



**let a = [3, 21, 15, 61, 9, 54];
console.log(a.every(function(num) { // Comprueba
si cada número es menor a 100 return num < 100;
})); / cómo sabe que num es cada valor del array a?**

En JavaScript, cuando usas el método `every` en un array, por ejemplo:

```
let a = [3, 21, 15, 61, 9, 54];
console.log(a.every(function(num) {
    return num < 100;
}));
```

El método `every` ejecuta la función que le pasas (llamada función `callback`) para *cada elemento* del array. El parámetro `num` representa el valor del elemento actual del array que se está evaluando en cada iteración.

Esto ocurre porque `every` internamente recorre el array y en cada paso toma un elemento, se lo pasa a la función como argumento y evalúa si cumple la condición.

En resumen:

- `num` es solo un nombre de variable para ese parámetro de la función que representa el valor actual del array.
- `every` llama a esa función tantas veces como elementos tenga el array, cada vez con un valor distinto.
- Así sabe que `num` es cada valor del array porque lo hace automáticamente para ti al iterar el array.

Esto es un comportamiento estándar de muchos métodos de array en JavaScript, que usan funciones `callback` con parámetros para representar los valores que procesan en las iteraciones.
[\[1\]](#) [\[2\]](#) [\[3\]](#) [\[4\]](#)

**

¿Qué es una función `callback` en JavaScript?

Una **función `callback`** es una función que se pasa **como argumento a otra función** para que esta última la llame *después* o *durante* su ejecución. Es decir, en lugar de ejecutar una función y ya, pasamos una función a otra para que sea llamada cuando se necesite.

¿Por qué usamos callbacks?

Imagina que tienes una tarea con varios pasos, y quieres decirle al programa exactamente qué hacer cuando termine un paso, sin detener todo el programa mientras espera. Para eso, pasas una función callback: la función que quieras que se ejecute justo después.

En los métodos como `array.every()` la función callback se ejecuta para cada elemento del array y reciben valores como el elemento actual, el índice, etc. Así, puedes definir la condición que quieras aplicar a cada elemento.

Sintaxis básica de una función callback

```
// Esta función recibe otra función (callback) como parámetro
definirFuncion(callback) {
    // Al hacer esto, llamamos a la función pasada
    callback();
}

// Crear una función para pasar como callback
function saludar() {
    console.log('Hola!');
}

// Llamar a definirFuncion pasando la función saludar
definirFuncion(saludar); // Esto imprimirá 'Hola!'
```

También puedes usar funciones anónimas (sin nombre) directamente al llamar:

```
definirFuncion(function() {
    console.log('Saludo desde callback anónima');
});
```

O de forma moderna con funciones flecha:

```
definirFuncion(() => {
    console.log('Hola desde función flecha');
});
```

Resumen rápido

- Una función callback es *una función que se pasa a otra función para ser llamada dentro de ella*.
- Se usa para definir acciones que suceden **después** o **durante** el trabajo de otra función.
- Es clave en métodos de arrays como `.every()`, `.forEach()`, y para programar eventos o tareas que tardan (como esperas).

¿Quieres que te ayude a practicar con un ejemplo sencillo de callback? ¿O prefieres que te explique cómo funcionan en métodos de arrays con ejercicios?

**

No lo entiendo En every veo que sí pasas la función como argumento. ¿Pero dónde la llamas?

En el caso de `every`, tú le **pasas** la función callback como argumento:

```
a.every(function(num) {  
    return num < 100;  
});
```

Pero **quién la llama** es internamente el método `every` que está definido en JavaScript. No la llamas tú directamente, sino que ese método:

1. Recorre el array elemento por elemento.
2. Para cada elemento, llama a la función que tú pasaste (la callback), pasándole el elemento actual como argumento (`num`).
3. Espera a que la función le devuelva `true` o `false`.
4. Si la función devuelve `false` para algún elemento, detiene la iteración y devuelve `false` como resultado.
5. Si todas las llamadas devuelven `true`, devuelve `true`.

Es decir, la función callback la llama automáticamente el propio método `every` para poder comprobar la condición que le definiste.

Esto es parte de cómo funcionan todos los métodos que usan callbacks en arrays (como `.forEach()`, `.map()`, `.filter()`, etc.). El método recorre el array y llama a la función que le diste, para decidir qué hacer con cada valor.

Por eso tú sólo escribes la función, y `every` la invoca internamente en su bucle (no tienes que invocarla tú manualmente).

Ejemplo simple para ilustrar esto:

```
function llamarCallbackParaCadaElemento(array, callback) {  
    for (let i = 0; i < array.length; i++) {  
        callback(array[i]); // Aquí se llama a la función que pasamos  
    }  
}  
  
llamarCallbackParaCadaElemento([1,2,3], function(x) {  
    console.log(x); // Imprime cada número  
});
```

En resumen:

- La función que pasas a `every` es una callback porque es llamada por otro código (`every`) durante su ejecución.
- Tú defines qué hace la función, pero no la llamas tú, la llama `every` internamente para cada elemento.[\[22\]](#) [\[23\]](#) [\[24\]](#)

Función callback vs bucle for en JavaScript

¿Qué haces cuando escribes una callback?

Cuando escribes una función callback en JavaScript, estás definiendo una función que **vas a pasar como argumento a otra función** para que esta última la **ejecute en algún momento**. No la ejecutas tú directamente, sino que la función que recibe la callback será la que la llame — generalmente muchas veces, una por cada elemento o evento.

Por ejemplo, en un método de array como `forEach()` o `every()`, pasas una función callback que define qué hacer con cada elemento del array. Este método recorre el array internamente y ejecuta la callback en cada paso.

¿En qué se diferencia de un bucle for?

- Un bucle `for` es una instrucción explícita que tú controlas paso por paso para recorrer un array o repetir código.
- Cuando usas una función callback con métodos como `.forEach()`, **delegas esa iteración** al método, y solo defines el "qué hacer" con cada elemento.

Ventajas de usar callbacks en vez de bucles for manuales:

- Código más corto y claro.
- El método controla la iteración y evita errores comunes en bucles (como índices fuera de rango).
- Más fácil de expresar lógica con funciones, favoreciendo un estilo funcional.

Ejemplo paralelo

```
const array = [1, 2, 3];

// Usando un bucle for
for (let i = 0; i < array.length; i++) {
  console.log(array[i]);
}

// Usando un callback con forEach
array.forEach(function(element) {
```

```
    console.log(element);
});
```

En ambos casos imprime cada número. La diferencia está en quién controla la repetición:

- En `for`, tú lo escribes.
- En `forEach`, lo hace JavaScript, tú solo defines la función a ejecutar.

¿Quieres que te ayude a practicar con ejemplos para entender mejor la diferencia o a escribir tus propias funciones callback?

**

cuando en el ejemplo `forEach` pongo `function(element)` cómo sabe javascript que tiene que leer el array?

Cuando usas `forEach` y pones `function(element)`, JavaScript **sabe automáticamente** que debe leer el array en ese momento porque ese método está diseñado para recorrer cada elemento del array y ejecutar la función callback en cada uno de esos elementos. [\[41\]](#) [\[42\]](#)

¿Cómo funciona?

- Cuando tú escribes `array.forEach(function(element) { ... })`, lo que pasa es que `forEach`:
 1. **Recorre** el array internamente.
 2. En cada paso, **lee** un elemento del array.
 3. **Llama a la función callback** que le pasaste, **pasándole ese elemento** como argumento (en ese caso, como `element`).

¿Dónde está esa acción de leer el array?

- La acción de leer o recorrer el array la realiza automáticamente `forEach`.
- Tú solo defines qué hacer con cada elemento en la función callback, y `forEach` se encarga de llamar a la función con el elemento actual en cada iteración.

Resumen sencillo:

- Tú pasas la función con `function(element){...}`.
- `forEach` internamente **lee** cada elemento del array y **ejecuta** esa función, enviándole como argumento el elemento que está leyendo en ese momento.

¿Quieres que te explique con un ejemplo paso a paso?

**

Funciones sencillas en JavaScript

Una función es un bloque de código que puedes reutilizar para hacer tareas específicas. La defines una vez y la llamas cuando la necesitas.

Ejemplo básico de función que suma dos números:

```
function sumar(num1, num2) {  
    return num1 + num2;  
}  
  
console.log(sumar(3, 5)); // Imprime 8
```

Aquí:

- `function` es la palabra que usamos para declarar la función.
- `sumar` es el nombre de la función.
- `(num1, num2)` son los parámetros, valores que recibe la función.
- `return` entrega el resultado de sumar esos dos números.

Puedes llamar `sumar` con distintos números para obtener la suma sin repetir el código.

¿Qué es una función callback?

Una función callback es una función que **se pasa como argumento** a otra función para que esta última la **ejecute dentro de su proceso**. Esto permite que puedas personalizar lo que sucede en cada paso de otra función, por ejemplo, mientras recorre un array.

Ejemplo con función callback: recorrer un array y hacer algo con cada elemento.

```
let numeros = [1, 2, 3];  
  
function procesarArray(arr, callback) {  
    for(let i = 0; i < arr.length; i++) {  
        callback(arr[i]); // Aquí llamamos la función pasada, con el elemento actual  
    }  
}  
  
// Pasamos una función que imprime el número  
procesarArray(numeros, function(numero) {  
    console.log(numero);  
});
```

Aquí:

- `procesarArray` es una función que recibe un array y una **callback**.
- La **callback** es una función que va a ejecutarse para cada elemento.
- Dentro de `procesarArray` usamos un bucle `for` y en cada paso llamamos `callback(arr[i])`.

Esto permite separar:

- La repetición (hecha dentro de `procesarArray`)
- La acción específica (definida en la callback que se pasa)

¿Por qué usamos callbacks y cómo funciona internamente?

- Usamos callbacks para hacer que funciones puedan **personalizar su comportamiento** recibiendo otra función que se ejecuta cuando se necesite.
- Internamente, la función que recibe la callback la **llama** en el momento que quiera, por ejemplo, en cada iteración o cuando ocurre un evento.
- Así, el flujo de ejecución es: tu defines la callback (la acción) y la función que la recibe (como `forEach`, `map`, o una función propia) controla cuándo y cómo ejecutarla.

Ejemplo rápido:

```
// Función que ejecuta la callback con el mensaje "Hola"
function saludar(callback) {
  callback("Hola");
}

// Llamamos a saludar pasando una función que recibe y muestra el mensaje
saludar(function(mensaje) {
  console.log(mensaje);
});
// Imprime: Hola
```

¿Qué te parece si intentamos ahora juntos crear una función con callback? ¿Quieres que te guíe paso a paso?

**

Tu reto: Crear un ejemplo sencillo con función callback

Quiero que escribas una función llamada `procesarTexto` que reciba dos cosas:

1. Un texto (por ejemplo, una cadena de caracteres).
2. Una función callback que procese ese texto de alguna manera (por ejemplo, imprimirlo en mayúsculas, en minúsculas, o mostrar su longitud).

Dentro de `procesarTexto`, debes llamar a la función callback y pasarle el texto para que ejecute la acción que definiste.

Luego, llama a `procesarTexto` pasando:

- Un texto cualquiera.
- Y una función callback (puede ser una función anónima o nombrada) que realice alguna operación con ese texto y muestre el resultado.

Importante: No te preocupes por la solución aún, solo intenta escribir esta función y cómo llamarla con una callback.

Cuando termines, dime qué hiciste y vemos juntos si quieras mejorar o entender más la lógica.

**

```
function procesarTexto(txt,callback) {
    callback(txt.toUpperCase());} let textito="el
perro"; procesarTexto(textito, function(lectura){
    console.log(lectura);});
```

¡Muy bien! Has escrito una función callback correctamente:

```
function procesarTexto(txt, callback) {
  callback(txt.toUpperCase());
}

let textito = "el perro";
procesarTexto(textito, function(lectura) {
  console.log(lectura);
});
```

Esto hace que:

- `procesarTexto` reciba un texto `txt` y una función `callback`.
- Dentro de `procesarTexto`, la llamas con `callback(...)`, pasándole `txt.toUpperCase()`, es decir, el texto en mayúsculas.
- Al llamar `procesarTexto`, pasas la función anónima que recibe `lectura` y la muestra en consola.

Así, el programa imprime "EL PERRO".

¿Quieres que analicemos juntos cada parte para entender bien la lógica y la forma de usar callbacks? O prefieres probar otro ejemplo con diferente proceso dentro del callback?

**

Sí, porque en la primera parte entiendo que le indico que le meto un texto y una callback y defino la callback. Y luego llamo a la función primera, le meto el texto que quiero que procese pero me cuesta entender que uso una función nueva con la que defino que hacer con todo lo que he hecho, puesto que imprimo lectura. esa parte me cuesta de entender. ayúdame

Vamos a aclarar esa parte que te cuesta entender paso a paso, poniendo todo muy claro y sencillo:

1. Definir la función callback que hará algo con el resultado

Cuando haces esto:

```
procesarTexto(textito, function(lectura) {  
    console.log(lectura);  
});
```

Estás **definiendo una función nueva anónima** (sin nombre) que recibe un parámetro, lectura, y luego hace algo con él (en este caso, imprimirlo).

Esta función no se ejecuta inmediatamente aquí, sino que se pasa como argumento para que alguien más la use.

2. Pasar la función a otra función que la va a llamar

Dentro de procesarTexto, tienes esta línea:

```
callback(txt.toUpperCase());
```

Aquí **es donde se llama o se ejecuta tu función callback** y se le pasa como parámetro el texto transformado a mayúsculas.

Eso quiere decir que la función anónima que escribiste recibe ese texto (que llega en el parámetro lectura), y entonces ejecuta el código dentro: imprimir en consola.

3. Por qué usamos esta forma

- Pasas la función que define qué hacer con el texto (callback) para que la función procesarTexto pueda ejecutar esa acción cuando termine su trabajo.
- Separa qué hacer con el texto del proceso de transformar texto.

- Da flexibilidad para que `procesarTexto` no tenga que saber cómo usar el resultado, solo se lo entrega a quien definió la función callback.

En resumen:

- La función callback es solo una pequeña receta que dices: "Cuando me des el texto procesado, imprime esto".
- La función externa (`procesarTexto`) es quien efectivamente ejecuta esa receta cuando tenga el dato listo.

Si lo ves como una cadena:

Yo (usuario) hago una receta (callback) y se la doy al cocinero (función `procesarTexto`) para que cuando termine de preparar la comida (transformar el texto), use la receta (ejecute la callback).

¿Quieres que hagamos juntos otro ejemplo similar y lo veamos línea por línea para que quede clarísimo? ¿O prefieres que te ayude con alguna metáfora o dibujo mental para entender mejor esta lógica de "dar la receta para que la ejecuten"?

**

creo que tu ejemplo de la receta no es muy válido, puesto que al cocinero primero se le da la receta y luego prepara la comida y no al revés

Tienes razón, la metáfora de la receta al revés puede confundir. Mejor pongamos una metáfora más adecuada para entender cómo funcionan las funciones callback.

Metáfora mejor para la función callback: El mensajero y el encargo

Imagina que tú das un encargo (una tarea o mensaje) a un mensajero y le das también instrucciones específicas sobre qué hacer cuando entregue ese encargo.

- Tú (programador) defines las instrucciones (función callback).
- El mensajero (función que recibe la callback) lleva el encargo y cuando lo entrega, sigue esas instrucciones.

Es decir:

1. Tú defines qué instrucciones quieres que se sigan (la función callback).
2. La función principal hace su trabajo (por ejemplo, procesa un texto).
3. Cuando termina, llama a la función callback para que ejecute esas instrucciones con el resultado (como el mensajero que sigue tus instrucciones al entregar el encargo).

En código:

- La función principal es el mensajero.
- La función callback son tus instrucciones para el mensajero al entregar el encargo.

```
function mensajero(encargo, instrucciones) {  
    // procesa el encargo  
    let resultado = encargo.toUpperCase();  
    // cuando termina, sigue las instrucciones  
    instrucciones(resultado);  
}  
  
// Definimos instrucciones como función callback  
function instruccionesAsegurar(texto) {  
    console.log('Resultado recibido:', texto);  
}  
  
// Llamamos al mensajero con encargo e instrucciones  
mensajero('hola', instruccionesAsegurar);
```

El "mensajero" solo hace su trabajo y después sigue las instrucciones que le diste, que pueden ser cambiar mayúsculas, imprimir, guardar, etc.

¿Quieres un ejemplo más simple o prefieres que te explique con algún dibujo mental? También podemos probar otro ejercicio para afianzar esta idea.

**

Orden de ejecución en un programa con callback

Aunque el código esté escrito de arriba hacia abajo, el orden que sigue el programa al ejecutarlo es este:

1. **Se define la función principal** con su parámetro callback, pero no se ejecuta todavía.
2. **Se llama a la función principal** con los argumentos que pases (por ejemplo, un texto y una función callback).
3. Dentro de esa función principal, cuando llega a la instrucción que dice `callback(...)`, **se ejecuta la función callback que se le pasó como argumento**, en ese momento y con los valores que se le proporcionen.
4. La ejecución vuelve a la función principal y continúa (si hay más código).

Es decir, la función callback **se ejecuta cuando la función principal la invoca**, no antes.

¿Siempre son dos parámetros en un callback?

No, **no siempre hay dos parámetros en una función callback**. El número de parámetros de la función callback depende del método o función que la llame y le pase datos.

Por ejemplo:

- En `array.forEach((element) => {})`, el callback recibe 1 parámetro: el elemento actual.
- En `array.forEach((element, index) => {})`, el callback tiene 2 parámetros: el elemento y su índice.
- En `setTimeout(() => {}, 2000)`, el callback no recibe ningún parámetro.

Cada función o método que use callbacks define qué argumentos le pasa a esa callback y cuántos. Tú defines la función con esos parámetros para recibir la información que necesites.

¿Quieres que veamos un ejemplo concreto para practicar el orden de ejecución y ver cuántos parámetros puede tener una callback en distintos casos?

**

Sí, ponme un ejemplo concreto para practicar el orden de ejecución, evita imágenes mentales como cocinero o mensajero pues no me ayudan. y céntrate en explicarme como al principio, línea por línea y el orden del programa

Vamos a analizar cómo funciona el orden de ejecución en un ejemplo paso a paso:

```
function procesarTexto(txt, callback) {
  // Línea A
  callback(txt.toUpperCase()); // Línea B
}

let textito = "el perro";

// Línea C
procesarTexto(textito, function(lectura) {
  // Línea D
  console.log(lectura); // Línea E
});
```

Ahora, ordenamos la ejecución desde que el programa empieza:

1. **Primero**, el programa encuentra la definición de la función `procesarTexto` (línea A y B). Es decir, la crea en memoria, pero no la ejecuta todavía.
2. **Luego**, define la variable `textito` con el valor `

**

me falta el resto de la resolución. sólo veo el paso 1.

Claro, continúo explicando el orden completo paso a paso:

```
function procesarTexto(txt, callback) {  
    // Línea A  
    callback(txt.toUpperCase()); // Línea B  
}  
  
let textito = "el perro";  
  
// Línea C  
procesarTexto(textito, function(lectura) {  
    // Línea D  
    console.log(lectura); // Línea E  
});
```

Orden de ejecución línea por línea

1. Se **define** la función procesarTexto (líneas A y B), pero no se ejecuta aún.
2. Se **define** la variable textito con el valor "el perro".
3. Se llega a la línea C, donde se **llama** a procesarTexto con dos argumentos:
 - textito = "el perro"
 - Una función anónima (callback) que recibe un parámetro lectura y luego imprime su valor (líneas D y E).
4. El programa entra a la función procesarTexto con los valores recibidos.
5. En la línea B:
 - La función callback (anónima) se **ejecuta**.
 - Se le pasa como argumento el texto procesado: txt.toUpperCase() que es "EL PERRO".
6. El programa ejecuta la función callback definida en la llamada, por eso entra en la línea D.
7. En la línea E:
 - Se imprime en consola el texto "EL PERRO".
8. El programa termina la ejecución de la callback, luego termina procesarTexto y sigue.

Resumen del flujo:

- Se **define** la función y la variable.
- Se **llama** a la función principal pasando la función callback.
- La función principal **procesa algo y ejecuta la callback pasándole el resultado**.
- La callback **realiza la acción final** (aquí, imprimir).

Sobre los parámetros en callbacks

No siempre son dos. Depende de la función que llame a la callback. Puede ser 0, 1, 2 o más parámetros.

Por ejemplo:

- .forEach pasa 3 argumentos (elemento, índice y array).
- .setTimeout no pasa ninguno.
- Tu función puede definir y usar los que necesite.

¿Quieres que lo practiquemos con otro ejemplo o que te explique otro detalle?

1. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/every
2. https://www.w3schools.com/jsref/jsref_every.asp
3. https://www.w3schools.com/js/js_array_iteration.asp
4. <https://4geeks.com/how-to/javascript-array-every>
5. <https://flexiple.com/javascript/javascript-loop-array>
6. <https://javascript.info/array-methods>
7. https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/every
8. <https://refine.dev/blog/javascript-every-method/>
9. https://eloquentjs-es.thedojo.mx/Eloquent_JavaScript.pdf
10. <https://stackoverflow.com/questions/70309953/javascript-every-on-array-of-object-doesnt-loop-through-all-the-array>
11. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach
12. https://developer.mozilla.org/es/docs/Glossary/Callback_function
13. <https://desarrolloweb.com/articulos/funciones-callback-javascript>
14. <https://lenguajejs.com/asincronia/callbacks/que-son/>
15. <https://www.youtube.com/watch?v=kjCH7vvISsE>
16. <https://www.freecodecamp.org/espanol/news/que-es-una-funcion-callback-javascript/>
17. <https://antonioweb.dev/articulo/funcion-callback-javascript-guia-completa>
18. <https://www.freecodecamp.org/espanol/news/funciones-callback-en-javascript-que-son-los-callback-en-js-y-como-usarlos/>
19. <https://www.youtube.com/watch?v=EofriyLQohk>
20. <https://jscodez.com/javascript-funciones-de-callback/>
21. <https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/que-es-un-callback/>
22. https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/every
23. <https://jscodez.com/javascript-metodos-some-every/>
24. <https://platzi.com/cursos/javascript-30-dias/metodos-de-arrays-every-find-y-findindex/>
25. https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Iterators_and_generators
26. https://www_escuelafrontend_com/metodos-de-arreglos

27. <https://www.youtube.com/shorts/VVLbP-kU7Xk>
28. <https://es.javascript.info/array-methods>
29. <https://es.stackoverflow.com/questions/570396/metodo-every-para-array-en-javascript>
30. <https://www.youtube.com/watch?v=37RiYprt1CU>
31. <https://keepcoding.io/blog/metodos-some-y-every-en-javascript/>
32. <https://lenguajejs.com/asincronia/callbacks/que-son/>
33. https://developer.mozilla.org/es/docs/Glossary/Callback_function
34. <https://www.freecodecamp.org/espanol/news/funciones-callback-en-javascript-que-son-los-callback-en-js-y-como-usarlos/>
35. <https://www.youtube.com/watch?v=kjCH7vvISsE>
36. <https://www.freecodecamp.org/espanol/news/que-es-una-funcion-callback-javascript/>
37. <https://antonioweb.dev/articulo/funcion-callback-javascript-guia-completa>
38. <https://keepcoding.io/blog/que-son-los-callbacks-en-javascript/>
39. <https://www.youtube.com/watch?v=EofriyLQohk>
40. <https://jscodez.com/javascript-funciones-de-callback/>
41. https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/every
42. https://developer.mozilla.org/es/docs/Glossary/Callback_function
43. https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach
44. <https://desarrolloweb.com/articulos/recorridos-foreach-array-javascript>
45. <https://4geeks.com/es/how-to/metodo-foreach-javascript>
46. <https://www.acontracorrientech.com/guia-practica-sobre-las-callback-functions/>
47. <https://lenguajejs.com/asincronia/callbacks/que-son/>
48. <https://www.freecodecamp.org/espanol/news/foreach-en-javascript-como-recorrer-un-arreglo-en-js/>
49. <https://jscodez.com/javascript-funciones-de-callback/>
50. [https://studyeeasy.org/es/course-articles/javascript-articles/foreach-methods-es/](https://studyeasy.org/es/course-articles/javascript-articles/foreach-methods-es/)
51. <https://www.meltstudio.co/post/como-usar-los-metodos-map-y-foreach-hacks-de-javascript-meltstudi o>
52. [https://es.stackoverflow.com/questions/306947/foreach-javascript-cómo-funciona](https://es.stackoverflow.com/questions/306947/foreach-javascript-c%c3%b3mo-funciona)
53. <https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Functions>
54. <https://es.javascript.info/function-basics>
55. <https://trazos.net/blog-como-hacer-funciones-en-javascript/>
56. <https://lenguajejs.com/javascript/funciones/que-son/>
57. <https://www.youtube.com/watch?v=1BOwGU2Zj-o>
58. <https://www.aluracursos.com/blog/articulo-funciones-en-javascript->
59. <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions>
60. <https://sidmac.com/blog/funciones-en-javascript-ejemplos-practicos-y-explicaciones>
61. <https://somospnt.com/blog/274-tipos-de-funciones-en-javascript>
62. https://developer.mozilla.org/es/docs/Glossary/Callback_function
63. <https://certidevs.com/ejercicios-javascript-callbacks-multirespuesta>

64. <https://lenguajejs.com/asincronia/callbacks/que-son/>
65. <https://desarrolloweb.com/articulos/funciones-callback-javascript>
66. <https://www.freecodecamp.org/espanol/news/que-es-una-funcion-callback-javascript/>
67. <https://www.youtube.com/watch?v=EofriyLQohk>
68. <https://antonioweb.dev/articulo/funcion-callback-javascript-guia-completa>
69. <https://jscodez.com/javascript-funciones-de-callback/>
70. <https://es.javascript.info/callbacks>
71. <https://www.freecodecamp.org/espanol/news/funciones-callback-en-javascript-que-son-los-callback-en-js-y-como-usarlos/>
72. <https://desarrolloweb.com/articulos/funciones-callback-javascript>
73. <https://somospnt.com/blog/208-callback-hell>
74. <https://www.freecodecamp.org/espanol/news/funciones-callback-en-javascript-que-son-los-callback-en-js-y-como-usarlos/>
75. <https://www.aprendejavascript.dev/clase/programacion-asincrona/callbacks>
76. <https://lenguajejs.com/javascript/funciones/callbacks/>
77. <https://www.youtube.com/watch?v=EofriyLQohk>
78. <https://jscodez.com/javascript-funciones-de-callback/>
79. <https://mrebi.com/es/javascript/js-callbacks/>
80. <https://www.acontracorrientech.com/guia-practica-sobre-las-callback-functions/>
81. <https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/que-es-un-callback/>
82. <https://lenguajejs.com/asincronia/callbacks/que-son/>
83. <https://keepcoding.io/blog/que-son-los-callbacks-en-javascript>
84. https://developer.mozilla.org/es/docs/Glossary/Callback_function
85. <https://www.youtube.com/watch?v=kjCH7vvISsE>
86. <https://www.freecodecamp.org/espanol/news/que-es-una-funcion-callback-javascript/>
87. <https://antonioweb.dev/articulo/funcion-callback-javascript-guia-completa>
88. <https://www.freecodecamp.org/espanol/news/funciones-callback-en-javascript-que-son-los-callback-en-js-y-como-usarlos/>
89. <https://www.youtube.com/watch?v=EofriyLQohk>
90. <https://jscodez.com/javascript-funciones-de-callback/>
91. <https://desarrolloweb.com/articulos/funciones-callback-javascript>
92. <https://mrebi.com/es/javascript/js-callbacks/>
93. <https://lenguajejs.com/javascript/funciones/callbacks/>
94. <https://jscodez.com/javascript-funciones-de-callback/>
95. <https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/que-es-un-callback/>
96. https://developer.mozilla.org/es/docs/Glossary/Callback_function
97. <https://thecodest.co/es/diccionario/funcion-callback/>
98. <https://bahiaxip.com/entrada/callbacks>

99. <https://www.studysmarter.es/resumenes/ciencias-de-la-computacion/programacion-de-computadoras/callback-de-javascript/>
100. <https://www.freecodecamp.org/espanol/news/que-es-una-funcion-callback-javascript/>
101. <https://lenguajejs.com/asincronia/callbacks/que-son/>
102. <https://www.freecodecamp.org/espanol/news/funciones-callback-en-javascript-que-son-los-callback-en-js-y-como-usarlos/>
103. <https://antonioweb.dev/articulo/funcion-callback-javascript-guia-completa>
104. <https://jscodez.com/javascript-funciones-de-callback/>
105. <http://developinginspanish.com/2019/03/18/javascript-que-demonios-es-un-callback/>
106. <https://www.freecodecamp.org/espanol/news/que-es-una-funcion-callback-javascript/>
107. <https://es.stackoverflow.com/questions/239811/ejecutar-funciones-de-manera-secuencial-javascript>
108. https://developer.mozilla.org/es/docs/Glossary/Callback_function
109. <https://lenguajejs.com/javascript/funciones/callbacks/>
110. <https://www.freecodecamp.org/espanol/news/que-es-una-funcion-callback-javascript/>
111. <https://antonioweb.dev/articulo/funcion-callback-javascript-guia-completa>
112. <https://www.youtube.com/watch?v=kjCH7vvISsE>
113. <https://desarrolloweb.com/articulos/funciones-callback-javascript>
114. <https://www.freecodecamp.org/espanol/news/funciones-callback-en-javascript-que-son-los-callback-en-js-y-como-usarlos/>
115. <https://www.aprendejavascript.dev/clase/programacion-asincrona/callbacks>
116. <https://lenguajejs.com/javascript/funciones/callbacks/>
117. <https://jscodez.com/javascript-funciones-de-callback/>
118. https://developer.mozilla.org/es/docs/Glossary/Callback_function