

wawiwa

ESTRUCTURAS DATOS Y ALGORITMOS

JAVASCRIPT

Asignación de desestructuración de arreglo 1

La sintaxis de la **Asignación de desestructuración** es una expresión de Javascript que hace posible desempacar los valores de los arreglos, o propiedades de los objetos, en distintas variables.

```
let a, b;  
[a, b] = [1, 2];  
console.log(a);  
// resultado esperado: 1  
console.log(b);
```

Asignación de desestructuración de arreglo 2

```
let a, b;
[a, b] = [1, 2];
console.log(a); // 1
console.log(b); // 2

[a, b] = [1, 2, 3, 4, 5];
console.log(a); // 1
console.log(b); // 2

({ a, b } = { a: 1, b: 2 });
console.log(a); // 1
console.log(b); // 2
```

Asignación de desestructuración de arreglo 3

```
// Asignación de variable básico
const foo = ['one', 'two', 'three'];
const [red, yellow, green] = foo;
console.log(red); // "one"
console.log(yellow); // "two"
console.log(green); // "three"
```

Asignación de desestructuración de array 4

```
// Asignación separada de la declaración
// Una variable puede ser asignada a un valor con desestructuración,
// separada de la declaración de la variable.

let a, b;
[a, b] = [1, 2];
console.log(a); // 1
console.log(b); // 2
```

Asignación de desestructuración de arreglo 5

```
// Valor predeterminado
// Una variable puede ser asignado con un valor predeterminado,
// en el caso de que el valor desempacado del arreglo sea indefinido .

let a, b;
[a = 5, b = 7] = [1];
console.log(a); // 1
console.log(b); // 7
```

Asignación de desestructuración de arreglo 6

```
// Intercambiando variables
let a = 1;
let b = 3;

[a, b] = [b, a];
console.log(a); // 3
console.log(b); // 1

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

Asignación de desestructuración de arreglo 7

```
function f () {  
    return [1, 2];  
}  
  
let a, b;  
[a, b] = f ();  
console.log(a); // 1  
console.log(b); // 2
```

Asignación de desestructuración de arreglo 8

```
// Ignorando algunos valores retornados
function f() {
    return [1, 2, 3];
}

const [a, , b] = f();
console.log(a); // 1
console.log(b); // 3
const [c] = f();
console.log(c); // 1
```

Asignación de desestructuración de objeto 1

```
const user = {  
  id: 42,  
  isVerified: true  
};  
  
const { id, isVerified } = user;  
  
console.log(id); // 42  
console.log(isVerified); // true
```

Asignación de desestructuración de objeto 2

```
// Asignación sin declaración
let a, b;
({ a, b } = { a: 1, b: 2 });
```

Asignación de desestructuración de objeto 3

```
// Asignando a nuevos nombres de variables
const o = { p: 42, q: true };
const { p: foo, q: bar } = o;
console.log(foo); // 42
console.log(bar); // true
```

Asignación de desestructuración de objeto 4

// Valores predeterminados

```
const { a = 10, b = 5 } = { a: 3 };
console.log(a); // 3
console.log(b); // 5
```

Asignación de desestructuración de objeto 5

```
// Desempaquetar campos de objetos pasados como parámetro de función
const user = {
  id: 30,
  displayName: 'Raz',
  fullName: {
    firstName: 'Mike',
    lastName: 'Laris'
  }
};

function getUserId({ id }) {
  return id;
}

function whois({ displayName, fullName: { firstName: name } }) {
  return `${displayName} is ${name}`;
}

console.log(userId(user)); // 30
console.log(whois(user)); // "Raz is Mike"
```

El operador de propagación de arreglo (...)

La sintaxis del propagador (...) permite a un iterable como un arreglo o un array ser expandido en lugares donde se esperan cero o más argumentos (para llamadas de funciones) o elementos (para literales de arreglo), o una expresión de objeto se expanda en lugares donde se esperan cero o más pares de key-value (para literales de objeto).

```
let arr = [30, 40, 50];
let oldWayArr = [5, 6, arr[0], arr[1], arr[2]];
console.log(oldWayArr);

let newWayArr = [10, 20, ...arr];
console.log(newWayArr);
console.log(...newWayArr);
console.log(10, 20, 30, 40, 50);
```

El operador de propagación de arreglo (...)

2

```
let arr = [30, 40, 50];
// Copia el arreglo
const arrCopy = [...arr];
console.log(arrCopy) // 30 40 50
```

El operador de propagación de arreglo (...)

3

```
let arr1 = [1, 2, 3]
let arr2 = [4, 5, 6]
// Une 2 arreglos
let arrJoin = [...arr1, ...arr2];
console.log(arrJoin);
```

El operador de propagación de arreglo (...) 4

```
// Iterables: arreglos, string, mapas, sets. NO para objetos
let strContinental = 'Europe';
let letters = [...strContinental, ' ', 'and Asia'];
console.log(letters);
console.log(...strContinental);
```

El operador de propagación de arreglo (...)

5

```
const rankPlayers = [
    prompt("1-5 what do you think about Mike1"),
    prompt("1-5 what do you think about Lisa"),
    prompt("1-5 what do you think about Luis"),
];

console.log(rankPlayers);

function average(n1, n2, n3) {
    return (parseInt(n1) + parseInt(n2) + parseInt(n3)) / 3;
}

console.log(rankPlayers);
console.log(average(rankPlayers[0], rankPlayers[1], rankPlayers[2]));
console.log(average(...rankPlayers));
```

El operador de propagación de objeto (...) 1

```
// Asignamiento básico  
  
const user = { id: 42, isVerified: true };  
  
const { id, isVerified } = user;  
console.log(id); // 42  
console.log(isVerified); // true
```

El operador de propagación de objeto (...) 2

```
// Propaga en el los literales del objeto
let obj1 = { foo: 'bar', x: 42 };

let obj2 = { foo: 'baz', y: 13 };

let clonedObj = { ...obj1 };

// Objeto { foo: "bar", x: 42 }

let mergedObj = { ...obj1, ...obj2 };

// Objeto { foo: "baz", x: 42, y: 13 }
```

El operador de propagación de objeto // Objetos 3

```
const newShift = {  
    Country: "England",  
    ...shift,  
    shiftManager: 'Beker'  
};  
  
const shift = {  
    shiftId: '1',  
    workplace: 'L.A.',  
    workers: [],  
    start: "",  
    end: ""  
};  
  
// Literales de objeto mejorados de ES6  
startShift(startTime = '8:00') {  
    console.log(`Shift started! ${startTime}`);  
},  
endShift(endTime = '0:00') {  
    console.log(`Shift ended! ${endTime}`);  
},  
workerForShift(w1, w2, w3) {  
    console.log(`Worker at first shift ${w1}, ${w2} and ${w3}`);  
}  
};  
  
shiftCopy.workerForShift(...arrWorkers);
```

El parámetro rest 1

```
// sintaxis
// El último parámetro de la definición de una función puede ser prefijado
// con "... " (tres U+002E FULL STOP caracteres),
// que pueda causar que todos los parámetros (dados por el usuario) que sobran
// sean colocados en un arreglo de Javascript estándar.
// sólo el último parámetro de una función puede ser
// un parámetro rest.

function f(a, b, ...theArgs) {
    // ...
}
```

El parámetro rest 2

```
// Una función puede tener solamente un ...restParam.  
// foo(...one, ...wrong, ...)  
// El parámetro rest debe ser el último parámetro de la función.  
// foo(...wrong, arg2, arg3)  
// foo(arg1, arg2, ...correct)
```

El parámetro rest 3

```
// Usando parámetros rest
// En este ejemplo, el primer argumento es mapeado en "a" y el segundo en "b",
// por lo que estos argumentos nombrados se utilizan como normales.

// Sin embargo, el tercer argumento, manyMoreArgs,
// será un arreglo que contenga el tercer,
// cuarto, quinto, sexto ... — cuantos argumentos
// que el usuario incluya.

function myFun (a, b, ...manyMoreArgs) {
    console.log("a", a);
    console.log("b", b);
    console.log("manyMoreArgs", manyMoreArgs);
}

myFun ("one", "two", "three", "four", "five", "six");
// a, "one"
// b, "two"
// manyMoreArgs, ["three", "four", "five", "six"] <-- notice it's an array
```

El parámetro rest 4

```
// Largo del Argumento
// Ya que theArgs es un arreglo, le cuenta de sus elementos es dada por la propiedad length.

function fun1 (...theArgs) {
    console.log(theArgs.length);

}

fun1(); // 0
fun1(5); // 1
fun1(5, 6, 7); // 3
```

La corta circulación 1 bueno saber

```
a || (b * c); // evalua primero `a`, después
produce `a` si `a` es "truthy"
a && (b < c); // evalua primero `a`, después
produce `a` si `a` es
a ?? (b || c); // evalua primero `a`, después
produce `a`, si `a` no es `null` y no es
`undefined`
a ? .b.c; // evalua primero `a`, luego produce
`undefined` si `a` es `null` o `undefined`
```

"Short-circuiting" es un término técnico para la evaluación condicional. Por ejemplo, en la expresión `a && (b + c)`, si `a` es **falsy**, entonces la sub-expresión `(b + c)` no va a llegar a ser evaluada, aunque esté entre paréntesis. Podríamos decir que el operador de disyunción ("OR") es "short-circuited" (que corta). Junto con la disyunción lógica, otros operadores "short-circuited" incluyen el operador ("AND"), la coalescencia nula, el encadenamiento opcional y el operador condicional. A continuación hay unos ejemplos.

```
a || (b * c); // evalua primero `a`, después produce `a` si `a` es "truthy"
a && (b < c); // evalua primero `a`, después produce `a` si `a` es "falsy"
a ?? (b || c); // evalua primero `a`, después produce `a` si `a` no es `null` y no es `undefined`
a?.b.c; // evalua primero `a`, luego produce `undefined` si `a` es `null` o `undefined`
```

La corta circulación 2 bueno saber

```
3 > 2 && 2 > 1
// retorna true
3 > 2 > 1
// Retorna false porque 3 > 2 es true, después true se convierte en 1
// en operadores de desigualdad, por lo tanto true > 1 se convierte en 1 > 1, el cual
// es false. Agregar paréntesis lo hace más claro: (3 > 2) > 1.
```

El operador de coalescencia nula(??)

```
// El operador de coalescencia nula (??)
const foo = null ?? 'default string';
console.log(foo);
// resultado esperado: "default string"

const baz = 0 ?? 42;
console.log(baz);
// resultado esperado: 0
```

El operador de coalescencia nula (??) es un operador lógico que retorna el operando a su derecha cuando el operando a su izquierda es **null** o **undefined**, y en caso contrario, retorna el operando a su izquierda.

Esto puede ser contrastado con el **operador lógico OR(|||)**, que retorna el operando del lado derecho si es que el operando del lado izquierdo tiene cualquier valor **falsy**, no únicamente **null** o **undefined**. En otras palabras, si usas **||** para proporcionar algún valor predeterminado a otra variable **foo**, puedes enfrentarte a algunos comportamientos inesperados si consideras valores “falsy” como válidos (ejemplo: **"0"** o **0**). Mira aquí para más ejemplos.

El operador de coalescencia nula tiene el The nullish coalescing operator has the quinta **prioridad** más baja de los operadores , directamente más bajo que **||** y directamente arriba que el **operador ternario condicional**.

El operador de coalescencia nula(??)

Sintaxis :

leftExpr ?? rightExpr

```
const nullValue = null;
const emptyText = ""; // falsy
const someNumber = 42;

const valA = nullValue ?? "default for A";
const valB = emptyText ?? "default for B";
const valC = someNumber ?? 0;

console.log(valA); // "default for A"
console.log(valB); // "" (el string vacío no es ni null ni undefined)
console.log(valC); // 42
```

El operador de coalescencia nula(??)

```
// Asignar un valor predeterminado a una variable
// Anteriormente, cuando uno quería asignar un valor predeterminado a una variable, un
patrón común era usar el operador lógico OR (||):
let foo;
// foo nunca fue asignado a ningún valor entonces sigue siendo undefined
let someDummyText = foo || 'Hello!';
```

El operador de coalescencia nula(??)

```
// sin embargo, debido a que || es un operador lógico boolean,  
// el operando del lado izquierdo fue forzado a un boolean  
// para la evaluación y cualquier valor falsy (0, '', NaN, null, undefined)  
// no se retornó. Este comportamiento puede  
// causar consecuencias inesperadas si consideras 0,  
// '', o NaN como valores válidos.  
  
let count = 0;  
let text = "";  
let qty = count || 42;  
let message = text || "hi!";  
console.log(qty); // 42 and not 0  
console.log(message); // "hi!" and not ""
```

El operador de coalescencia nula(??)

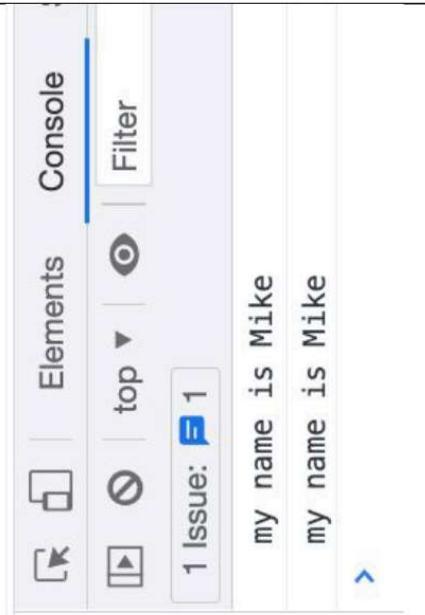
```
// short-circuiting
// Como los operadores lógicos OR Y AND,
// la expresión del lado derecho no es evaluado
// si la del lado izquierdo prueba no ser null o undefined.
function A() { console.log('A was called'); return undefined; }
function B() { console.log('B was called'); return false; }
function C() { console.log('C was called'); return "foo"; }

console.log(A() ?? C());
// imprime "A was called" cuando "C was called" y después "foo"
// como A() retornó undefined así que las 2 expresiones son evaluadas

console.log(B() ?? C());
// imprime "B was called" luego "false"
// como B() retornó false (Y no null o undefined), la expresión de la derecha no fue
evaluada
```

El uso del carácter backtick (comilla invertida) (`) in JavaScript

```
const name = "Mike";  
console.log("my name is " + name);  
console.log(`my name is ${name}`);
```



Iterando sobre un arreglo

```
const restaurant = {  
    timeShift1: '8:00-16:00',  
    timeShift2: '16:00-00:00',  
    shiftsWorkers: [  
        ['Mike', 'James', 'John', 'Michael', 'William', 'David'],  
        ['Joseph', 'Thomas', 'Charles', 'Christopher', 'Daniel', 'Mark']  
    ],  
    totalTipShift1: 4000,  
    totalTipShift2: 5000,  
    tipped: ['Mike', 'James', 'Michael', 'David', 'Thomas'],  
    date: '12/12/2021',  
};  
  
const [a, b] = [...restaurant.shiftsWorkers];  
const allWorkers = [...a, ...b];  
console.log(allWorkers);  
  
for(const item of allWorkers) { console.log(item); }  
  
for(const item of allWorkers.entries()) { console.log(item); }  
  
for(const [i, el] of allWorkers.entries()) { console.log(`#${i + 1} • ${el}`); }
```

Encadenando objetos

```
const restaurant = {  
    timeshift1: '8:00-16:00',  
    timeshift2: '16:00-00:00',  
    shiftsWorkers: [  
        'Mike', 'James', 'John', 'Michael', 'William', 'David',  
        'Joseph', 'Thomas', 'Charles', 'Christopher', 'Daniel', 'Mark',  
    ],  
    totalTipShift1: 4000,  
    totalTipShift2: 5000,  
    tipped: ['Mike', 'James', 'Michael', 'David', 'Thomas'],  
    date: '12/12/2021',  
    address: { city: "NY", street: "Arguban 6" }  
};  
  
console.log(restaurant.address.city);
```

Looping object keys and values

```
const days = [ 'monday', 'tuesday', 'wednesday', 'thursday',
  'friday', 'saturday', 'sunday' ];

const restaurant = {
  timeShift1: '8:00-16:00',
  timeShift2: '16:00-00:00',
  shiftsWorkers: [
    ['Mike', 'James', 'John', 'Michael', 'William', 'David'],
    ['Joseph', 'Thomas', 'Charles', 'Christopher', 'Daniel', 'Mark'],
  ],
  totalTipShift1: 4000,
  totalTipShift2: 5000,
  tipped: ['Mike', 'James', 'Michael', 'David', 'Thomas'],
  date: '12/12/2021',
  address: { city: "NY", street: "Arguban 6" },
  opendays: [days[0], days[3], days[4]],
};

const properties = Object.keys(restaurant);
console.log(properties);

for (const day of Object.values(restaurant.opendays)) {
  console.log(day);
}
```

Set

El objeto **Set** nos permite almacenar valores únicos de cualquier tipo, ya sean valores primitivos o referencias de objetos.

Descripción

Los objetos Set son colecciones de valores. Se puede iterar sobre los elementos de un set en orden de inserción. Un valor en un set puede ocurrir solamente una vez; es único en la colección del Set.

Set has, delete y add

```
const mySet1 = new Set();

mySet1.add(1); // Set [ 1 ]
mySet1.add(5); // Set [ 1, 5 ]
mySet1.add(5); // Set [ 1, 5 ]
mySet1.add('some text'); // Set [ 1, 5, 'some text' ]

const o = { a: 1, b: 2 };
mySet1.add(o);

mySet1.add({ a: 1, b: 2 });
// o está referenciando un objeto diferente, entonces está bieno is referencing a different object, so this is okay

mySet1.has(1); // true
mySet1.has(3); // false, ya que el 3 no ha sido agregado al set
mySet1.has(5); // true
mySet1.has(Math.sqrt(25)); // true
mySet1.has('Some Text'.toLowerCase()); // true
mySet1.has(o); // true

mySet1.size; // 5

mySet1.delete(5); // remueve 5 del set
mySet1.has(5); // false, 5 ha sido removido

mySet1.size; // 4, ya que recientemente removimos un valor

console.log(mySet1);
// logs Set(4) [ 1, 'some text', ... ] en Firefox
// logs Set(4) { 1, 'some text', ... } en Chrome
```

Set add

```
const mySet1 = new Set();
mySet1.add(1); // Set [ 1 ]
mySet1.add(5); // Set [ 1, 5 ]
mySet1.add(5); // Set [ 1, 5 ]
mySet1.add('some text'); // Set [ 1, 5, 'some text' ]
const o = { a: 1, b: 2 };
mySet1.add(o);

mySet1.add({ a: 1, b: 2 });
// o hace referencia a un diferente objeto, entonces está bien
```

Set has

```
mySet1.has(1) // true
mySet1.has(3) // false, ya que 3 no ha sido agregado al set
mySet1.has(5) // true
mySet1.has(Math.sqrt(25)) // true
mySet1.has('Some Text'.toLowerCase()) // true
mySet1.has(o) // true

mySet1.size // 5
```

Set delete

```
mySet1.delete(5); // remueve 5 del set  
mySet1.has(5); // false, 5 ha sido removido  
  
mySet1.size; // 4, ya que recientemente removimos un valor  
  
console.log(mySet1);  
// logs Set(4) [ 1, "some text", ..., ... ] en Firefox  
// logs Set(4) { 1, "some text", ..., ... } en Chrome
```

Todo el archivo set junto

```
const mySet1 = new Set();
mySet1.add(1); // Set [ 1 ]
mySet1.add(5); // Set [ 1, 5 ]
mySet1.add(5); // Set [ 1, 5 ]
mySet1.add('some text'); // Set [ 1, 5, 'some text' ]

const o = { a: 1, b: 2 };
mySet1.add(o);

mySet1.add({ a: 1, b: 2 }); // o hace referencia a un diferente objeto, entonces está bien

mySet1.has(1); // true
mySet1.has(3); // false, ya que 3 no ha sido agregado al set
mySet1.has(5); // true
mySet1.has(Math.sqrt(25)); // true
mySet1.has('Some Text'.toLowerCase()); // true
mySet1.has(o); // true

mySet1.size; // 5

mySet1.delete(5); // remueve 5 del set
mySet1.has(5); // false, 5 ha sido removido

mySet1.size; // 4, ya que recientemente removimos un valor

console.log(mySet1);
// logs Set(4) [ 1, "some text", ..., ... ] en Firefox
// logs Set(4) { 1, "some text", ..., ... } en Chrome
```

Iterar en un set - clave y valores

```
// iterar en los ítems de un set
// imprime los ítems en este orden: 1, "some text", {"a": 1, "b": 2}, {"a": 1, "b": 2}
for (let item of mySet1) console.log(item);

// imprime los ítems en este orden : 1, "some text", {"a": 1, "b": 2}, {"a": 1, "b": 2}
for (let item of mySet1.keys()) console.log(item);

// imprime los ítems en este orden : 1, "some text", {"a": 1, "b": 2}, {"a": 1, "b": 2}
for (let item of mySet1.values()) console.log(item);

// imprime los ítems en este orden : 1, "some text", {"a": 1, "b": 2}, {"a": 1, "b": 2}
// (clave y valor son los mismos en este caso)
for (let [key, value] of mySet1.entries()) console.log(key);
```

Iterando sobre un set.

```
// convirtiendo entre Set y arreglo  
const mySet2 = new Set([1, 2, 3, 4]);  
mySet2.size; // 4  
// [...mySet2] [1, 2, 3, 4]
```

set y arreglos - remover un duplicado

```
const myArray = [ 'value1' , 'value2' , 'value3' ] ;  
  
// Usa el constructor de Set normal para transformar un arreglo en un Set  
const mySet = new Set (myArray) ;  
  
mySet .has ( 'value1' ) ; // retorna true  
  
// Usa en operador de propagación ... para transformar un Set en un arreglo  
console .log ([ ...mySet ]) ; // te va a mostrar exactamente el mismo arreglo que myArray
```

Set y arreglos

```
const myArray = [ 'value1' , 'value2' , 'value3' ] ;  
  
// Usa el constructor de Set normal para transformar un arreglo en un Set  
const mySet = new Set (myArray) l  
  
mySet .has ( 'value1' ) ; // retorna true  
  
//Usa en operador de propagación ... para transformar un Set en un arreglo  
console .log ( [ ...mySet ] ) ; // va a mostrar exactamente el mismo arreglo que myA
```

set y arreglos - remover elementos duplicados

```
// Se usa para remover elementos duplicados de un arreglo
const numbers =
  [2, 3, 4, 4, 2, 3, 3, 4, 4, 5,
  5, 6, 6, 7, 5, 32, 3, 4, 5];
console.log([...new Set(numbers)]);
// [2, 3, 4, 5, 6, 7, 32]
```

set y string

```
const text = 'India';
const mySet = new Set(text); // Set(5) { 'I', 'n', 'd', 'i', 'a' }

mySet.size; // 5

// sensible a mayúsculas y minúsculas y omisión de duplicados
new Set("FireFox"); // Set(7) { "F", "i", "r", "e", "f", "o", "x" }
new Set("firefox"); // Set(6) { "f", "i", "r", "e", "o", "x" }
```

Map

El objeto Map tiene pares de key-value (clave - valor) y recuerda el orden original de la inserción de las claves. Cualquier valor puede ser usado como clave o valor.

```
const map1 = new Map();
map1.set('a', 1);
map1.set('b', 2);
map1.set('c', 3);

console.log(map1.get('a')) // resultado esperado: 1
map1.set('a', 97);
console.log(map1.get('a')) // resultado esperado : 97

console.log(map1.size); // resultado esperado : 3
map1.delete('b');
console.log(map1.size); // resultado esperado : 2
```

Map

```
// El uso correcto para guardar datos en el objeto Map es por medio del método
set(key, value);

const contacts = new Map();

contacts.set('Jessie', { phone: "213-555-1234", address: "123 N 1st Ave" });

contacts.has('Jessie'); // true
contacts.get('Hillary'); // undefined

contacts.set('Hillary', { phone: "617-555-4321", address: "321 S 2nd St" });

contacts.get('Jessie'); // { phone: "213-555-1234", address: "123 N 1st Ave" }

contacts.delete('Raymond'); // false
contacts.delete('Jessie'); // true

console.log(contacts.size); // 1
```

Map - usando el objeto Map

```
// Usando el objeto map
const myMap = new Map();
const keyString = 'a string';
const keyObj = {};
const keyFunc = function() {};

// configurando los valores
myMap.set(keyString, "valor asociado con ' string' ");
myMap.set(keyObj, 'valor asociado con keyObj');
myMap.set(keyFunc, 'valor asociado con keyFunc');
myMap.size; // 3

// obteniendo los valores
myMap.get(keyString); // "valor asociado con 'a string'"
myMap.get(keyObj); // "valor asociado con keyObj"
myMap.get(keyFunc); // "valor asociado con keyFunc"
myMap.get('a string'); // "valor asociado con 'a string"
// porque keyString === 'a string'

myMap.get({}); // undefined, porque keyObj !== {}
myMap.get(function() {}); // undefined, porque keyFunc !== function() {}
```

Map - Usando NaN como claves de map

```
// Usando NaN como claves de Map
// NaN también puede ser usado como clave.
// Aunque todo NaN es equivalente a
// si mismo (NaN != NaN is true),
// El siguiente ejemplo funciona porque NaN son
// indistinguible uno del otro:
const myMap = new Map();

myMap.set(NaN, 'not a number');
myMap.get(NaN);
// "not a number"

const otherNaN = Number('foo');
myMap.get(otherNaN); // "not a number"
```

Map - iterando sobre un mapa

```
// Los Maps pueden ser iterados usando un ciclo for...of:  
const myMap = new Map();  
myMap.set(0, 'zero');  
myMap.set(1, 'one');  
  
for (const [key, value] of myMap) {  
    console.log(key + ' = ' + value);  
} // 0 = zero// 1 = one  
  
for (const key of myMap.keys()) {  
    console.log(key);  
} // 0// 1  
  
for (const value of myMap.values()) {  
    console.log(value);  
} // zero// one  
  
for (const [key, value] of myMap.entries()) {  
    console.log(key + ' = ' + value);  
} // 0 = zero// 1 = one
```

Map - relación objeto arreglo - bueno saberlo

```
// Relación con objeto arreglo
const kvArray = [ ['key1', 'value1'], ['key2', 'value2'] ];

// Usa el constructor normal Map para transformar un arreglo 2D clave-valor en un map
const myMap = new Map(kvArray);

myMap.get('key1'); // retorna "value1"

// Usa Array.from() para transformar un mapa en un arreglo 2D clave-valor
console.log(Array.from(myMap)); // Te va a mostrar el mismo arreglo que kvArray

// Una manera suficiente de hacer lo mismo usando el sintaxis propagador
console.log(...myMap);

// O utiliza los iteradores keys() o values(), y conviertelos en un arreglo
console.log(Array.from(myMap.keys())); // ["key1", "key2"]
```

Map - clonando un map - bueno saberlo

```
// Clonando y uniendo maps
// Asi como los arreglos, los mapas pueden ser clonados:
const original = new Map([
  [1, 'one']
]);

const clone = new Map(original);

console.log(clone.get(1)) // one
console.log(original === clone) // false (util para comparaciones superficiales)
```

Map - clonando un map - bueno saberlo

```
// Los Maps pueden ser fusionados manteniendo su unicidad de claves:  
const first = new Map([  
  [1, 'one'],  
  [2, 'two'],  
  [3, 'three'],  
]);
```

```
const second = new Map([  
  [1, 'uno'],  
  [2, 'dos'],  
]);
```

```
// Fusiona dos maps. La última clave repetida gana.  
// El operador de propagación esencialmente convierte un mapa en un arreglo  
const merged = new Map([...first, ...second]);  
console.log(merged.get(1)); // uno  
console.log(merged.get(2)); // dos  
console.log(merged.get(3)); // three
```

Map - fusionar un map - bueno saberlo

```
// Los Maps pueden ser fusionados con Arreglos, también:  
const first = new Map([  
  [1, 'one'],  
  [2, 'two'],  
  [3, 'three']  
]);  
  
const second = new Map([  
  [1, 'uno'],  
  [2, 'dos']  
]);  
  
// Fusiona un mapa con un arreglo. La última clave repetida gana.  
const merged = new Map([...first, ...second, [1, 'eins']]);
console.log(merged.get(1)); // eins
console.log(merged.get(2)); // dos
console.log(merged.get(3)); // three
```

Resumen - Arreglo, Set, Map

Los datos vienen de:

1. Programador
2. UI
3. Fuente externa

¿Dónde guardarla ?

`Lista simple == > arreglo de sets`

- arreglo – cuando necesitas una lista ordenada de valores.
- Sets cuando necesitas trabajar con valores únicos y remover los duplicados

Resumen - Arreglo, Set, Map

map	object
maps: mejor desempeño	objects: mas tradicional clave/
maps: las claves pueden tener	valor.
cualquier tipo de datos	objects: las claves pueden solo
maps: fácil de iterar	ser strings.
maps: fácil de calcular el	objects: se usa cuando
tamaño	trabajamos con json

