Variables

Variables. Una variable de JavaScript puede contener un valor de cualquier tipo de datos. Esta característica se denomina **escritura dinâmica**

En principio, existen dos ámbitos muy bien definidos:

- **Ámbito global**: Existe a lo largo de todo el programa o aplicación.
- **Ámbito local**: Existe sólo en una pequeña región del programa. Normalmente, esto se determina muy fácilmente observando las llaves que abren un nuevo ámbito o contexto.

Ámbitos de variables

```
let a = 1;
console.log(e); // Uncaught ReferenceError: e is not defined
if (a == 1)
{           let e = 40;
            console.log(e); // 40, existe
}
console.log(e); // Uncaught ReferenceError: e is not defined
```

La variable a existe en un ámbito global, y la variable e sólo existe en el interior del if: es un ámbito local.

Variables

Ámbitos de variables: let y const

A *partir de ECMAScript 2015* se introduce la palabra clave let en sustitución de la antigua var para declarar variables y const para declarar constantes. Tanto con let como con const, estaremos utilizando los ámbitos clásicos de programación: **ámbito global y ámbito local**.

```
console.log("Antes: ", p); // En este punto, p no está definida
for (let p = 0; p < 3; p++)
{               console.log("Valor de p: ", p); // Aquí, p estará definida como 0, 1, 2
}
console.log("Después: ", p); // En este punto, vuelve a no estar definida
```

1. Utilizando let en el bucle for, la variable p sólo está definida dentro del bucle (*ámbito local*)
2. Tanto antes como después del bucle, p no existe.

Variables

Ámbitos de variables: var (legacy)

Declarar variables con var ya se considera legacy (*obsoleto*) y no debe usarse.

```
console.log(“Antes:”, p); // En este punto, p vale undefined
for (var p = 0; p < 3; p++) {
        console.log("Valor de p: ", p); // Aquí, p estará definida como 0, 1, 2
}
console.log("Después: ", p); // Después: 3
```

Si utilizamos var la variable p sigue existiendo fuera del bucle, aunque haya sido declarada en su interior. Esto ocurre porque en ese caso se usa un ámbito a **nivel de función**.

Variables

```
var a = 1;
console.log(a); // Aquí accedemos a la "a" global, que vale 1
function x() {
        console.log(a); // En esta línea el valor de "a" es undefined
        var a = 5; // Aquí creamos una variable "a" a nivel de función
        console.log(a); // Aquí el valor de "a" es 5 (a nivel de función)
        console.log(window.a); // Aquí el valor de "a" es 1 (ámbito global)
}
x(); // Aquí se ejecuta el código de la función x()
console.log(a); // En esta línea el valor de "a" es 1
```

El valor de a dentro de una función no es el 1 inicial, sino que estamos en otro ámbito diferente donde la variable a anterior no existe: un **ámbito a nivel de función**. Mientras estemos dentro de una función, las variables inicializadas en ella estarán en el **ámbito** de la propia función.
Estamos usando el objeto especial window para acceder directamente al ámbito global independientemente de donde nos encontremos. Esto ocurre así porque las variables globales se almacenan dentro del objeto window (*la pestaña actual del navegador web*).
Hoy en día, en lugar de utilizar window sería preferible utilizar globalThis.

Variables

Constantes

`const` crea una referencia de solo lectura a un valor
Se añade en ECMAScript 2015.
Las constantes son de ámbito de bloque, al igual que las variables definidas mediante la instrucción `let`

- Las constantes no pueden ser reasignadas a un valor
- Una constante no puede ser re-declarada
- Una constante requiere un inicializador. Esto significa que las constantes deben inicializarse durante su declaración
- El valor asignado a una variable **`const`** es inmutable

```
const name = “Pedro";
console.log(name);
name = "Paco"; // Uncaught TypeError: Assignment to constant variable.
```

La diferencia respecto al let es que const asigna un valor y no permite reasignarlo o redeclararlo posteriormente. Además, tampoco puedes crear una constante sin indicarle un valor concreto.
Algunas constantes (*objetos, arrays...*) si que pueden ser alteradas, aunque nunca reasignadas.

Variables

Constantes

Si le asignamos un objeto a una constante, éste no será totalmente inmutable ya que le podremos cambiar los valores de sus propiedades y métodos. Lo mismo sucederá con un *array*, al que le podremos 'mutar' el valor de cada uno de sus elementos

```
const  OBJETO = {nombre:'Maria'};
OBJETO = {nombre:Maria',apellido:'Garcia'};
console.log(OBJETO.nombre);// No se puede
alterar la estructura de un objeto. Dará error
```

```
const OBJETO = {
   name: 'David'
}
OBJETO.name = 'Miguel';
console.log(OBJETO.name);//Miguel
```

Las propiedades del objeto en sí se pueden modificar, añadir o borrar

```
const OBJETO = {nombre:'Rosa'};
OBJETO.apellido = 'Garcia';
console.log(OBJETO.nombre +' '+OBJETO.apellido);
```

La constante es el propio objeto, no sus propiedades. El objeto en sí está con mayúsculas pero sus propiedades no, porque no son constantes.

Variables

Constantes

Otra excepción son los arrays, ya que podremos añadir y eliminar elementos a un array definido con `const`

```
const MISaLUMNOS = ['Antonio', 'Luis'];
MISaLUMNOS.push ('Pedro');
console.log (MISaLUMNOS);
```

Podemos **congelar** un objeto o array utilizando el método `freeze` que no nos permitirá cambiar ni propiedades ni valores

```
const OBJETO = {nombre:'Rosa'};
const NUEVOoBJETO = Object.freeze(OBJETO);
NUEVOoBJETO.apellido = 'Garcia';
console.log(NUEVOoBJETO.nombre) ;
```

Arrays

¿Qué es un array?

Un es una colección o agrupación de elementos en una misma variable, cada uno de ellos ubicado por la posición que ocupa en el array. Se pueden definir de varias formas:

```
// Forma tradicional (no se suele usar en Javascript)
const letters = new Array("a", "b", "c"); // Array con 3 elementos
const letters = new Array(3); // Array vacío de tamaño 3
// Mediante literales (notación preferida)
const letters = ["a", "b", "c"]; // Array con 3 elementos
const letters = []; // Array vacío (0 elementos)
const letters = ["a", 5, true]; // Array mixto (String, Number, Boolean)
```

Javascript permite que se puedan realizar arrays de **tipo mixto** (elementos de diferente tipo).

El operador []

Sirve para acceder a alementos del array.

La propiedad .length no dice el tamaño del array

```
const letters = ["a", "b", "c"];
letters.length; // 3
letters[0]; // 'a'
letters[5]; // undefined
```

Arrays

Para modificar elementos del array:
letters[1] = "Z"; // modifica letters a ["a", "Z", "c"]
Además del operador [], podemos utilizar el método .at(), añadido en ES2022 . Con él, se puede hacer exactamente lo mismo que con [pos], sólo que además permite valores negativos, mediante los cuales se puede obtener elementos en orden inverso, es decir, empezando a contar desde el último elemento:

```
const letters = ["a", "b", "c"];
letters.at(0); // "a"
letters.at(-1); // "c"
```

Añadir o eliminar elementos

- Los métodos .push() y .pop() actuan al **final del array**.
- Los métodos .unshift() y .shift() actuan al **inicio del array**.

Los métodos de inserción .push() o .unshift() insertan los elementos pasados por parámetro en el array y devuelve el tamaño actual del array después de la inserción. Los métodos de extracción, .pop() o .shift(), extraen y devuelven el elemento extraído.

```
const elements = ["a", "b", "c"]; // Array inicial
elements.push("d"); // Devuelve 4. Ahora elements = ['a', 'b', 'c', 'd']
elements.pop(); // Devuelve 'd'. Ahora elements = ['a', 'b', 'c']
elements.unshift("Z"); // Devuelve 4. Ahora elements = ['Z', 'a', 'b', 'c']
elements.shift(); // Devuelve 'Z'. Ahora elements = ['a', 'b', 'c']
```

Arrays

Aunque hemos visto las formas principales de crear un en Javascript, existen otras que permiten generar un partiendo de otras estructuras de datos.

Convertir a array

El método estático Array.from() se utiliza para convertir variables «parecidas» a los **arrays** (*pero que no son arrays*) en un Array. Este es el caso de variables como Strings o de la lista de elementos del DOM.

```
const texto = "12345";
console.log(texto.constructor.name); // "String"
const letras = Array.from(texto); // ["1", "2", "3", "4", "5"]
console.log(letras);
const letras2 = [...texto]; // ["1", "2", "3", "4", "5"]
console.log(letras2);
```

```
String
▶ Array(5) [ "1", "2", "3", "4", "5" ]
1 2 3 4 5
1 2 3 4 5
▶ Array(5) [ "1", "2", "3", "4", "5" ]
```

¿Qué es el operador …?

Es el operador spread, y sirve para pasar un array o un strig a una lista de argumentos

```
const texto = "12345";
console.log(...texto); //
const letras = ["1", "2", "3", "4", "5"];
console.log(...letras);
```

```
1 2 3 4 5
1 2 3 4 5
```

Arrays

VER CUANDO SE ESTUDIEN FUNCIONES

```
const divs = document.querySelectorAll("div");
divs.constructor.name; // "NodeList"
const elements = Array.from(divs); // [div, div, div]
const elements = [...divs]; // [div, div, div]
```

Como se puede ver, es algo muy similar a lo que hacemos con la desestructuración de arrays. Pero no todos los elementos se pueden convertir a arrays. Por ejemplo, si intentamos convertir un undefined o un null, nos dará un error similar a **Uncaught TypeError: null is not iterable**.

De forma opcional, Array.from(obj) puede recibir un parámetro adicional: una función que actuará de forma idéntica a una función map(). Veamos el funcionamiento:

```
const text = "12345";
const numbers = Array.from(text, (number) => Number(number)); // [1, 2, 3, 4, 5]
const numbers = Array.from(text, Number); // Equivalente al anterior // Equivalente a los dos anteriores
const numbers = [...text].map(Number);
```

Observa que en este caso, la función pasada por segundo parámetro del Array.from() se ejecutará por cada uno de los elementos de text, y en este caso concretamente, la función (number) => Number(number) fuerza a convertir cada elemento en un número. La diferencia respecto al ejemplo anterior, es que en este caso obtienes un array de números, mientras que el anterior obtenías un array de textos.

Si lo que buscas es convertir un objeto de Javascript en un Array, probablemente te interese el tema de Iteradores de objetos donde vemos métodos como Object.keys(), Object.values() u

Arrays

Concatenar arrays

El método concat(), nos permite unir los elementos pasados por parámetro en un array a la estructura que estamos manejando.
Se podría pensar que los métodos .push() y concat() funcionan de la misma forma, pero no es exactamente así:

```
const elements = [1, 2, 3];
elements.push(4, 5, 6); // Devuelve 6. Ahora elements = [1, 2, 3, 4, 5, 6]
elements.push([7, 8, 9]); // Devuelve 7. Ahora elements = [1, 2, 3, 4, 5, 6, [7, 8, 9]]
```

El mismo ejemplo con el método .concat():

```
const firstPart = [1, 2, 3];
const secondPart = [4, 5, 6];
firstPart.concat(firstPart); // Devuelve [1, 2, 3, 1, 2, 3]
firstPart.concat(secondPart); // Devuelve [1, 2, 3, 4, 5, 6] // Se pueden pasar elementos sueltos
firstPart.concat(4, 5, 6); // Devuelve [1, 2, 3, 4, 5, 6] // Se pueden concatenar múltiples arrays e incluso mezclarlos con elementos sueltos
firstPart.concat(firstPart, secondPart, 7); // Devuelve [1, 2, 3, 1, 2, 3, 4, 5, 6, 7]
```

El método concat(), a diferencia de push(), no modifica el array sobre el cuál trabajamos, sino que simplemente lo devuelve. Además, al pasar un array por parámetro, push() lo inserta como un array, mientras que concat() inserta cada uno de sus elementos.

Arrays

Separar y unir strings

El método .split() permite crear un Array a partir de un String. El método .join() es su contrapartida. Con .join() podemos crear un String con todos los elementos del array, separándolo por el texto que le pasemos por parámetro:

```
const letters = ["a", "b", "c"];
letters.join("->"); // Devuelve 'a->b->c'
letters.join("."); // Devuelve 'a.b.c'

"a.b.c".split("."); // Devuelve ['a', 'b', 'c']
"5-4-3-2-1".split("-"); // Devuelve ['5', '4', '3', '2', '1']
```

.join() devolverá un String y .split() devolverá un Array.

```
"Hola a todos".split(""); // ['H', 'o', 'l', 'a', ' ', 'a', ' ', 't', 'o', 'd', 'o', 's']
```

En este caso, hemos pedido dividir sin indicar ningún separador, por lo que toma la unidad mínima como separador y nos devuelve un array con cada carácter del string original.

Arrays

Buscar elementos en un array

Método	Descripción
.includes(element)	Comprueba si element está incluido en el array.
.includes(element, from)	Idem, pero partiendo desde la posición from del array.
.indexOf(element)	Devuelve la posición de la primera aparición de element o -1 si no existe.
.indexOf(element, from)	Idem, pero partiendo desde la posición from del array.
.lastIndexOf(element)	Devuelve la posición de la última aparición de element. Devuelve -1 si no existe.
.lastIndexOf(element, from)	Idem, pero partiendo desde la posición from del array.

Ejemplos:

```
const numbers = [5, 10, 15, 20, 25,10];
numbers.includes(3); // false
numbers.includes(15); // true
numbers.includes(15,4);//false
numbers.indexOf(5); // 0
numbers.indexOf(15, 4); // -1
numbers.lastIndexOf(10); // 5
numbers.lastIndexOf(10, 3); // 1
```

Arrays

Ordenacion de arrays

Método	Descripción
.reverse() ⚠	Invierte el orden de elementos del array.
.toReversed() ✓	Devuelve una copia del array, con el orden de los elementos invertido.
.sort() ⚠	Ordena los elementos del array bajo un criterio de **ordenación alfabética.**
.sort(criterio) ⚠	Idem, pero bajo un criterio de ordenación indicado por criterio.
.toSorted() ✓	Devuelve una copia del array, con los elementos ordenados.
.toSorted(criterio) ✓	Idem, pero ordenado por el criterio establecido por parámetro.

✓ El array original está seguro (*no muta*).
⚠ El array original cambia (*muta*).

Ejemplo:

```
const elements = ["A", "B", "C", "D", "E", "F"];
const reversedElements = elements.reverse();
reversedElements // ["F", "E", "D", "C", "B", "A"]
elements // ["F", "E", "D", "C", "B", "A"]
reversedElements === elements // true
```

El array original elements ha mutado, y el nuevo array resultante no es más que una referencia al original.

Arrays

Si queremos crear un nuevo array independiente del original, tendríamos que hacer lo siguiente:

```
const elements = ["A", "B", "C", "D", "E", "F"];
const reversedElements = structuredClone(elements).reverse();
reversedElements // ["F", "E", "D", "C", "B", "A"]
elements // ['A', 'B', 'C', 'D', 'E', 'F']
reversedElements === elements // false
```

Con structuredClone() creamos una copia de la estructura original elements, y luego se invierte. Ahora tenemos dos arrays independientes.
Otra solución es utilizar el nuevo método .toReversed(), ES2023, el cuál funciona exactamente igual que .reverse(), pero sin mutar el original:

```
const elements = ["A", "B", "C", "D", "E", "F"];
const reversedElements = elements.toReversed();
reversedElements // ["F", "E", "D", "C", "B", "A"]
elements // ["A", "B", "C", "D", "E", "F"]
reversedElements === elements // false
```

Arrays

```
const names = ["Alberto", "Zoe", "Ana", "Mauricio", "Bernardo"];
const sortedNames = names.sort();
sortedNames // ["Alberto", "Ana", "Bernardo", "Mauricio", "Zoe"]
names // ["Alberto", "Ana", "Bernardo", "Mauricio", "Zoe"]
sortedNames === names // true
```

Usamos structuredClone() para hacer una copia y que sea independiente:

```
const names = ["Alberto", "Zoe", "Ana", "Mauricio", "Bernardo"];
const sortedNames = structuredClone(names).sort();
sortedNames // ["Alberto", "Ana", "Bernardo", "Mauricio", "Zoe"]
names // ["Alberto", "Zoe", "Ana", "Mauricio", "Bernardo"];
sortedNames === names // false
```

Utilizando .toSorted() se mantiene el array original.

Si intentamos ordenar un array de elementos Number, falla la ordenación:

```
const numbers = [1, 8, 2, 32, 9, 7, 4];
const sortedNumbers = numbers.toSorted();
sortedNumbers // [1, 2, 32, 4, 7, 8, 9]
numbers // [1, 8, 2, 32, 9, 7, 4]
```

Esto es porque sort trabaja con la ordenación alfabética, y ahora necesitamos ordenación natural. Se puede arreglar pasando una función de Comparación a sort (MÁS ADELANTE)

Arrays

Algoritmo de ordenación NOOOOOOOO

Si eres una persona de naturaleza curiosa, es posible que te preguntes que hace exactamente esa **función de comparación**, que a priori, parece que sólo realiza una operación. Primero, analicemos los pasos hechos hasta ahora: const elements = [8, 4]; const sortedElements = elements.toSorted(); // [4, 8]; // Recordemos que esto es equivalente a lo siguiente: const sortedElements = elements.toSorted(function(a, b) { return a + b }); // Si extraemos la función de ordenación: const alphaOrder = function(a, b) { return a + b }; const sortedElements = elements.toSorted(alphaOrder); // Si la cambiamos a función flecha const alphaOrder = (a, b) => a + b; const sortedElements = elements.toSorted(alphaOrder);

Ahora, si profundizamos en la tarea que realiza el sort() o toSorted(), lo que hace concretamente es analizar pares de elementos del array en cuestión para ordenarlos. El primer elemento es a y el segundo elemento es b. Entonces, se ejecutará la función, la cuál devolverá un resultado: const elements = [8, 4]; const alphaOrder = (a, b) => { console.log(`a(${a}) + b(${b}) = ${a+b}`); return a + b; } const sortedElements = elements.toSorted(alphaOrder);

Observa que la primera vez que ejecutas el toSorted(), la función de

Arrays

Crear array bidimensional:

```
let tablaNotas=[[,,],[,,]]; //cada nivel de anidamiento de corchetes indica
una dimensión del array, y el número de elementos separadospor comas es el
número de elementos de esa dimensión
```

Para almacenar valores:

```
tablaNotas[0][0]=1; //Fila 0-Columna 0
tablaNotas[0][1]=2; //Fila 0-Columna 1
tablaNotas[0][2]=3; //Fila 0-Columna 2
tablaNotas[1][0]=4; //Fila 1-Columna 0
tablaNotas[1][1]=5; //Fila 1-Columna 1
tablaNotas[1][2]=6; //Fila 1-Columna 2
```

Otra forma:

```
let tablaNotas=new Array(2);
tablaNotas[0]=new Array(3);
tablaNotas[1]=new Array(3);
```

A partir de ahí, se rellenan los valores como antes.

Arrays

Otras formas de recorrer un array:

For in

```
Let precios=[60,12,99,35,76];
for (let i in precios){//no es necesario inicializar el contador ni incrementarlo
        console.log(`El precio ${i} es:${precios[i]}`);
}
```

For of: simplifica más el proceso, ya que ni siquiera se utiliza una variable para iterar por posición, ya que se realiza automáticamente. Desventaja: se desconocen los índices y que en la salida se muestran los elementos vacíos

```
let precios=[60,12,99,35,76];
for (let precio of precios){
        console.log(precio);
}
```

forEach: Se verá más adelante con las funciones

Conjuntos

Los conjuntos o sets son estructuras de datos parecidas a los arrays, pero que no permiten valores duplicados. Previene esa duplicidad de forma automática.

Crear un conjunto: usando new, ya que se trata de un objeto

```
Let conjunto=new Set();
```

Para indicarle, desde su declaración, los elementos que lo componen inicialmente, hay que pasarle un objeto de tipo iterable (array, map, string, set…)

```
let conjunto1=new Set([34,1,"Girasol",25.9]);
let conjunto2=new Set("cadena");
```

```
▸ Set(4) [ 34, 1, "Girasol", 25.9 ]
▸ Set(5) [ "c", "a", "d", "e", "n" ]
```

Automáticamente ha eliminado los elementos duplicados

Recorrido: con for of

```
For (let elemento of conjunto1){
        console.log(elemento);
}
```

Conjuntos

Adición de elementos: con add

```
let conjunto=new Set();
conjunto.add(7);
conjunto.add("Samuel").add(70).add("hola");
//conjunto{7,"Samuel",70,"hola"}
```

Eliminación de elementos: delete, y devuelve true o false indicando el resultado de la operación.

```
conjunto.delete(70);
//conjunto{7,"Samuel","hola"}
```

Para eliminar todos los elementos se usa clear

```
conjunto.clear();
```

Tamaño de un conjunto: size

```
let conjunto=new Set().add(1).add(2).add(3).add(2);
console.log(conjunto.size);
//3
```

Conjuntos

Búsqueda de un elemento: has. Devuelve true si se ha encontrado el elemento

```
let conjunto=new Set().add(1).add(2).add(3).add(2); If
(conjunto.has(3))
          console.log("Encontrado");
```

Conversiones: al principio, vimos cómo se usaba el operador spread, ahora lo utilizamos para convertir un array a un conunto. Set, al ser una estructura iterable (*se puede recorrer*), es muy sencillo de utilizar con **desestructuración** y convertirlo a un array (*o viceversa).*

```
const conjunto = new Set([5, "A", [99, 10, 24]]);
conjunto.size; // 3 (Contiene 3 elementos)
conjunto.constructor.name; // "Set"
const vector= [...conjunto];
vector.constructor.name; // "Array"
vector; // [5, "A", [99, 10, 24]]
```

Unión: con el operador spread:

```
let   array1=[10,20,30,40,50];   let
array2=[30,50,60,70,80];         let
array3=[60,70,80,90,100];
let conjunto=new Set([…array1,…array2,…array3]);
//conjunto {10,20,30,40,50,60,70,80,90,100}
```

Conjuntos

Cuidado cuando se tengan conjuntos con tipos de datos más complejos con elementos anidados (*arrays, objetos, etc...*), ya que son referencias y modificar un elemento referenciado, modificará el original. Para evitar esto: structuredClone():

```
const conjunto = new Set([5, "A", [99, 10, 24]]);
conjunto.size; // 3
const clonedArray = [...structuredClone(conjunto)];
const array = [...conjunto];
clonedArray[2][0] = "Modified";
[...conjunto][2][0]; // 99 (El original se mantiene intacto)
array[2][0] = "Modified";
[...conjunto][2][0]; // "Modified" (El original ha mutado)
```

Como ya se ha visto, se puede hacer la operación inversa para convertir un array en un set:

```
const array = [5, 4, 3, 3, 4];
const conjunto= new Set(array);
conjunto;// Set({ 5, 4, 3 })
```

Hay más operaciones con Conjuntos: Intersección, Diferencia y Exclusión. SE VERÁN MÁS ADELANTE.

WeakSet

Es una estructura de datos muy parecida a los Sets (no permiten datos duplicados), pero no permiten elementos primitivos.

```
// *** Set
const conjunto = new Set([1, "A", true]); // OK
const conjunto2 = new Set([{ name: “Mario" }, [2, 30]]); // OK
// *** WeakSet
const wconjunto = new WeakSet([1, "A", true]); // ERROR: Uncaught TypeError: Invalid value
                                                                    used in weak set
const wconjunto2 = new WeakSet([{ name: "Mario" }, [2, 30]]); // OK
```

Maps

Un mapa en Javascript es una estructura de datos nativa que permiten implementar una estructura de tipo **mapa**, es decir, una estructuras donde tiene **valores** guardados a través de una **clave** para identificarlos. Comúnmente, esto se denomina **pares clave-valor**.

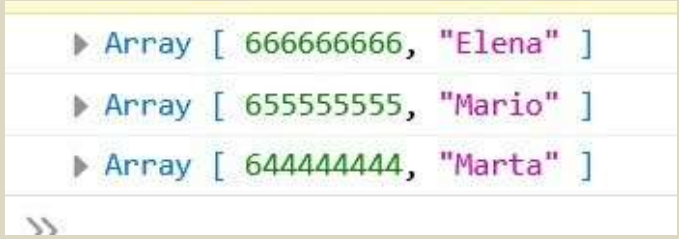
Crear un mapa

```
let map = new Map(); // Mapa vacío
let map = new Map([[1, "uno"]]); // Map({ 1=>"uno" })
let map = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]); // Map({ 1=>"uno",
2=>"dos", 3=>"tres" })
```

En el tercer caso se pasa un array de entradas, o array de array.

Recorrido de un mapa: for..of

```
let telefonos= new Map([
          [666666666,”Elena”],
          [655555555,”Mario”],
          [644444444,”Marta”]
])
for(let persona of telefonos)
          console.log(persona);
```



Esta forma no es muy útil para tratar datos por separado

Maps

Para obtener en diferentes variables las claves y los valores:

```
for (let [teléfono,persona] of teléfonos)
        console.log(`El teléfono de ${persona} es ${telefono}.`);
```

```
El telefono de Elena es 666666666.
El telefono de Mario es 655555555.
El telefono de Marta es 644444444.
```

Además, si sólo se tiene que trabajar con las claves o con los valores, también se pueden usar los métodos: keys y values

```
for(let telefono of telefonos.keys())
        console.log(telefono);
```

```
666666666
655555555
644444444
```

```
for(let persona of telefonos.values())
        console.log(persona);
```

```
Elena
Mario
Marta
```

Maps

size: devuelve tamaño del map

```
const mapa = new Map();
mapa.size; // 0
const mapa2 = new Map([[1, "uno"], [2, "dos"]]);
mapa2.size; // 2
const mapa3 = new Map([[1, "uno"], [2, "dos"], [1, "tres"]]);
mapa3.size; // 2 (El 1->"tres" sobreescribe al anterior)
```

Observa que si introducimos un nuevo par clave-valor que tiene la misma clave que otro (*tenemos dos que comparten la clave 1*), se sobreescribirá. No pueden existir dos pares clave-valor con la misma clave.

Establecer elementos (set) : fija un par clave-valor en el mapa.

- Si usamos .set() para una **clave** que no existe, se añade al mapa.
- Si usamos .set() para una **clave** que ya existe, la sobreescribe.

```
const mapa = new Map();
mapa.set(5, "cinco");
mapa.set("A", "letra A");
mapa.set(5, "cinco sobreescrito"); // Sobreescribe el anterior map; // Map({ 5=>"cinco sobreescrito",
"A"=>"letra A" })
```

Los Mapas pueden utilizar como clave cualquier tipo de dato.

Maps

Comprobar si existen (has): si la clave existe devuelve true, si no false

```
const mapa = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]);
mapa.has(2); // true
mapa.has(34); // false
mapa.set(34, "treinta y cuatro");
mapa.has(34); // true
```

Si se está utilizando tipos de datos más complejos como objetos o arrays , se deberían tener almacenados en una variable, ya que si se crean al momento de pasarlos por parámetro, se pasa su referencia, y podrían no ser los mismos objetos aunque se escriban igual.

Borrar elementos (delete): Devuelve true si lo consigue eliminar, en caso contrario, false.

```
const mapa = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]);
mapa.delete(3); // true
mapa.delete(39); // false
mapa; // Map({ 1=>"uno", 2=>"dos" })
```

Vacíar conjunto (clear): borra todos los elementos del mapa, dejándolo vacío.

```
const mapa = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]);
mapa.clear();
mapa.size; // 0
```

Obtener el valor a partir de una clave(get)

```
const mapa = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]);
mapa.get(1);// devuelve “uno”
```

Maps

Convertir a Arrays: Mediante la desestructuración, podemos convertir los Map en Array o en Objeto.

```
const mapa = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]);
mapa.size; // 3 (Contiene 3 elementos)
mapa.constructor.name; // "Map"
const entries = [...structuredClone(map)];
entries.constructor.name; // "Array"
entries; // [[1, "uno"], [2, "dos"], [3, "tres"]]
```

Recuerda utilizar structuredClone() para clonar la estructura si tiene elementos anidados, ya que sino sólo realizará una **copia superficial** y utilizará referencias para los elementos anidados.

Este array de entradas que nos da como resultado, lo podríamos utilizar para crear un nuevo map, o incluso un objeto:

```
const mapa2 = new Map(entries);
mapa2; // Map({ 1=>"uno", 2=>"dos", 3=>"tres" })
const object = Object.fromEntries(entries);
object; // { 1: "uno", 2: "dos", 3: "tres" }
```

WeakMaps

Se trata de una estructura derivada, muy similar a los Map, pero éstos no permiten utilizar tipos primitivos (, ,) como **clave**, mientras que el Map si lo permite:

```
// *** Map
const mapa = new Map([[1, "uno"]]); // OK
const mapa2 = new Map([[{ id: 1, type: "number" }, "uno"]]); // OK
// *** WeakMap
const wmapa = new WeakMap([[1, "uno"]]); // ERROR: Uncaught TypeError:
                                         Invalid value used in weak map key
const wmapa2 = new WeakMap([[{ id: 1, type: "number" }, "uno"]]); // OK
```