



Explicame esto

La nota del margen dice:

"En las arrow functions, el 'this' refiere a 'window'".

Esto significa que cuando se usan funciones flecha (arrow functions) en JavaScript, el valor de la palabra clave `this` no se comporta igual que en las funciones tradicionales. En una función flecha, `this` no toma el valor del objeto desde donde es llamada, sino que hereda el valor de `this` del contexto en el que fue definida. En el ámbito global del navegador, ese contexto es normalmente el objeto `window`^[1].

Por ejemplo:

```
const obj = {
  metodo: () => {
    console.log(this);
  }
};
obj.metodo(); // Imprime 'window' en el navegador
```

En cambio, si se usara una función normal, `this` sí sería el objeto `obj`.

Esto es una diferencia fundamental a tener en cuenta cuando se trabaja con objetos y métodos en JavaScript^[1].

**

Ponme este ejemolo junto a uno de objeto sin flecha

Claro, aquí tienes dos ejemplos en JavaScript, uno con función flecha (arrow function) y otro con función normal, para que veas la diferencia en el comportamiento de `this` en un objeto:

```
const objFlecha = {
  nombre: "Objeto Flecha",
  metodo: () => {
    console.log(this); // En función flecha, 'this' refiere a 'window'
  }
};

const objNormal = {
  nombre: "Objeto Normal",
  metodo: function() {
```

```
        console.log(this); // En función normal, 'this' refiere al objeto que llama al método  
    }  
};  
  
objFlecha.metodo(); // Imprime 'window' (o contexto global)  
objNormal.metodo(); // Imprime el objeto 'objNormal'
```

En el primer caso (objFlecha), la función flecha no tiene su propio this, así que toma el this del contexto exterior, que en el navegador es el objeto window.

En el segundo caso (objNormal), el método normal tiene su propio this y apunta al objeto objNormal desde donde se llama.

Esto es clave para entender cómo utilizar this dependiendo del tipo de función que uses en JavaScript.

Cuándo se usa la función flecha y cuándo no.

Las funciones flecha se usan principalmente cuando se desea una sintaxis más corta y sencilla para funciones, especialmente para funciones pequeñas, funciones de retorno simple o callbacks. Son muy útiles cuando se quiere que el valor de this sea el mismo que el contexto donde fue declarada la función, como por ejemplo dentro de métodos que usan temporizadores (setTimeout) o eventos del DOM.

En cambio, no se deben usar funciones flecha para definir métodos de objetos porque no crean su propio contexto this. En esos casos, this siempre apuntará al contexto externo (normalmente window en navegador) y no al objeto, lo cual suele ser un comportamiento indeseado.

Cuándo usar función flecha

- Funciones pequeñas, de una línea.
- Callbacks donde se quiere heredar this del contexto donde se define la función.
- Métodos internos de temporizadores (setTimeout, setInterval).
- Funciones usadas en métodos de arrays como map, filter, reduce.

Cuándo NO usar función flecha

- Métodos de objetos donde se necesita que this refiera al objeto.
- Funciones constructoras.
- Cuando se necesita un this dinámico (por ejemplo, uso de call, apply o bind para cambiar this).
- Cuando el código se vuelve menos legible o claro por el uso excesivo de funciones flecha.

En resumen, las funciones flecha son para funciones simples y para casos donde se quiere que this sea el del contexto padre, mientras que las funciones normales son necesarias cuando se requiere que this se vincule al objeto que llama al método [\[2\]](#) [\[3\]](#) [\[4\]](#).

Anplíq e caso a caso cuándo isar función flecha (temporizador, callback...)con ejemolos ampliados, explicafos línea a línea como si fuera estudiante bachillerato

Aquí tienes una explicación ampliada, caso a caso, con ejemplos explicados línea a línea sobre cuándo usar función flecha en JavaScript, pensando en un estudiante de bachillerato.

Caso 1: Función flecha en temporizador (setTimeout)

```
const obj = {
  nombre: "Mi objeto",
  mostrarNombreConDelay: function() {
    setTimeout(() => {
      console.log(this.nombre);
    }, 1000);
  }
};

obj.mostrarNombreConDelay();
```

Explicación línea a línea:

- `const obj = { ... };`: Creamos un objeto con una propiedad `nombre` y un método llamado `mostrarNombreConDelay`.
- `mostrarNombreConDelay: function() { ... };`: Definimos el método usando función normal para que `this` dentro del método sea el propio objeto `obj`.
- `setTimeout(() => { ... }, 1000);`: Aquí usamos una función flecha dentro del `setTimeout` que se ejecutará después de 1000 milisegundos (1 segundo).
- `console.log(this.nombre);`: Dentro de la función flecha, `this` se refiere al mismo contexto donde fue creada (el método del objeto), así que `this.nombre` es "Mi objeto".
- `obj.mostrarNombreConDelay();`: Llamamos al método, que tras 1 segundo nos mostrará "Mi objeto" en consola.

Caso 2: Callback en método de array (map)

```
const numeros = [1, 2, 3];
const numerosDuplicados = numeros.map(n => n * 2);
console.log(numerosDuplicados);
```

Explicación:

- `const numeros = [1, 2, 3];`: Creamos un array con números.
- `numeros.map(n => n * 2);`: Usamos una función flecha para pasar un callback al método `map`. Para cada número `n`, se multiplica por 2.
- `n => n * 2` es una función flecha corta: recibe `n` y devuelve `n * 2` sin necesidad de usar `return`.
- `console.log(numerosDuplicados);`: Imprime `[2, 4, 6]`, el array con los números duplicados.

Caso 3: NO usar función flecha en métodos de objetos

```
const obj = {
  nombre: "Objeto",
  metodo: () => {
    console.log(this.nombre);
  }
};
obj.metodo();
```

Explicación:

- `metodo: () => {}`: La función flecha no tiene su propio `this`.
- `console.log(this.nombre);`: `this` aquí apunta al contexto global, no a `obj`.
- Al hacer `obj.metodo()`, imprimirá `undefined` (o el nombre en `window` si existiera), porque `this` no es `obj`.

Resumen

Caso	Usar función flecha	Por qué	Ejemplo corto
Temporizador (<code>setTimeout</code>)	Sí	Para que <code>this</code> sea el del objeto	<code>setTimeout(() => console.log(this.nombre), 1000)</code>
Callbacks de array	Sí	Sintaxis corta y clara	<code>[1,2].map(n => n*2)</code>
Métodos de objeto	No	Necesita su propio <code>this</code> dinámico	<code>metodo: function() { console.log(this.nombre) }</code>
Funciones constructoras	No	Función flecha no puede ser constructora	N/A

Usar función flecha te facilita trabajar cuando quieras mantener el contexto de `this` sin perderlo dentro de funciones internas (como callbacks o temporizadores). Pero en métodos de objetos o constructores, es mejor la función normal porque necesitas que `this` cambie dependiendo de cómo se llame la función.

Si quieras, puedo hacer más ejemplos detallados como estos.

Dentro de la función flecha, `this` se refiere al mismo contexto donde fue creada (el método del objeto), así que `this.nombre` es "Mi objeto" explícame mejor esto, como si fuera estudiante de bachillerato

Aquí va una explicación sencilla para un estudiante de bachillerato sobre la frase:

Dentro de la función flecha, `this` se refiere al mismo contexto donde fue creada (el método del objeto), así que `this.nombre` es "Mi objeto".

Contexto

En JavaScript, `this` es una palabra que cambia según dónde y cómo se use. Suele apuntar al objeto que "está usando" la función en ese momento.

Las funciones normales

Cuando usas una función normal dentro de un método (una función que está dentro de un objeto), `this` dentro de esa función puede cambiar y no apuntar al objeto, sino al contexto global (en navegador es `window`). Por eso a veces no funciona como esperas.

Las funciones flecha

Las funciones flecha **no crean su propio `this`**. En lugar de eso, toman el valor de `this` del lugar donde fueron creadas. Esto se llama "**this léxico**".

Ejemplo

```
const obj = {
  nombre: "Mi objeto",
  metodo: function() {
    // Aquí 'this' es 'obj', porque 'metodo' es un método de ese objeto
    const funcionFlecha = () => {
      // 'this' de la función flecha es el mismo que el de donde fue creada, o sea, 'obj'
      console.log(this.nombre);
    };
    funcionFlecha();
  }
};

obj.metodo(); // Imprime: "Mi objeto"
```

Por qué:

- `metodo` es una función normal dentro de `obj`, entonces `this` dentro de `metodo` es el objeto `obj`.
- `funcionFlecha` es una función flecha, y fue creada dentro de `metodo`.
- La función flecha no tiene su propio `this`, así que usa el mismo `this` que había fuera de ella, que es `obj`.
- Por eso, cuando hace `console.log(this.nombre)`, "this" es realmente el objeto `obj`, y muestra "Mi objeto".

Resumen simple

- Función flecha → usa el `this` de donde fue creada.
- Función normal → puede tener un `this` diferente dentro de otra función.

Esto ayuda a que dentro de funciones internas (como temporizadores o callbacks) `this` siga apuntando al objeto esperado, y así no hay confusión.

¿Quieres que explique más con otro ejemplo?

**

`const objFlecha = {`

```
nombre: "Objeto Flecha",
metodo: () => {
  console.log(this); // En función flecha, 'this' refiere a 'window'
}
}; const obj = {
  nombre: "Mi objeto",
  metodo: function() {
    // Aquí 'this' es 'obj', porque 'metodo' es un método de ese objeto
    const funcionFlecha = () => {
      // 'this' de la función flecha es el mismo que el de donde fue creada, o sea, 'obj'
      console.log(this.nombre);
    };
    funcionFlecha();
  }
};

obj.metodo(); // Imprime: "Mi objeto". Por qué el yhis se comporta diferente si son 2 funciones flecha.
```

El comportamiento diferente de `this` en los dos ejemplos con funciones flecha se debe al contexto en el que fueron definidas, no a que sean ambas funciones flecha por se. Aquí va una explicación clara:

Ejemplo 1: Función flecha como método del objeto (objFlecha)

```
const objFlecha = {
  nombre: "Objeto Flecha",
  metodo: () => {
    console.log(this); // 'this' se refiere a 'window' (contexto global)
  }
};

objFlecha.metodo();
```

- Aunque `metodo` es una función flecha, **fue definida en el contexto global**, no dentro de una función tradicional.
- Las funciones flecha no tienen su propio `this`, así que usan el `this` del contexto donde fueron creadas.
- Aquí, ese contexto es el global (`window` en navegador), no el objeto `objFlecha`.
- Por eso, `this` dentro de `metodo` apunta a `window` y no a `objFlecha`.

Ejemplo 2: Función flecha dentro de método normal (obj)

```
const obj = {
  nombre: "Mi objeto",
  metodo: function() {
    const funcionFlecha = () => {
      console.log(this.nombre); // 'this' se refiere a 'obj'
    };
    funcionFlecha();
  }
};

obj.metodo();
```

- `metodo` es una función normal, cuando se llama usando `obj.metodo()`, `this` dentro de `metodo` es el objeto `obj`.
- La función flecha `funcionFlecha` está **dentro de la función normal** `metodo`.
- La función flecha usa el `this` del contexto donde fue creada, que en este caso es el `this` dentro de `metodo` (el objeto `obj`).
- Por eso, `this.nombre` es "Mi objeto".

Resumen

Situación	¿Qué pasa con <code>this</code> en la función flecha?
Función flecha definida directamente en objeto (como método)	Usa <code>this</code> del contexto global (no el objeto).
Función flecha definida dentro de función normal del objeto	Usa <code>this</code> del método, que es el objeto.

La clave es que las funciones flecha **usan el this del contexto donde fueron definidas**, no donde son llamadas. Si se definen en el contexto global (o fuera de un método tradicional), this será el global. Si se definen dentro de un método tradicional, this será el del objeto que llama.

¿Quieres que haga un ejemplo visual con comentarios paso a paso?

*

Sí

Claro, aquí va un ejemplo visual y explicado paso a paso para entender por qué this se comporta diferente en funciones flecha según dónde se definan:

```
const objFlecha = {
  nombre: "Objeto Flecha",

  // Función flecha definida DIRECTAMENTE como método del objeto
  metodo: () => {
    // Aquí 'this' NO es el objeto, sino el contexto global (window)
    console.log(this);
  }
};

objFlecha.metodo();
// Salida: window (porque el 'this' de la flecha es el contexto donde fue creada, que es

const objNormal = {
  nombre: "Objeto Normal",

  // Método normal, que sí tiene su propio 'this'
  metodo: function() {
    // Ahora, 'this' es el objeto objNormal
    console.log("Este es 'this' dentro del método normal:", this);

    // Función flecha definida DENTRO de método normal
    const funcionFlecha = () => {
      // La función flecha toma el mismo 'this' del método normal (es decir, del objeto)
      console.log("Este es 'this' dentro de la función flecha:", this);
      console.log(this.nombre); // Imprime "Objeto Normal"
    };

    funcionFlecha();
  }
};

objNormal.metodo();
// Salida:
// "Este es 'this' dentro del método normal:" {nombre: "Objeto Normal", metodo: f}
// "Este es 'this' dentro de la función flecha:" {nombre: "Objeto Normal", metodo: f}
// "Objeto Normal"
```

Explicación paso a paso

1. `objFlecha.metodo` es una función flecha definida directamente como método del objeto, pero ella **hereda el `this` de su contexto de creación**, que es el global (fuera del objeto). Por eso imprime `window`.
2. En `objNormal.metodo`, la función es una función normal, por eso cuando la llamamos con `objNormal.metodo()`, dentro de ella `this` es `objNormal`.
3. Dentro de este método normal (`objNormal.metodo`) se define una función flecha llamada `funcionFlecha`. Esta función flecha **no tiene su propio `this`**, así que toma el mismo `this` donde fue creada, que es `objNormal`.
4. Por eso, dentro de `funcionFlecha`, `this.nombre` es "Objeto Normal", porque la función flecha hereda el contexto `this` del método normal.

Este patrón es útil cuando quieras que una función interna use el mismo `this` del método donde está definida para evitar problemas comunes con el valor dinámico de `this`.

Si algo no quedó claro, puedo explicarlo con otro ejemplo o con analogías.

Usamos una función flecha para pasar un callback al método `map`. Para cada número `n`, se multiplica por 2. Qué significa pasar un callback?

Pasar un callback significa pasar una función como argumento (o parámetro) a otra función para que esa función externa la llame (o "ejecute") durante su proceso.

En otras palabras, un callback es una función que se le da a otra función para que la "llame de vuelta" (de ahí el nombre "callback") cuando sea necesario. Esto permite que las funciones sean más flexibles y originales, pues pueden ejecutar código específico en momentos precisos.

Ejemplo sencillo:

```
function decirHola() {  
  console.log("Hola!");  
}  
  
function ejecutarFuncion(func) {  
  func(); // Aquí se llama a la función que recibimos como argumento  
}  
  
ejecutarFuncion(decirHola);  
// Sale en consola: "Hola!"
```

En el contexto del método `map`:

```
const numeros = [1, 2, 3];  
const duplicados = numeros.map(n => n * 2);
```

- `map` es una función que recibe otra función (`callback`) y la ejecuta para cada elemento del array.
- Esa función `callback` recibe un número `n` y devuelve `n * 2`.
- El método `map` crea un nuevo array con los valores devueltos por la función `callback`.

Así, pasar un `callback` es darle a otra función un trozo de código para que lo use y lo ejecute más tarde, muchas veces varias veces, por ejemplo, por cada elemento de un array.

¿Quieres que prepare más ejemplos para entenderlo?

**

Entonces no tiene por qué recibir otra función puede ser una variable?

No, una función `callback` **debe ser necesariamente una función** que se pasa como argumento a otra función para que esta última la llame y ejecute cuando sea necesario.

Una variable simple (como un número, texto u objeto) no es un `callback` porque no es una función que se pueda ejecutar. Cuando se habla de "pasar un `callback`", siempre se refiere a pasar una función (normal, anónima o flecha).

Por ejemplo:

```
function saludar() {
  console.log("Hola");
}

function ejecutar(fn) {
  fn(); // Ejecuta la función que recibió como argumento
}

ejecutar(saludar); // Aquí pasamos una función como callback
```

Pasar solo una variable así no funciona como `callback`:

```
const mensaje = "Hola";

function ejecutar(fn) {
  fn(); // Intentará ejecutar `fn` que debería ser función
}

ejecutar(mensaje); // Error porque `mensaje` no es función
```

En resumen, para que algo funcione como `callback`, debe ser una función o expresión de función, capaz de ser llamada o ejecutada.

¿Quieres que prepare más ejemplos o una explicación con código?

**

const numeros = [1, 2, 3];

const duplicados = numeros.map(n => n * 2);

Y aquí? Dónde está la función?

En el código que mostraste:

```
const numeros = [1, 2, 3];
const duplicados = numeros.map(n => n * 2);
```

La función está en la parte `n => n * 2`. Esa es la función callback que se pasa al método `map`.

¿Dónde está la función?

- `n => n * 2` es una función flecha que toma un argumento `n` (cada número del array).
- Esa función devuelve `n * 2`, es decir el número multiplicado por 2.
- El método `map` ejecuta esa función para cada elemento del array `numeros`.

Qué hace el método `map`

- Toma la función callback (`n => n * 2`) y la aplica a cada elemento del array.
- El resultado de esa función para cada elemento se pone en un nuevo array.
- Por eso, el resultado `duplicados` es `[2, 4, 6]`, el array original pero con cada número multiplicado por 2.

Es como si `map` dijera:

"Por cada número del array, pasa este número a la función que me diste y guarda el resultado en un nuevo array".

Si quieras, puedo mostrar un ejemplo menos abreviado para que veas mejor la función explícita.

**

1. [image-1.jpg](#)

2. https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions/Arrow_functions

3. <https://www.freecodecamp.org/espanol/news/cuando-y-por-que-debes-usar-las-funciones-flecha-de-es6-y-cuando-no/>

4. <https://www.aulascript.com/lenguaje/funciones-flecha.htm>

5. <https://es.stackoverflow.com/questions/477240/función-flecha-vs-función-regular>

6. <https://es.javascript.info/arrow-functions-basics>

7. <https://www.youtube.com/watch?v=TmWBSMu4pm4>

8. <https://desarrolloweb.com/faq/funciones-flecha-que-las-diferencia-de-las-funciones-comunes>

9. <https://lenguajejs.com/javascript/funciones/arrow-functions/>
10. <https://www.youtube.com/watch?v=HVEkbCZAuqA>
11. <https://didacticode.com/curso/curso-javascript-es6/leccion/javascript-es6-06-arrow-functions-funcion-es-flecha-2/>
12. https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions/Arrow_functions
13. <https://es.javascript.info/arrow-functions-basics>
14. <https://lenguajejs.com/javascript/funciones/arrow-functions/>
15. <https://www.aulascript.com/lenguaje/funciones-flecha.htm>
16. <https://desarrolloweb.com/faq/funciones-flecha-que-las-diferencia-de-las-funciones-comunes>
17. <https://www.freecodecamp.org/espanol/news/cuando-y-por-que-debes-usar-las-funciones-flecha-de-es6-y-cuando-no/>
18. <https://www.youtube.com/watch?v=c3LMuRmMonk>
19. <https://uriandviera.com/javascript/arrow-functions-en-javascript>
20. <https://es.stackoverflow.com/questions/363155/explicación-de-funciones-flecha-en-javascript>
21. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this>
22. https://www.w3schools.com/js/js_arrow_function.asp
23. <https://www.geeksforgeeks.org/javascript/behavior-of-arrow-functions-and-regular-functions-for-this-keyword/>
24. <https://syntackle.com/blog/this-keyword-in-arrow-functions-javascript/>
25. <https://www.scaler.com/topics/nodejs/this-keyword-in-arrow-functions/>
26. <https://stackoverflow.com/questions/72292221/why-this-keyword-in-arrow-function-represents-the-object-rather-than-global-ob>
27. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions
28. https://www.reddit.com/r/learnjavascript/comments/o6m7i2/arrow_function_and_the_this_keyword/
29. <https://web.dev/learn/javascript/functions/this>
30. https://developer.mozilla.org/es/docs/Glossary/Callback_function
31. <https://lenguajejs.com/asincronia/callbacks/que-son/>
32. <https://www.freecodecamp.org/espanol/news/funciones-callback-en-javascript-que-son-los-callback-en-js-y-como-usarlos/>
33. <https://www.youtube.com/watch?v=kjCH7vvISsE>
34. <https://keepcoding.io/blog/que-son-los-callbacks-en-javascript/>
35. <https://antonioweb.dev/articulo/funcion-callback-javascript-guia-completa>
36. <https://www.freecodecamp.org/espanol/news/que-es-una-funcion-callback-javascript/>
37. <https://desarrolloweb.com/articulos/funciones-callback-javascript>
38. https://es.linkedin.com/posts/midudev_explicación-callbacks-y-promesas-en-javascript-activity-7118216336853327872-HUbz
39. https://developer.mozilla.org/es/docs/Glossary/Callback_function
40. <https://lenguajejs.com/asincronia/callbacks/que-son/>
41. <https://www.freecodecamp.org/espanol/news/que-es-una-funcion-callback-javascript/>
42. <https://es.javascript.info/callbacks>

43. https://jstips.co/es_es/javascript/passing-arguments-to-callback-functions/
44. <https://jscodez.com/javascript-funciones-de-callback/>
45. https://es.linkedin.com/posts/midudev_te-explico-callbacks-y-promesas-en-javascript-activity-7187087947723100161-M3kP
46. <https://antonioweb.dev/articulo/funcion-callback-javascript-guia-completa>
47. <https://www.youtube.com/watch?v=EofriyLQohk>
48. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map
49. <https://www.freecodecamp.org/espanol/news/javascript-map-como-utilizar-la-funcion-js-map-metodo-de-arreglo/>
50. https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/map
51. <https://www.freecodecamp.org/news/javascript-map-method/>
52. https://www.reddit.com/r/learnjavascript/comments/17sqevm/is_my_understanding_of_the_map_callback_function/
53. <https://talent500.com/blog/how-to-use-javascripts-map-filter-and-reduce-methods/>
54. <https://blog.boot.dev/javascript/javascript-map-function/>
55. <https://www.ionos.com/digitalguide/websites/web-development/javascript-map/>
56. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map
57. <https://stackoverflow.com/questions/10872370/js-callback-using-map-with-a-function-that-has-one-additional-parameter>