

Desarrollo Web en Entorno Cliente

Tema 15





Callbaks
Promesas. Más
tipos de datos

callback

Una *callback* (llamada de vuelta o devolver la llamada) es una función que se pasa a otra función como argumento para que se invoque dentro de la función externa y así completar algún tipo de acción.

Ejemplo:

```
function saludar(nombre){  
    alert('Hola ' + nombre);  
}  
  
function procesarEntradaUsuario(callback){  
    const nombre=prompt('Dime tu nombre:');  
    callback(nombre);  
}  
  
procesarEntradaUsuario(saludar);
```

[callback]

Una *callback* puede ser síncrona, el código se detiene o se ejecuta de forma lineal cada instrucción o llamada a función se ejecuta después de que la anterior termine, o asíncrona, de modo que el código continúa y cuando se produzca “algo”, como cuando pasado un tiempo se ejecuta un *setTimeout*, se ejecutará la función pasada por argumento.

Este último caso es el uso habitual con AJAX.

Ejemplo:

```
function AJAX(url, callback){  
    http_request = new XMLHttpRequest();  
    http_request.onreadystatechange=callback;  
    http_request.open('GET',url,true);  
    http_request.send();  
}
```

(callback hell)

Las funciones callback, a pesar de ser una forma flexible y potente de controlar la asincronía, tienen ciertas desventajas evidentes.

Entre las que destaca que el código creado con las funciones callback anidadas se vuelve poco legible, caótico y quizás difícil de entender por programadores ajenos a la aplicación.

Por ejemplo, a la hora de tener que gestionar la asincronía varias veces en una misma función, mediante funciones callbacks anidadas, se obtiene una estructura en forma triangular conocida como *Callback Hell* o *Pyramid of Doom*.



[promesa]

En JS hay ciertas acciones que no se sabe cuánto tardaran, como, por ejemplo, cargar una imagen con JS desde el navegador, solicitar datos al servidor que están almacenados en una base de datos..., el tiempo que tardarán estas acciones depende de la velocidad de conexión, del tamaño de la imagen, de la cantidad de datos que se solicitan, etc.

Si no hubiera código asíncrono, el usuario no podría hacer nada mientras se está esperando que se completen estas tareas, ya que el navegador estará bloqueado esperando que se cargue la imagen, o se descarguen los datos.

Gracias al código asíncrono, se puede responder a otras acciones que el usuario haga mientras se está esperando a que se completen estas acciones.

Las promesas son una manera elegante de usar código asíncrono.

Antes de la existencia de las promesas, todo se hacía con funciones callback, que anidadas una dentro de otra daba lugar a lo que llama el “infierno de callbacks” (*Callback Hell*).

[promesa]

Las promesas (*promises*), que surgen como una solución a los problemas que ocasionan las callback, son un tipo especial de objeto que representa el éxito o el fracaso de una operación asíncrona.

Básicamente, es un objeto al que se le adjuntan funciones, en lugar de pasarlas como argumentos a una función. Dichas funciones se ejecutarán dependiendo de si se completó o no una operación asíncrona.

Las promesas en JS, como las de la vida real, tienen tres estados: “*Te prometo que mañana te entrego el ejercicio*”

- ❑ **Promesa pendiente** (*pending*) hoy, la promesa está en un estado incierto, hay que esperar a ver que sucede.

Y, mañana, pueden ocurrir dos cosas:

- ❑ **Promesa resuelta** (*fulfilled*), la promesa se cumple.

- ❑ **Promesa se rechaza** (*rejected*), la promesa no se cumple.

[promesa]

Las promesas en JS se representan a través de un objeto de tipo *Promise* que puede hallarse en un estado muy concreto: *pendiente*, *aceptada* o *rechazada*.

Cada objeto promesa tiene los siguientes métodos que se pueden utilizar:

<i>Métodos</i>	<i>Descripción</i>
<i>then(resolve)</i>	Ejecuta la función <i>resolve</i> cuando la promesa se cumple (también se dice que la promesa se acepta).
<i>catch(reject)</i>	Ejecuta la función <i>reject</i> cuando la promesa no se cumple (también se dice que la promesa se rechaza).
<i>then(resolve, reject)</i>	Método equivalente a los dos anteriores en el mismo <i>then</i> .
<i>finally(end)</i>	Se ejecutará la función <i>end</i> se cumpla o no la promesa.

[promesa]

La sintaxis de una promesa es la siguiente:

```
const promesa=new Promise(funcion1(resolve,reject));  
promesa  
  .then(funcion2())  
  .catch(funcion3())  
  [.finally(funcion4())];
```

funcion1, se conoce como el cuerpo de la promesa y se ejecutará de manera asíncrona; dentro de esta función se comprobará si se cumple o no la promesa, en caso de que se cumpla, desde *funcion1* se ejecutará la sentencia *resolve()*, con los argumentos que se necesiten. La sentencia *resolve* ejecutará el código del *then*, en este caso la *funcion2*.

Si en *funcion1* se comprueba que no se cumple la promesa, en caso se ejecutará la sentencia *reject()*, con los argumentos que se necesiten. La sentencia *reject* ejecutará el código del *catch*, en este caso la *funcion3*.

La *funcion4* se ejecutará siempre, al finalizar la promesa, ya que está en la sección *finally*.

[promesa]

Hay que tener en cuenta que las promesas son asíncronas:

Inicio del código
Cuerpo de la promesa
Fin del código
Resuelta

```
console.log('Inicio del código');  
  
const promesa=new Promise((resolve,reject)=>{  
    console.log('Cuerpo de la promesa');  
    resolve("Resuelta");  
});  
promesa.then(console.log);  
console.log('Fin del código');
```

Observa que mediante *resolve("Resuelta")* se envía *Resuelta* como argumento a la función que hubiera dentro de los paréntesis de *promesa.then()*.

Cuando un *arrow function* recibe un único argumento que a continuación se pasa como argumento a otra función se puede resumir, de la siguiente forma:

`promesa.then(console.log)` equivale a `promesa.then(mensaje=>console.log(mensaje))`

[promesa]

El siguiente ejemplo crea una promesa que se resuelve automáticamente:

```
const promesa=new Promise((resolve,reject)=>{  
    resolve('Resuelta');  
});  
promesa.then(console.log);
```

Cuando una promesa no se cumple y no se ha utilizado el método *catch()* para definir una función en ese caso se obtiene un error:

```
✖ Uncaught (in promise) DOMException: play() failed  
because the user didn't interact with the document  
first.
```

Como el error que se produce al intentar reproducir un sonido en el ejemplo *Counter-Sonido* del tema 15, donde se estudiaron los eventos.

[promesa]

En el siguiente ejemplo, la función suerte devuelve un objeto promesa, que se ejecutará de forma asíncrona generando un número aleatorio entre el 1 y el 6, si el resultado es 6 la promesa se cumple, en cualquier otro caso se rechaza:

```
function suerte(){  
  return new Promise((resolve,reject)=>{  
    const aleatorio=1+Math.floor(Math.random()*6);  
    if (aleatorio==6)  
      resolve(`Resuelta: ${aleatorio}`);  
    else  
      reject(`Rechazada: ${aleatorio}`);  
  })  
  
  suerte().then(console.Log).catch(console.Log);
```

[promesa]

El objeto *Promise* dispone de varios métodos estáticos que devuelven una promesa:

<i>Métodos</i>	<i>Descripción</i>
<i><code>Promise.all(array)</code></i>	Acepta la promesa sólo si todas las promesas del array se cumplen.
<i><code>Promise.allSettled(array)</code></i>	Acepta la promesa sólo si todas las promesas del array se cumplen o rechazan.
<i><code>Promise.any(array)</code></i>	Acepta la promesa con el valor de la primera promesa del array que se cumpla.
<i><code>Promise.race(value)</code></i>	Acepta o rechaza la promesa solo teniendo en cuenta la primera promesa que se procese.
<i><code>Promise.resolve(value)</code></i>	Devuelve un valor envuelto en una promesa que se cumple directamente.
<i><code>Promise.reject(value)</code></i>	Devuelve un valor envuelto en una promesa que se rechaza directamente.

[Promise.all]

El método *all* de *Promise* es como una promesa compuesta de muchas promesas, devuelve una promesa que se cumple cuando todas las promesas del array que recibe como argumento se cumplen.

Si alguna de ellas se rechaza, *Promise.all()* también se rechaza.

Ejemplo:

```
function suerte(){
  return new Promise((resolve,reject)=>{
    const aleatorio=1+Math.floor(Math.random()*6);
    console.log(aleatorio);
    if (aleatorio==6)
      resolve(`${aleatorio}`);
    else
      reject(`${aleatorio}`);})
}
Promise.all([suerte(),suerte(),suerte()])
  .then(n=>console.log("Ganamos!!"))
  .catch(n=>console.log("Perdimos por: ",n))
```

2
3
6
Perdimos por: 2

6
6
6
Ganamos!!

[Promise.allSettled]

El método *allSettled* de *Promise* sirve para procesar un conjunto de promesas, y devuelve una promesa que se cumple cuanto todas las promesas del array que se pasan como argumento se hayan procesado, independientemente de que se hayan cumplido o rechazado.

Ejemplo:

```
function suerte(){
  return new Promise((resolve,reject)=>{
    const aleatorio=1+Math.floor(Math.random()*6);
    console.log(aleatorio);
    if (aleatorio==6)
      resolve(`${aleatorio}`);
    else
      reject(`${aleatorio}`);})
}
Promise.allSettled([suerte(),suerte(),suerte()])
.then(n=>console.log("Ganamos!!"))
.catch(n=>console.log("Perdimos por: ",n))
```

6
2
1
Ganamos!!

2
1
5
Ganamos!!

4
3
6
Ganamos!!

[Promise.allSettled]

A su vez, la promesa devuelta por método *allSettled* devuelve un array de objetos que hacen referencia a cada una de las promesas que se procesaron, en cada objeto hay una propiedad *status* donde se indica si la promesa ha sido cumplida o rechazada, y una propiedad *reason* con los valores devueltos por la promesa.

Ejemplo:

```
function suerte(){
  return new Promise((resolve,reject)=>{
    const aleatorio=1+Math.floor(Math.random()*6);
    console.log(aleatorio);
    if (aleatorio==6)
      resolve(`${aleatorio}`);
    else
      reject(`${aleatorio}`);})
}
Promise.allSettled([suerte(),suerte(),suerte()])
.then(n=>console.log(n));
```

```
2
5
2
▼ (3) [{...}, {...}, {...}] ⓘ
  ► 0: {status: 'rejected', reason: '2'}
  ► 1: {status: 'rejected', reason: '5'}
  ► 2: {status: 'rejected', reason: '2'}
  length: 3
  ► [[Prototype]]: Array(0)
```


[Promise.any]

El método *any* de *Promise* devuelve una promesa con el valor de la primera promesa individual del array pasado como argumento que se cumpla. Si todas las promesas se rechazan, entonces devuelve una promesa rechazada.

Ejemplo:

```
function suerte(){
  return new Promise((resolve,reject)=>{
    const aleatorio=1+Math.floor(Math.random()*6);
    console.log(aleatorio);
    if (aleatorio==6)
      resolve(`${aleatorio}`);
    else
      reject(`${aleatorio}`);})
}
Promise.any([suerte(),suerte(),suerte()])
.then(n=>console.log("Ganamos!!"))
.catch(n=>console.log("Perdimos por: ",n))
```

3

6

3

Ganamos!!

5

3

5

Perdimos por: AggregateError: All promises were rejected

[Promise.race]

El método *race* de *Promise* devuelve la primera promesa, del array que se pasa como argumento, que sea procesada, independientemente de que se haya cumplido o rechazado. Si esta promesa se cumple, devuelve una promesa cumplida, en caso negativo, devuelve una rechazada.

Ejemplo:

```
function suerte(){
  return new Promise((resolve,reject)=>{
    const aleatorio=1+Math.floor(Math.random()*6);
    console.log(aleatorio);
    if (aleatorio==6)
      resolve(`${aleatorio}`);
    else
      reject(`${aleatorio}`);})
}
Promise.race([suerte(),suerte(),suerte()])
.then(n=>console.log("Ganamos!!"))
.catch(n=>console.log("Perdimos por: ",n))
```

4
4
6
Perdimos por: 4

6
4
4
Ganamos!!

Promise.resolve Promise.reject

Los métodos *resolve* y *reject* de *Promise* permiten devolver una promesa cumplida o rechazada respectivamente sin necesidad de crear una promesa con *new Promise()*.

Ejemplo:

```
function suerte(){  
  const aleatorio=1+Math.floor(Math.random()*6);  
  console.log(aleatorio);  
  if (aleatorio==6)  
    Promise.resolve(`${aleatorio}`);  
  else  
    Promise.reject(`${aleatorio}`);  
}
```

[async]

La palabra reservada *async*, colocada, en la declaración de función, justo antes de la palabra reservada *function* o justo antes de los paréntesis en un *arrow function*, define la función con un comportamiento asíncrono. Además envolverá lo que la función devuelva en una promesa.

Ejemplo:

```
const funcionAsincrona=async ()=>{  
  const horaActual=Date.now();  
  while(true)  
    if ((horaActual+1000)<Date.now())  
      break;  
  return("Saliendo de la espera");}  
  
console.log('Inicio del código');  
funcionAsincrona().then(console.log);  
console.log('Fin del código');
```

Inicio del código

Fin del código

Saliendo de la espera

[await]

Cualquier función definida con *async*, o lo que es lo mismo, cualquier promesa puede utilizarse junto a la palabra reservada *await* para manejarla. Lo que hace *await* es esperar a que se resuelva la promesa, antes de continuar ejecutando otras tareas. *await* devuelve el sincronismo al código que no lo tiene.

async junto con *await* no es más que una forma de azúcar sintáctico para gestionar las promesas de una forma más sencilla.

En el ejemplo de *async* supongamos que tenemos que sacar en orden los mensajes, esto puede solucionarse con la palabra *await*:

Inicio del código

Saliendo de la espera

Fin del código

```
const funcionAsincrona=async ()=>{  
  const horaActual=Date.now();  
  while(true)  
    if ((horaActual+1000)<Date.now())  
      break;  
  return("Saliendo de la espera");}  
  
console.log('Inicio del código');  
await funcionAsincrona().then(console.log);  
console.log('Fin del código');
```

[async - await]

await, además, obtendrá el valor devuelto por la promesa, no una promesa. Esto hace que la forma de trabajar con *async* y *await* sea mucho más fácil y trivial para usuarios que no están acostumbrados a las promesas y a la asincronía en general, ya que el código parece síncrono.

Compara lo devuelto por los dos siguientes códigos:

```
const funcionAsincrona=async ()=>{
  const horaActual=Date.now();
  while(true)
    if ((horaActual+1000)<Date.now())
      break;
  return("Saliendo de la espera");}
```

```
console.log('Inicio del código');
console.log(funcionAsincrona());
console.log('Fin del código');
```

Inicio del código

► Promise {<fulfilled>: 'Saliendo de la espera'}

Fin del código

```
const funcionAsincrona=async ()=>{
  const horaActual=Date.now();
  while(true)
    if ((horaActual+1000)<Date.now())
      break;
  return("Saliendo de la espera");}
```

```
console.log('Inicio del código');
console.log(await funcionAsincrona());
console.log('Fin del código');
```

Inicio del código

Saliendo de la espera

Fin del código

[for await]

La sentencia *for await* crea un bucle que itera tanto sobre objetos iterables síncronos como asíncronos, como son *Strings*, *Arrays*, *Sets*, *Maps*... Al igual, que *async* y *await*, *for await* permite que trabajar con la asincronía con facilidad, haciendo que el código parece síncrono.

La sintaxis de *for await* es:

```
for await (variable of iterable) {  
    [sentencia...]  
}
```

variable es el valor de un elemento diferente del objeto iterable en cada iteración.

iterable especifica un iterable.

Lo que hace diferente a *for await* de un *for* normal es que puede trabajar con iterables asíncronos, como son un array de promesas, y esperará a que se complete cada uno de ellos antes de pasar a la siguiente iteración.

[for await]

Observa la ejecución de los dos trozos de código:

```
async function suerte(){  
  return(1+Math.floor(Math.random()*6));}  
let i=0  
for(let j=0;j<2;j++)  
  suerte().then(n=>i+=n)  
console.log(i);
```

```
async function suerte(){  
  return(1+Math.floor(Math.random()*6));}  
let i=0  
for await(let f of [suerte,suerte,suerte])  
  suerte().then(n=>i+=n)  
console.log(i);
```

En este primer caso, se podría pensar que se va a obtener un número que es la suma de tres números generados de forma aleatoria, pero siempre se obtiene cero, el valor inicial de la variable *i*.

Esto es debido a que la función que los genera se ejecuta de manera asíncrona y todavía no se ha generado el número cuando se utiliza para realizar la suma:

0

En el segundo caso, el *for await* hace que se procesen de manera síncrona, en cada iteración se espera a que finalice antes de pasar a la siguiente, por lo que se genera el número aleatorio antes de procesar su suma, y se obtiene el resultado esperado:

12

[function*]

La declaración *function**, palabra reservada *function* seguida de un asterisco *, define una función generadora, esto es, una función que devuelve un objeto de tipo *Generator*.

La sintaxis de una función generadora es igual a la de una función normal, solo que lleva un asterisco detrás de la palabra *function*:

```
function* nombre([param[, param[, ... param]]]) {  
    [sentencia...]  
}
```

Las funciones generadoras en JS son funciones en las que su contexto, el valor asociado a las variables, permanece entre distintas invocaciones.

La llamada a una función generadora no ejecuta las instrucciones contenidas en su cuerpo, en su lugar, devuelven un objeto *Generator* de la función.

[generator]

Un objeto *Generator* (generador) es un objeto que no puede ser instanciado directamente, solo se crea cuando se invoca una función generadora.

Un objeto *Generator* es un objeto similar a un iterador, pero en cada iteración ejecuta las sentencias de la función generadora hasta encontrar la sentencia *yield*, en la siguiente iteración continúa a partir de la instrucción que sigue a dicha sentencia *yield*.

Los métodos de un objeto *Generator* son:

- ❑ ***next()*** devuelve un objeto que es creado por la sentencia *yield* de la función generadora.
- ❑ ***return()*** devuelve un objeto que es creado por la sentencia *yield* de la función generadora y finaliza el generador.
- ❑ ***throw()*** lanza un error al generador, también finaliza el generador a menos que sea atrapado desde ese generador.

[generator.next()]

Cuando el método *next()* de un objeto *Generator* es invocado, el cuerpo de la función generadora es ejecutado hasta que encuentre una sentencia *yield*, la cual actúa como una sentencia *return*, esto es, inmediatamente devuelve el valor que se encuentre a su derecha.

El método *next()* devuelve *protocolo iterador*: un objeto *IteratorResult* con una propiedad *value* que contiene el valor devuelto por *yield* y una propiedad *done* que indica, con un booleano, si la función generadora ha alcanzado el último *yield*.

Con carácter opcional, el método *next()* puede recibir un argumento que será accesible por la sentencia *yield*.

El siguiente ejemplo es capaz de mostrar en una ventana *confirm* todos los números pares uno a uno, mediante el uso de una función generadora:

```
function* par(){
  let index=2;
  while (true){
    yield index;
    index+=2;}
}
const iterador=par();
do{}while(confirm(iterador.next().value));
```

[yield]

Como ya se ha comentado, la sentencia *yield* se usa para pausar y reanudar una función generadora.

La sintaxis de *yield* es la siguiente:

```
[valor] = yield [expresion]
```

expresion define el valor que se devolverá desde la función generadora a través del protocolo iterador. Si se omite devuelve *undefined*.

valor recupera el valor opcional pasado al método *next()* del generador para reanudar su ejecución.

[yield*]

Se puede utilizar la sentencia *yield** en lugar de la sentencia *yield* para delegar en otro *Generator* u objeto iterable.

*yield** itera sobre el operador realizando *yield* de cada valor retornado por este.

La sintaxis de *yield** es la siguiente:

```
yield* [[expresion]]
```

expresion define un objeto iterable que será devuelto.

Ejemplo:

```
function* nones(){  
  let inicio=1;  
  while (true){  
    yield `Numero: ${inicio}`;  
    yield* primos(inicio);  
    inicio=inicio+2;}  
}
```

```
function* primos(inicio) {  
  yield `[${inicio.toString().split("")}]`;  
}  
  
const iterador=nonos();  
do{}while(confirm(iterador.next().value));
```

[iterador personalizado]

Se pueden crear iteradores personalizados idénticos a los iteradores que vienen con JS que permita recorrer algún dato iterable.

El funcionamiento de un iterador, como se ha visto, consiste en llamar a una función que crea y devuelve el iterador sobre un objeto iterable concreto.

A continuación, siguiendo el *protocolo iterador*, cada vez que se ejecuta la función *next()* del iterador se obtiene un objeto con una propiedad *value*, que contiene el valor devuelto en esa iteración, y una propiedad *done*, que indica con un booleano si ya no hay más elementos sobre los que iterar.

Cuando no haya más elementos sobre los que iterar se obtendrá como valor *undefined* y propiedad *done* será *true*.

Así se debe crear una función que permita crear y devolver un iterador, el iterador será un objeto con un método, la función *next()*, y cada vez que se invoque devolverá un objeto con las dos propiedades ya expuestas: *value* y *done*.

[iterador personalizado]

Nuestro iterador personalizado va a iterar sobre un array llamado carrito que contendrá objetos producto:

```
class Producto{  
  constructor(nombre){this.nombre=nombre;}}  
  
const carrito=[];
```

Y a continuación el iterador:

```
function iteradorCarrito(carrito){  
  let i=0;  
  return{  
    next:()=>{  
      const done=(i>=carrito.length);  
      const value= !done? carrito[i++]  
                    :undefined;  
      return {done, value};  
    }  
  }  
}
```

[iterador personalizado]

Ejemplo:

```
carrito.push(new Producto("Patatas"));
carrito.push(new Producto("Croquetas"));
carrito.push(new Producto("Jamón"));
console.log(carrito);
```

```
const it=iteradorCarrito(carrito);
console.log(it.next().value);
console.log(it.next().value);
console.log(it.next());
console.log(it.next());
```

```
▼ (3) [Producto, Producto, Producto] ⓘ
  ► 0: Producto {nombre: 'Patatas'}
  ► 1: Producto {nombre: 'Croquetas'}
  ► 2: Producto {nombre: 'Jamón'}
    length: 3
  ► [[Prototype]]: Array(0)
```

```
► Producto {nombre: 'Patatas'}
► Producto {nombre: 'Croquetas'}
► {done: false, value: Producto}
► {done: true, value: undefined}
```


[set]

Un *Set* (conjunto) es objeto que representa un tipo de dato que permite almacenar múltiples elementos de diferentes tipos.

La sintaxis que se utiliza para definir un *Set* es a través de su constructor:

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

Y la forma de representarlos es colocando los elementos separados por comas entre llaves: $\{val1, val2, \dots, valN\}$

Las características que los definen son:

- ☐ La colección de elementos que forman un *Set* no se puede indexar pero sí son iterables.
- ☐ Los elementos que forman un *Set* no puede estar duplicados.
- ☐ Un *Set* es más rápido y eficiente que un objetos cuando se manejan grandes de datos.

[set]

Para añadir elementos al conjunto se utiliza el método *add* de *Set*. Este método añade como elemento al conjunto el argumento que se le pasa como argumento, si el elemento ya estaba en el *Set* no hace nada.

Los *Set* disponen de la propiedad *size* que indica el número de elementos presentes en el conjunto.

Ejemplo:

```
const conjunto = new Set();
conjunto.add("Patatas");
conjunto.add(true);
console.log(conjunto);
console.log(conjunto.size);
```

```
▼ Set(2) {'Patatas', true} ⓘ
  ▼ [[Entries]]
    ► 0: "Patatas"
    ► 1: true
    size: 2
    ► [[Prototype]]: Set
2
```

```
conjunto.forEach(elemento => console.log(elemento));
```

```
Patatas
true
```

[set]

Para eliminar elementos del conjunto se utiliza el método *delete* de *Set*, que elimina del conjunto el elemento pasado como argumento, si el elemento no está en el *Set* no hace nada. Devuelve un booleano indicando si se completó la tarea.

Además, se dispone del método *clear* que elimina todos los elementos presentes en el conjunto.

Otro método útil es *has* que devuelve un booleano indicando si el elemento pasado como argumento existe o no en el conjunto.

Ejemplo:

```
console.log(conjunto);  
conjunto.has("Patatas");  
conjunto.delete("Patatas");  
console.log(conjunto);  
conjunto.clear();  
console.log(conjunto);
```

```
▼ Set(2) {'Patatas', true} ⓘ  
  ▼ [[Entries]]  
    ► 0: "Patatas"  
    ► 1: true  
    size: 2  
    ► [[Prototype]]: Set
```

2

WeakSet

Un *WeakSet* (un conjunto débil) es casi idéntico a un *Set* pero únicamente puede contener objetos.

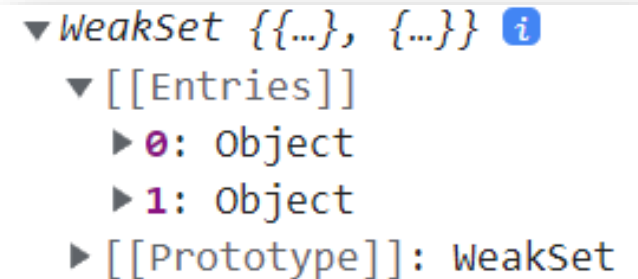
La sintaxis que se utiliza para definir un *WeakSet* es a través de su constructor:

```
const conjunto = new WeakSet();
```

Y la forma de representarlos es colocando los objetos separados por comas entre llaves: *{obj1, obj2, ..., objN}*

Ejemplo:

```
const conjunto = new WeakSet();  
const producto1={nombre:"Patatas"};  
conjunto.add(producto1);  
conjunto.add({nombre:"Jamón"});  
console.Log(conjunto);
```



```
▼ WeakSet {{...}, {...}} ⓘ  
  ▼ [[Entries]]  
    ► 0: Object  
    ► 1: Object  
    ► [[Prototype]]: WeakSet
```

WeakSet

Las diferencias entre un *WeakSet* y un *Set* son:

- ☐ Solo pueden contener objetos.
- ☐ No tiene la propiedad *size*.
- ☐ No son iterables.

Ejemplo:

```
const conjunto = new WeakSet();  
conjunto.add({nombre: "Patatas"});  
console.log(conjunto.size);
```

undefined

[map]

Un *Map* (mapa, en otros lenguajes también los llaman diccionarios) es un objeto que almacena parejas de elementos compuestos por una clave y un valor, pero la clave y el valor pueden ser cualquier tipo de dato, pero la clave debe ser única dentro del *Map*, no puede haber dos o más claves iguales.

La sintaxis que se utiliza para definir un *Map* es a través de su constructor:

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

Y la forma de representarlos es colocando los elementos, *clave* => *valor*, separados por comas y entre llaves: {*clave1* => *val1*, *clave2* => *val2*, ..., *claveN* => *valN*}

Las características que los definen son:

- ☐ A los elementos de un *Map* se accede por su clave y sí son iterables.
- ☐ Dos o más elementos de un *Map* no pueden tener la misma clave.
- ☐ Un *Map* tiene mejor rendimiento que los objetos cuando se manejan grandes de datos.

[map]

Para añadir elementos a un *Map* se utiliza el método *set*. Este método recibe dos argumentos, el primero es la clave y el segundo es el valor, que se añaden como elemento al Mapa, si el elemento ya estaba en el *Map* modifica dicho elemento con el nuevo valor.

Los *Map* disponen de la propiedad *size* que indica el número de elementos que contiene.

Ejemplo:

```
const miMap= new Map();  
miMap.set("Nombre", "Patatas");  
miMap.set("Precio", 10);  
console.log(miMap);  
console.log(miMap.size);
```

```
► Map(2) {'Nombre' => 'Patatas', 'Precio' => 10}  
2
```

[map]

Para acceder a los distintos elementos del *Map* se utiliza su clave junto con el método *get*, que recupera el valor del elemento que se corresponde con la clave que se le pasa como argumento.

Dispone del método *has* que devuelve un booleano indicando si la clave del elemento pasado como argumento existe o no en el *Map*.

Para eliminar elementos se utiliza el método *delete*, que elimina del *Map* el elemento que se corresponde la clave que se pasa como argumento, si el elemento no está en el *Map* no hace nada. Devuelve un booleano indicando si se completó la tarea.

Ejemplo:

```
console.log(miMap.get("Nombre"));  
console.log(miMap.has("Nombre"));  
miMap.delete("Nombre");  
console.log(miMap);
```

```
Croquetas  
true  
► Map(1) {'Precio' => 10}
```


[map]

Dispone del método *clear* que elimina todos los elementos presentes en el *Map*.

A través del constructor, en la creación del *Map* se puede inicializar con datos pasándolos como un *Array* de *Arrays* de 2 elementos, la clave y el valor.

Ejemplo:

```
const miMap= new Map([["Nombre", "Patatas"], ["Precio", 10]]);  
console.log(miMap);  
miMap.clear();  
console.log(miMap);
```

```
▼ Map(2) {'Nombre' => 'Patatas', 'Precio' => 10}  
  ▼ [[Entries]]  
    No hay ninguna propiedad  
    size: 0  
    ► [[Prototype]]: Map  
  ► Map(0) {size: 0}
```

[map]

Un *Map* se puede iterar.

Ejemplo:

```
const miMap= new Map([["Nombre", "Patatas"], ["Precio", 10]]);  
miMap.forEach(elemento=>console.log(elemento));
```

Patatas
10

```
const miMap= new Map([["Nombre", "Patatas"], ["Precio", 10]]);  
miMap.forEach((elemento, index)=>console.log(index, ":", elemento));
```

Nombre : Patatas
Precio : 10

WeakMap

Un *WeakMap* (un *Map* débil) es parecido a un *Map* pero únicamente puede contener como claves objetos, así, sus elementos están compuestos de una clave, un objeto, y un valor asociado a la misma.

La sintaxis que se utiliza para definir un *WeakMap* es a través de su constructor:

```
const weakMap = new WeakMap();
```

Y la forma de representarlos es colocando los elementos separados por comas y entre llaves: $\{obj1 \Rightarrow val1, obj2 \Rightarrow val2, \dots, objN \Rightarrow valN\}$

Ejemplo:

```
const weakMap = new WeakMap();  
const producto1={nombre:"Patatas"};  
weakMap.set(producto1,"Entrante");  
weakMap.set({nombre:"Jamón"},"Entrante");  
console.log(weakMap);
```

```
▼ WeakMap {[...]} => 'Entrante', {[...]} => 'Entrante'  
  ▼ [[Entries]]  
    ► 0: {Object => "Entrante"}  
    ► 1: {Object => "Entrante"}  
    ► [[Prototype]]: WeakMap
```

WeakMap

Las diferencias entre un *WeakMap* y un *Map* son:

- ❑ Las claves solo pueden ser objetos.
- ❑ No tiene la propiedad *size*.
- ❑ No son iterables.

Ejemplo:

```
const weakMap = new WeakMap();  
const producto1={nombre:"Patatas"};  
weakMap.set(producto1, "Entrante");  
console.Log(weakMap.has(producto1));  
console.Log(weakMap.get(producto1));
```

true

Entrante

[symbols]

Un *Symbol* (símbolo) permiten crear variables únicas.

No hay dos símbolos iguales.

La sintaxis que se utiliza para definir un *Symbol* es invocando la función *Symbol*:

```
const unico = Symbol();
```

Ejemplo:

```
const unico = Symbol ();  
const unico2 = Symbol ();  
console.log(unico2 == unico);
```

false

Los elementos *Symbol* no son iterables, por lo que si se crean propiedades de tipo *Symbol* en un objeto, no están disponibles en las iteraciones.

[symbols]

Además, para acceder a las propiedades de tipo *Symbol* en un objeto se deben utilizar los corchetes, el operador punto `.` no funciona.

Ejemplo:

```
const nombre=Symbol();  
const id=Symbol();  
  
const producto={};  
producto[nombre]="Patatas";  
producto[id]=1;  
producto.precio=10;  
  
console.log(producto);  
console.log(producto[nombre]);  
console.log(producto.nombre);
```

```
► {precio: 10, Symbol(): 'Patatas', Symbol(): 1}  
Patatas  
undefined
```

[symbols]

Cuando se crea un *Symbol* a través de su argumento se le puede pasar cualquier tipo de dato, aunque normalmente se corresponde con una cadena de texto a modo de descripción.

Ejemplo:

```
const nombre=Symbol("Nombre del producto");  
  
const producto={};  
producto[nombre]="Patatas";  
  
console.log(producto);  
console.log(nombre);
```

```
► {Symbol(Nombre del producto): 'Patatas'}  
Symbol(Nombre del producto)
```

Bibliografía y recursos online

- [https://developer.mozilla.org/es/docs/Glossary/Callback function](https://developer.mozilla.org/es/docs/Glossary/Callback_function)
- <https://lenguajejs.com/javascript/asincronia/callbacks/>
- [https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using promises](https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using_promises)
- <https://lenguajejs.com/javascript/asincronia/promesas/>
- <https://lenguajejs.com/javascript/asincronia/promise-api/>

Bibliografía y recursos online

- https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/async_function
- https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/function*
- <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/yield>