Temas a tratar

- Asincronia en Javascrip
- Set time out
- Promesas

¿Qué es la asincrónia?

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 el orden o flujo del programa es sencillo y fácil de observar en el código:

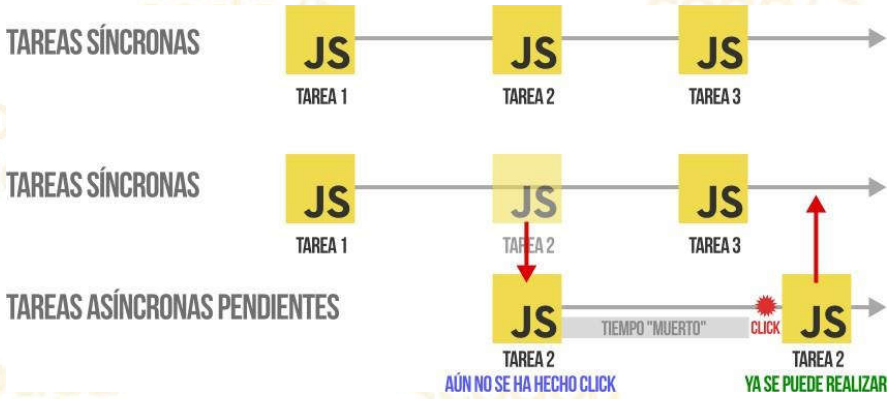
```
primera_funcion();   // Tarea 1: Se ejecuta primero

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 tareasólo cuando dicho suceso 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 resto de 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 hablaramosde un lenguaje bloqueante, hasta que el usuario no haga click, Javascript no seguiría ejecutando las 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 «prestándole atención» a medida que lo necesite, pudiendo continuar y retomar el resto de tareas a continuación de lasegunda.

¿Qué es la asincronía?

Pero esto no es todo. Ten en cuenta que pueden existir múltiples tareas asíncronas, dichas tareas puede que terminen realizandose 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 cuanto tiempo va a tardar enterminar una tarea, por lo que necesitamos un mecanismo para controlar todos estosfactores: las promesas, es uno de los ejemplos que veremos en esta unidad.

Ejemplos de tareas asíncronas

En Javascript no todas las tareas son asíncronas, pero hay ciertas tareas que sí lo son, y probablemente se entiendan mejor con ejemplos reales:

Un fetch() a una URL para obtener un archivo .json.

Un new Audio() de una URL con un .mp3 al que se hace .play() para reproducirlo.Una tarea programada con setTimeout() que se ejecutará en el futuro.

Una comunicación desde Javascript a la API de un sensor del smartphone.

Todos estos ejemplos se realizan mediante tareas asíncronas, ya que realizan un procedimiento que podría bloquear la ejecución del resto del programa al tardar mucho: la descarga de un fichero grande desde un servidor lento, una conexión a internet muy lenta, un dispositivo saturado a la hora de comunicarse con el sensor delmóvil, etc...

¿Cómo gestionar la asincronía?

Teniendo en cuenta el punto anterior, debemos aprender a buscar mecanismos para dejar claro 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 (que iremos viendo y profundizando en cada artículo deeste tema):

Método	Descripción
Mediante callbacks	Probablemente, la forma más clásica de gestionar la asincronía en 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.

Set TimeOut

Vamos a profundizar un poco con las funciones callbacks utilizadas para realizar tareas asíncronas. Probablemente, el caso más fácil de entender es utilizar un temporizador mediante la función setTimeout(callback, time).

Dicha función nos exige dos parámetros:

- La función callback a ejecutar
- El tiempo time que esperará antes de ejecutarlaEl

ejemplo sería el siguiente:

```
setTimeout(function() {
  console.log("He ejecutado la función");
}, 2000);
```

Simplemente, le decimos a setTimeout() que ejecute la función callback que le hemos pasado por primer parámetro cuando transcurran 2000 milisegundos (es decir, 2 segundos). Utilizandoarrow functions se puede simplificar el callback y hacer mucho más «fancy» y legible:

```
setTimeout(() => console.log("He ejecutado la función"), 2000);
```

Si lo prefieres y lo ves más claro (no suele ser habitual en código Javascript, pero cuando se empieza suele resultar más fácil entenderlo) podemos guardar el callback en una constante:

```
const action = () => console.log("He ejecutado la
función");setTimeout(action, 2000);
```

En cualquiera de los casos, lo importante es darse cuenta que estamos usando una función callback para pasársela a setTimeout(), que es otra función. En este caso, se trata de

«programar» un suceso que ocurrirá en un momento conocido del futuro, pero muchas vecesdesconoceremos cuando se producirá (o incluso si se llegará a producir).

Si probamos el código que verás a continuación, comprobarás que el segundo console.log() se ejecutará antes que el primero, dentro del setTimeout(), mostrando primero Código síncrono yluego Código asíncrono en la consola del navegador:

```
setTimeout(() => console.log("Código asíncrono."),

2000);console.log("Código síncrono.");
```

El último console.log del código se ejecuta primero (forma parte del flujo principal de ejecución del programa). El setTimeout() que figura en una línea anterior, aunque se ejecuta antes, pone en espera a la función callback, que se ejecutará cuando se cumpla una cierta condición (transcurran 2 segundos desde ese momento).

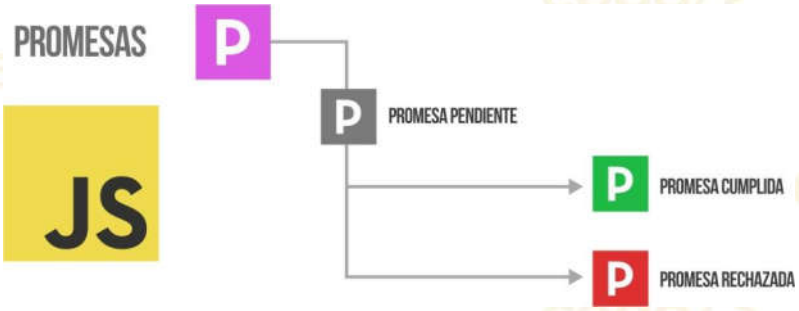
Esto puede llegar a sorprender a desarrolladores que llegan de otros lenguajes considerados bloqueantes; Javascript sin embargo se considera un lenguaje asíncrono y no bloqueante.

¿Qué significa esto? Al ejecutar la línea del setTimeout(), el programa no se queda bloqueadoesperando a que terminen los 2 segundos y se ejecute la función callback, sino que continúa con el flujo general del programa para volver más adelante cuando sea necesario a ejecutar elcallback, aprovechando así mejor el tiempo y realizando tareas de forma asíncrona.

Promesas

Las promesas son un concepto para resolver el problema de asincronía de una forma mucho más elegante y práctica que, por ejemplo, utilizando funciones callbacks directamente.

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)

Con estas sencillas bases, podemos entender el funcionamiento de una promesa en Javascript. Antes de empezar, también debemos tener claro que existen dos partes importantes de las promesas: como consumirlas (utilizar promesas) y como crearlas (preparar una función para que use promesas y se puedan consumir).

Promesas en Javascript

Las promesas en Javascript se representan a través de un , y cada promesa estará en 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 se cumple.
.catch(reject)	Ejecuta la función callback reject cuando la promesa se rechaza.
.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.

Consumir una promesa

La forma general de consumir una promesa es utilizando el .then() con un sólo parámetro, puesto que muchas veces lo único que nos interesa es realizar una acción cuando la promesase cumpla:

```
fetch("/robots.txt").then(function(response) {
  /* Código a realizar cuando se cumpla la promesa */

});
```

Lo que vemos en el ejemplo anterior es el uso de la función fetch(), la cuál devuelve una promesa que se cumple cuando obtiene respuesta de la petición realizada. De esta forma, estaríamos preparando (de una forma legible) la forma de actuar de nuestro código a la respuesta de la petición realizada, todo ello de forma asíncrona.

Recuerda que podemos hacer uso del método .catch() para actuar cuando se rechaza una promesa:

```
fetch("/robots.txt")
 .then(function(response) {
   /* Código a realizar cuando se cumpla la promesa */
 })
 .catch(function(error) {
   /* Código a realizar cuando se rechaza la promesa */
 });
```

Observa como hemos indentado los métodos .then() y .catch(), ya que se suele hacer así paraque sea mucho más legible para él. Además, se pueden encadenar varios .then() si se

siguen generando promesas y se devuelven con un return:

```
fetch("/robots.txt")
  .then(response => {
    return response.text(); // Devuelve una promesa
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => { /* Código a realizar cuando se rechaza la promesa */ });
```

No olvides indicar el return para poder encadenar las siguientes promesas con .then(). Tras un
.catch() también es posible encadenar .then() para continuar procesando promesas.

De hecho, usando arrow functions se puede mejorar aún más la legibilidad de este código, recordando que cuando sólo tenemos una sentencia en el cuerpo de la arrow function hay unreturn implícito:

```
fetch("/robots.txt")
  .then(response => response.text())
  .then(data => console.log(data))
  .finally(() => console.log("Terminado."))
  .catch(error => console.error(data));
```

Observese además que hemos añadido el método .finally() para añadir una función callbackque se ejecutará tanto si la promesa se cumple o se rechaza, lo que nos ahorrará tener que repetir la función en el .then() como en el .catch().

En todo este apartado hemos visto como utilizar o consumir una promesa haciendo uso de .then(), que es lo que en la mayoría de los casos necesitaremos.