Asincronía

La **asincronía** es uno de los conceptos principales que rige el mundo de JavaScript. Cuando comenzamos a programar, normalmente realizamos tareas de forma **síncrona**, llevando a cabo tareas secuenciales que se ejecutan una detrás de otra, de modo que elorden o flujo del programa es sencillo y fácil de observar en el código:

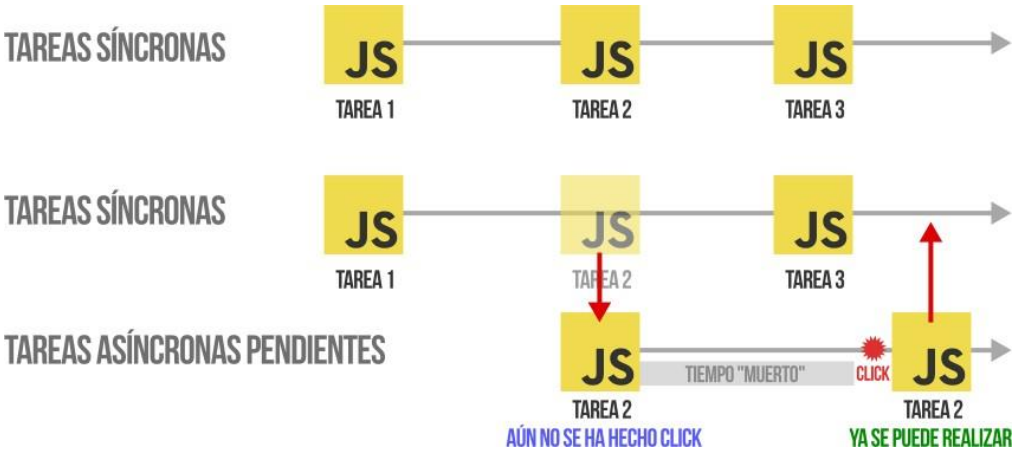
```
primera_funcion();          // Tarea 1: Se ejecuta primero
segunda_funcion(); // Tarea 2: Se ejecuta cuando termina primera_funcion()
tercera_funcion();          // Tarea 3: Se ejecuta cuando termina segunda_funcion()
```

Sin embargo, en el mundo de la programación, tarde o temprano necesitaremos realizar operaciones **asíncronas**, especialmente en ciertos lenguajes como JavaScript, donde tenemos que realizar tareas **que tienen que esperar a que ocurra un determinado suceso** que no depende de nosotros, y reaccionar realizando otra tarea sólo cuando dichosuceso ocurra.

Lenguaje no bloqueante

Cuando hablamos de JavaScript, habitualmente nos referimos a él como un lenguaje **no bloqueante**. Con esto queremos decir que las tareas que realizamos no se quedan bloqueadas esperando ser finalizadas, y por consiguiente, evitando proseguir con el restode tareas.

Imaginemos que la **segunda_funcion()** del ejemplo anterior realiza una tarea que depende de otro factor, como por ejemplo un click de ratón del usuario. Si hablasemos de un **lenguaje bloqueante**, hasta que el usuario no haga click, JavaScript no seguiría ejecutandolas demás funciones, sino que se quedaría bloqueado esperando a que se terminase esa segunda tarea:



Pero como JavaScript es un **lenguaje no bloqueante**, lo que hará es mover esa tarea auna lista de **tareas pendientes** a las que irá «*prestando atención*» a medida que lo necesite, pudiendo continuar y retomar el resto de tareas a continuación de la segunda.

¿Qué es la asincronía?

Pero esto no es todo. Ten en cuenta que pueden existir **múltiples** tareas asíncronas, dichastareas puede que terminen realizándose correctamente (*o puede que no*) y ciertas tareas pueden depender de otras, por lo que deben respetar un cierto orden. Además, es muy habitual que no sepamos previamente **cuánto tiempo** va a tardar en terminar una tarea, porlo que necesitamos un mecanismo para controlar todos estos factores.

¿Cómo gestionar la asincronía?

Teniendo en cuenta el punto anterior, debemos aprender a buscar mecanismos para dejarclaro en nuestro código JavaScript, que ciertas tareas tienen que procesarse de forma asíncrona para quedarse a la espera, y otras deben ejecutarse de forma síncrona.

En JavaScript existen varias formas de gestionar la **asincronía**, donde quizás las más populares son las siguientes:

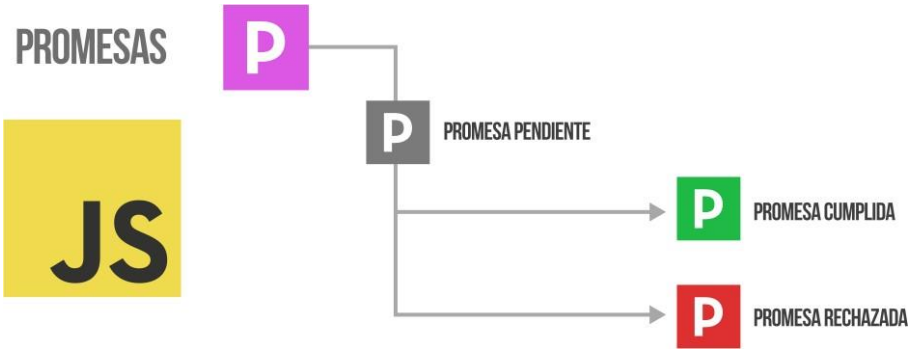
Método	Descripción
Mediante **callbacks**	Probablemente, la forma más clásica de gestionar la asincroníaen JavaScript.
Mediante **promesas**	Una forma más moderna y actual de gestionar la asincronía.
Mediante **async**/**await**	Seguimos con promesas, pero con **async**/**await** añadimos más azúcar sintáctico.

Fuente: lenguajejs.com / programacionymas.com

Promesas

Las **promesas** son un concepto para resolver el problema de asincronía de una forma mucho más elegante y práctica.

Como su propio nombre indica, una **promesa** es algo que, en principio pensamos que secumplirá, pero en el futuro pueden ocurrir varias cosas:



- La promesa **se cumple** (*promesa resuelta*)
- La promesa **no se cumple** (*promesa se rechaza*)
- La promesa se queda en un **estado incierto** indefinidamente (*promesa pendiente*)

Básicamente, las Promesas son similares a los Eventos, con las siguientes diferencias:

- Una promesa sólo puede tener éxito o fracasar una única vez. No puede tener éxito o fallar por una 2da vez, ni cambiar de éxito a fallo posteriormente, o viceversa.
- Si una promesa ha sido exitosa o ha fallado, y más adelante (recién)registramos un callback de success o failure, la función de callback correspondiente será llamada (incluso si el evento tuvo lugar antes).

Esto resulta muy útil para operaciones asíncronas, porque más allá de capturar el momentoexacto en que ocurre algo, nos enfocamos en reaccionar ante lo ocurrido.

Terminología asociada a las Promesas

Tenemos muchos términos relacionados a lo que son Promesas en Javascript. A continuaciónveamos lo más básico.

Una promesa puede presentar los siguientes estados:

- **fulfilled** - La acción relacionada a la promesa se llevó a cabo con éxito
- **rejected** - La acción relacionada a la promesa falló
- **pending** - Aún no se ha determinado si la promesa fue fulfilled o rejected
- **settled** - Ya se ha determinado si la promesa fue fulfilled o rejected

También se suele usar el término **thenable**, para indicar que un objeto tiene disponible unmétodo "then" (y que por tanto está relacionado con Promesas).

Promesas en JavaScript

Las **promesas** en JavaScript se representan a través de un **object**, y cada **promesa** estaráen un estado concreto: **pendiente**, **aceptada** o **rechazada**. Además, cada **promesa** tiene los siguientes métodos, que podremos utilizar para utilizarla:

Métodos	Descripción
.then(resolve)	Ejecuta la función callback **resolve** cuando la promesa secumple.
.catch(reject)	Ejecuta la función callback **reject** cuando la promesa serechaza.
.then(resolve,reject)	Método equivalente a las dos anteriores en el mismo **.then()**.
.finally(end)	Ejecuta la función callback **end** tanto si se cumple como si se rechaza.

Cómo crear una Promesa en JS

Un objeto Promise representa un valor, que no se conoce necesariamente al momento de crear la promesa.

Esta representación nos permite realizar acciones, con base en el valor de éxito devuelto, o la razón de fallo.

Es decir, los métodos asíncronos producen valores que aún no están disponibles. Pero la idea ahora es que, en vez de esperar y devolver el valor final, tales métodos devuelven un objeto Promise (que nos proveerá del valor resultante en el futuro).

Hoy en día, muchas bibliotecas JS se están actualizando para hacer uso de Promesas, en vez de simples funciones callback.

Nosotros también podemos crear nuestras promesas, basados en esta sintaxis:

```
new Promise(function(resolve, reject) { ... });
```

Este constructor es usado principalmente para envolver funciones que no soportan el uso dePromesas.

- El constructor espera una función como parámetro. A esta función se leconoce como **executor**.
- Esta función executor recibirá 2 argumentos: **resolve** y **reject**.
- La función executor es ejecutada inmediatamente al implementar el objeto Promise, recibiendo las funciones resolve y reject para su uso correspondiente. Esta función executor es llamada incluso antes que el constructor Promise devuelva el objeto creado.
- Las funciones resolve y reject, al ser llamadas, "resuelven" o "rechazan" la promesa. Es decir, modifican el estado de la promesa (como hemos visto antes, inicialmente es pending, pero posteriormente puede ser fulfilled o rejected).
- Normalmente el executor inicia alguna operación asíncrona, y una vez que ésta se completa, llama a la función resolve para resolver la promesa o bienreject si ocurrió algo inesperado.
- Si la función executor lanza algún error, la promesa también es rejected.
- El valor devuelto por la función executor es ignorado.

```
let promise = new Promise(function(resolve, reject) {function sayHello() {
  resolve('Hello World!')
  }
  setTimeout(sayHello, 3000)
  });
  console.log(promise);
```

Lo que ha de ocurrir es lo siguiente:

- Se ejecuta la función executor y se crea nuestro objeto Promise.
- Se llama al método then, expresando qué es lo que queremos hacer con elvalor que devolverá la promesa.
- Se imprime por consola [object Promise].
- A los 3 segundos tras ejecutar la función executor, se resuelve la promesa, y terminamos mostrando por consola el mensaje "Hello World!".

Ejemplo con .then(), .catch() y .finally()

```
let promesa = new Promise(function(resolve,reject){
if(true){
resolve(`Funcionó!`);
```

```
}
else{
reject(`Hay un error`);
}
});

promesa.then(function(respuesta){
console.log(`Respuesta: ${respuesta}`);
})
.catch(function(error){
console.log(`Error: ${error}`);
})
.finally(function(){
console.log(`Esto se ejecuta siempre`);
});
```

Objeto Promise

El objeto **Promise** de Javascript tiene varios métodos que podemos utilizar en nuestro código. Todos devuelven una promesa y son los que veremos en la siguiente tabla:

Métodos	Descripción
Promise.all([array]list)	Acepta sólo si **todas** las promesas del **array** secumplen.
Promise.allSettled([array]list)	Acepta sólo si **todas** las promesas del **array** secumplen o rechazan.
Promise.any([array]list)	Acepta con el valor de la primera promesa del **array** que se cumpla.
Promise.race([array]list)	Acepta o rechaza dependiendo de la primera promesaque se procese.
Promise.resolve(value)	Devuelve un valor envuelto en una promesa que secumple directamente.
Promise.reject(value)	Devuelve un valor envuelto en una promesa que se rechaza directamente.

En los siguientes ejemplos, vamos a utilizar la función **fetch()** para realizar varias peticionesy descargar varios archivos diferentes que necesitaremos para nuestras tareas.

Promise.all()

El método **Promises.all()** funciona como un «**todo o nada**»: devuelve una promesa que secumple cuando todas las promesas del **array** se cumplen. Si alguna de ellas se rechaza, **Promise.all()** también lo hace.
A **Promise.all()** le pasamos un **array** con las promesas individuales. Cuando **todas y cadauna** de esas promesas se cumplan favorablemente, **entonces** se ejecutará la función callback de su **.then()**. En el caso de que alguna se rechace, no se llegará a ejecutar.

Promise.allSettled()

El método **Promises.allSettled()** funciona como un «**todas procesadas**»: devuelve unapromesa que se cumple cuando todas las promesas del **array** se hayan procesado, independientemente de que se hayan cumplido o rechazado. });
Esta promesa nos devuelve un campo **status** donde nos indica si cada promesa individual ha sido cumplida o rechazada, y un campo **value** con los valores devueltos por la promesa.

Promise.any()

El método **Promise.any()** funciona como «**la primera que se cumpla**»: Devuelve una promesa con el valor de la primera promesa individual del **array** que se cumpla. Si todas laspromesas se rechazan, entonces devuelve una promesa rechazada.
Promise.any() devolverá una respuesta de la primera promesa cumplida.

Promise.race()

El método **Promise.race()** funciona como una «**la primera que se procese**»: la primerapromesa del **array** que sea procesada, independientemente de que se haya cumplido o rechazado, determinará la devolución de la promesa del **Promise.race()**. Si se cumple, devuelve una promesa cumplida, en caso negativo, devuelve una rechazada.
De forma muy similar a la anterior, **Promise.race()** devolverá la promesa que se resuelvaprimero, ya sea cumpliéndose o rechazándose.

Async/Await

La palabra clave async

En **ES2017** se introducen las palabras clave **async**/**await**, que no son más que una forma de **azúcar sintáctico** para gestionar las promesas de una forma más sencilla. Con **async**/**await** seguimos utilizando promesas, pero abandonamos el modelo de encadenamiento de **.then()** para utilizar uno en el que trabajamos de forma más tradicional.

En primer lugar, tenemos la palabra clave **async**. Esta palabra clave se colocará previamente a **function**, para definirla así como una **función asíncrona**, el resto de la función no cambia:

```
async function funcion_asincrona() {
  return 42;
}
```

En el caso de que utilicemos **arrow function**, se definiría como vemos a continuación, colocando el **async** justo antes de los parámetros de la arrow function:

```
const funcion_asincrona = async () => 42;
```

Al ejecutar la función veremos que ya nos devuelve una que ha sido cumplida, con el valordevuelto en la función (*en este caso, 42*). De hecho, podríamos utilizar un **.then()** para manejar la promesa:

```
funcion_asincrona().then(function(value){
  console.log("El resultado devuelto es: ", value);
});
```

Sin embargo, veremos que lo que se suele hacer junto a **async** es utilizar la palabra clave
await, que es donde reside lo interesante de utilizar este enfoque.

La palabra clave await

Cualquier función definida con **async**, o lo que es lo mismo, cualquier **Promise** puede utilizarse junto a la palabra clave **await** para manejarla. Lo que hace **await** es esperar a quese resuelva la promesa, mientras permite continuar ejecutando otras tareas que puedan realizarse:

```
const funcion_asincrona = async function(){return 42;}
```

```
const value = funcion_asincrona();                    // Promise { <fulfilled>: 42 }const
asyncValue = await funcion_asincrona(); // 42
```

Observa que en el caso de **value**, que se ejecuta sin **await**, lo que obtenemos es el

valor devuelto por la función, pero «envuelto» en una promesa que deberá utilizarse con **.then()** para manejarse. Sin embargo, en **asyncValue** estamos obteniendo un tipo de dato **Number**
, guardando el valor directamente ya procesado, ya que **await** espera a que se resuelva lapromesa de forma asíncrona y guarda el valor.

> Esto hace que la forma de trabajar con **async**/**await**, aunque se sigue trabajando exactamente igual con promesas, sea mucho más fácil y trivial para usuarios que no estén acostumbrados a las promesas y a la asincronía en general, ya que el código «parece» síncrono.

> Recuerda que en el caso de querer controlar errores o promesas rechazadas, siempre podrás utilizar bloques **try**/**catch**.