



No xplicame esto, dindle le dices qué hacer si es false?

En el ejemplo de la imagen, solo se muestra qué hacer cuando la promesa se resuelve exitosamente usando `resolve` y el método `.then`. Sin embargo, no se muestra el manejo para el caso de rechazo o error (cuando "es false", o sea, cuando ocurre un fallo).

Para indicar qué hacer cuando la promesa falla (es rechazada con `reject`), deberías añadir un segundo argumento en `.then` o, más comúnmente, usar `.catch` así:

```
promesa
  .then(function(data) {
    console.log(data); // Se ejecuta si la promesa se resuelve (resolve)
  })
  .catch(function(error) {
    console.log(error); // Se ejecuta si la promesa es rechazada (reject)
 });
```

De esta forma, el código sabría qué hacer tanto si la promesa se cumple como si falla.

**

Resolve y reject son palabras claves? Escribe la promesa entera junto a que hacer con éxito y fracaso

No, `resolve` y `reject` no son palabras reservadas ni palabras clave del lenguaje JavaScript, pero sí son nombres convencionales y especiales usados al crear promesas. Son funciones que se pasan como parámetros al constructor de la promesa y sirven para comunicar si la operación asíncrona fue exitosa o fallida.

Aquí tienes un ejemplo completo de una promesa con manejo de éxito y fallo:

```
let promesa = new Promise(function(resolve, reject) {
  // Simulación de operación asíncrona con setTimeout
  setTimeout(function() {
    const exito = true; // Cambia a false para simular fallo

    if (exito) {
      resolve(';Operación exitosa!'); // Aquí se indica que la promesa se cumple
    } else {
```

```

        reject('Ha ocurrido un error.'); // Aquí se indica que la promesa falla
    }
}, 1000);
});

promesa
.then(function(resultado) {
    console.log('Éxito:', resultado); // Se ejecuta si se llama a resolve
})
.catch(function(error) {
    console.log('Error:', error); // Se ejecuta si se llama a reject
});

```

En ese código:

- `resolve(valor)` notifica que la promesa se ha cumplido exitosamente entregando un valor.
- `reject(motivo)` notifica que la promesa ha fallado, pasando un motivo o error.
- Luego `.then` maneja el resultado exitoso.
- `.catch` maneja el error si ocurre un rechazo.

Así queda claro qué hacer tanto en caso de éxito como de fallo de la promesa [\[2\]](#) [\[3\]](#) [\[4\]](#).

**

Explícame linea a linea lo que hacemos como estudiante de bachillerato

Claro, aquí está la explicación línea a línea del código de la promesa, con un nivel para estudiantes de bachillerato:

```
let promesa = new Promise(function(resolve, reject) {
```

- Creamos una nueva promesa y la guardamos en la variable `promesa`. La promesa es una operación que no se completa ahora mismo, sino que puede terminar bien (`resolve`) o mal (`reject`) más adelante.
- La promesa recibe una función con dos parámetros: `resolve` y `reject`. Estas son funciones que usamos para decir si la promesa terminó bien o mal.

```
setTimeout(function() {
```

- Usamos `setTimeout` para simular que la operación tarda un tiempo, por ejemplo, 1 segundo (1000 milisegundos).
- Esto representa algo que tarda en completarse, como pedir datos a un servidor.

```
    resolve('timeout');
```

- Aquí llamamos a `resolve` con el mensaje 'timeout'. Esto significa que la promesa terminó correctamente y devuelve ese mensaje.
- Si hubiera un error o no queremos que termine bien, usaríamos `reject`.

```
}, 1000);
```

- Cerramos el `setTimeout`, que dice que después de 1000 milisegundos (1 segundo) ejecutará la función anterior.

```
});
```

- Cerramos la función del constructor de la promesa.

```
promesa.then(function(data) {
```

- Usamos el método `.then()` para decir qué hacer cuando la promesa se haya resuelto con éxito.
- La función que pasamos recibe el resultado (que en este caso es 'timeout') como `data`.

```
    console.log(data);
```

- En este punto mostramos el resultado en la consola, es decir, imprimimos `timeout`.

```
});
```

- Cerramos la función y el método `.then`.

En resumen: creamos una promesa que completa su trabajo después de 1 segundo y nos devuelve el mensaje 'timeout'. Cuando la promesa termina bien, mostramos ese mensaje.

Si quisieras hacer algo en caso de que la promesa falle (`reject`), tendrías que usar `.catch` con una función que maneje ese error.

Espero que esta explicación clara y paso a paso ayude a entender cómo funcionan las promesas en JavaScript con un ejemplo básico [image-1] [12] [13].



Cuando usamos promesa.then y llamamos a una función a la que le pasamos resultado como parámetro. Como sabe que es resultado? No es palabras clave.

Cuando usas `promesa.then(function(resultado) { ... })`, el nombre `resultado` no es una palabra clave, es simplemente el nombre que se le da al parámetro de la función que defines.

Así es como funciona:

- Cuando la promesa se resuelve, internamente se llama a la función que pasas a `.then`.
- La promesa pasa al primer parámetro de esa función el valor con el que se resolvió, es decir, el valor que se puso en `resolve(valor)`.
- El nombre del parámetro en la función es solo una variable que tú decides, por ejemplo, puede ser `resultado`, `data`, `valor`, o cualquier otro nombre.
- Esa variable recibe ese valor que pasó la promesa, para que puedas usarlo dentro de la función.

Por ejemplo:

```
promesa.then(function(miValor) {  
  console.log(miValor);  
});
```

Aquí la promesa pasa el valor a `miValor`, pero también podría llamarse `resultado` o `x` sin problema. Es solo el nombre del parámetro de la función.

En resumen, la promesa sabe qué valor pasar a esa función porque tú llamas a `resolve(valor)` cuando se resuelve, y el `.then` recibe ese valor como argumento en la función que le pasas. El nombre de la variable en la función es libre y no tiene impacto en el funcionamiento^[14] ^[15].

*

Explícame qué hace la promesa. Entiendo que las dificultades que tengo para entender es que leo de arriba a abajo y creo que ese no es el orden de ejecución de la promesa.

La confusión común con las promesas es que el código se lee de arriba hacia abajo, pero las promesas funcionan de manera asíncrona y su orden de ejecución no sigue ese flujo lineal. Esto es lo que pasa con la promesa:

1. Cuando creas una promesa con `new Promise(...)`, la función que pasas (con `resolve` y `reject`) se ejecuta inmediatamente y de forma síncrona hasta que llama a `resolve` o `reject`.

2. Sin embargo, cuando llamas a `resolve()` o `reject()`, eso no ejecuta inmediatamente las funciones que pasaste a `.then()` o `.catch()`. En vez de eso, la promesa queda "resuelta" o "rechazada", pero las funciones dentro de `.then()` o `.catch()` se ejecutan de forma asíncrona, justo después de que el resto del código síncrono termine y en un ciclo separado del "event loop" (bucle de eventos).
3. Esto quiere decir que si tienes código después de crear la promesa y de asignar `.then()`, ese código se ejecutará primero, antes que la función dentro del `.then()`.
4. Cuando la promesa se resuelve, JavaScript agenda la función dentro de `.then()` para que se ejecute luego, no inmediatamente, aunque la promesa ya haya sido resuelta.

Ejemplo del orden de ejecución:

```
let promesa = new Promise(function(resolve, reject) {
  console.log('1. Promesa empieza');
  resolve('Resultado de la promesa');
  console.log('2. Promesa termina');
});

promesa.then(function(resultado) {
  console.log('4. Función en then:', resultado);
});

console.log('3. Código después de then');
```

El resultado en consola es:

1. Promesa empieza
2. Promesa termina
3. Código después de then
4. Función en then: Resultado de la promesa

Esto muestra que `.then()` se ejecuta después de que todo el código síncrono termine, incluso si la promesa ya se resolvió.

En resumen, las promesas se comportan de forma asíncrona al momento de ejecutar las funciones `then` o `catch`, por eso el orden de ejecución no es lineal, sino que el código dentro de `.then()` se ejecuta en un ciclo posterior del evento mientras que el resto del código continúa corriendo primero [23] [24] [25].

**

1. image-1.jpg
2. <https://www.freecodecamp.org/espanol/news/como-escribir-una-promesa-de-javascript/>
3. <https://carontestudio.com/blog/que-es-una-promesa-en-javascript-todo-lo-que-necesitas-saber/>
4. <https://es.stackoverflow.com/questions/64265/qu%C3%A9-es-una-promesa-en-javascript>
5. https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using_promises

6. <https://www.mgpanel.org/post/promesas-y-async-await-en-javascript-simplificando-el-codigo-asincrono>
7. <https://keepcoding.io/blog/sintaxis-de-promesas-en-javascript/>
8. <https://www.espai.es/blog/2019/05/async-await-otra-forma-de-usar-promesas-en-javascript/>
9. https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Promise
10. <https://www.manualweb.net/javascript/promesas-en-javascript/>
11. <https://somospnt.com/blog/277-promesas-en-javascript>
12. <https://www.freecodecamp.org/espanol/news/como-escribir-una-promesa-de-javascript/>
13. <https://carontestudio.com/blog/que-es-una-promesa-en-javascript-todo-lo-que-necesitas-saber/>
14. https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Promise/then
15. <https://www.arsys.es/blog/promesas-javascript>
16. https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using_promises
17. https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Promise
18. <https://www.aulascript.com/avanzado/promesas.htm>
19. <https://es.javascript.info/promise-chaining>
20. <https://platzi.com/blog/que-es-y-como-funcionan-las-promesas-en-javascript/>
21. <https://lenguajejs.com/asincronia/promesas/new-promise/>
22. <https://es.stackoverflow.com/questions/523444/enviar-parámetros-a-una-promesa>
23. https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using_promises
24. https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Promise
25. https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Promise/then
26. <https://es.stackoverflow.com/questions/282632/orden-de-ejecución-de-funciones-js-con-promesas>
27. <https://lenguajejs.com/asincronia/control/que-es/>
28. <https://sgmonda.com/controlar-la-concurrencia-de-promesas-en-javascript>
29. <https://dev.to/duxtech/es6-promesas-en-javascript-5bpb>
30. <https://blogprog.gonzalopez.es/articulos/promesas-en-secuencia-o-paralelo-en-javascript.html>
31. <https://www.youtube.com/watch?v=DdDEzpjkW8>
32. https://www.reddit.com/r/learnjavascript/comments/vh7shu/can_anyone_explain.promise_order_of_execution_here/