebas\js> node conjuntos.js
Map(4) { 1 => 'Jose', 2 => 'Alvaro', 3 => 'Joaquín', 4 => 'María' }
[ [ 1, 'Jose' ], [ 2, 'Alvaro' ], [ 3, 'Joaquín' ], [ 4, 'María' ] ]
```

3.6.6 RECORRER MAPAS

El bucle **for...of** es el ideal para recorrer el contenido de los mapas. Cada valor que recoge este bucle es un array de dos elementos: el primero es la clave y el segundo el valor. La forma de recorrer los índices de un array es la siguiente:

```
const alumnos = new Map([[1,"Jose"],[2,"Alvaro"],[3,"Joaquín"],[4,"María"]]);

console.log(alumnos);
for (let elemento of alumnos){
  console.log(elemento);
}
```

```
Map(4) { 1 => 'Jose', 2 => 'Alvaro', 3 => 'Joaquín', 4 => 'María' }
[ 1, 'Jose' ]
[ 2, 'Alvaro' ]
[ 3, 'Joaquín' ]
[ 4, 'María' ]
```

En la salida vemos que cada elemento del mapa es un array de tamaño 2 con la clave y el valor.

También podemos desestructurar el array de los elementos para separar en dos variables la clave y el valor. Este podría ser otra versión del bucle para el mismo array:

```
const alumnos = new Map([[1,"Jose"],[2,"Alvaro"],[3,"Joaquín"],[4,"María"]]);

console.log(alumnos);
for (let [clave,valor] of alumnos){
  console.log(`Clave:${clave} Valor:${valor}`);
}
```

UT3.- Estructuras de datos básicas

```
PS C:\Users\Jose Antonio\Desktop\pruebas\js> node conjuntos.js
Map(4) { 1 => 'Jose', 2 => 'Alvaro', 3 => 'Joaquín', 4 => 'María' }
Clave:1 Valor:Jose
Clave:2 Valor:Alvaro
Clave:3 Valor:Joaquín
Clave:4 Valor:María
```

Este bucle es más versátil por separar de forma más cómoda la clave y el valor.

Los *métodos* **keys** y **valúes** también nos permiten recorrer las claves y los valores por separado. Podríamos hacer algo así: