**Asincronía en JavaScript**

La asincronía es un concepto clave en JavaScript, que se refiere a la capacidad del lenguaje de ejecutar tareas de manera no bloqueante. En otras palabras, JavaScript puede iniciar una operación y continuar con otras tareas sin esperar a que esa operación termine. Esto es especialmente útil cuando se trabajan con tareas que pueden tardar mucho tiempo, como solicitudes HTTP, lectura de archivos o temporizadores.

Por ejemplo, si JavaScript tuviera que esperar a que una solicitud HTTP se complete antes de ejecutar el siguiente código, el resto de la aplicación se bloquearía, lo que no es deseable. La asincronía permite manejar esas situaciones de forma eficiente.

**¿Cómo se maneja la asincronía en JavaScript?**

JavaScript maneja la asincronía mediante **callbacks**, **promesas** y **async/await**. A continuación, exploraremos estas herramientas.

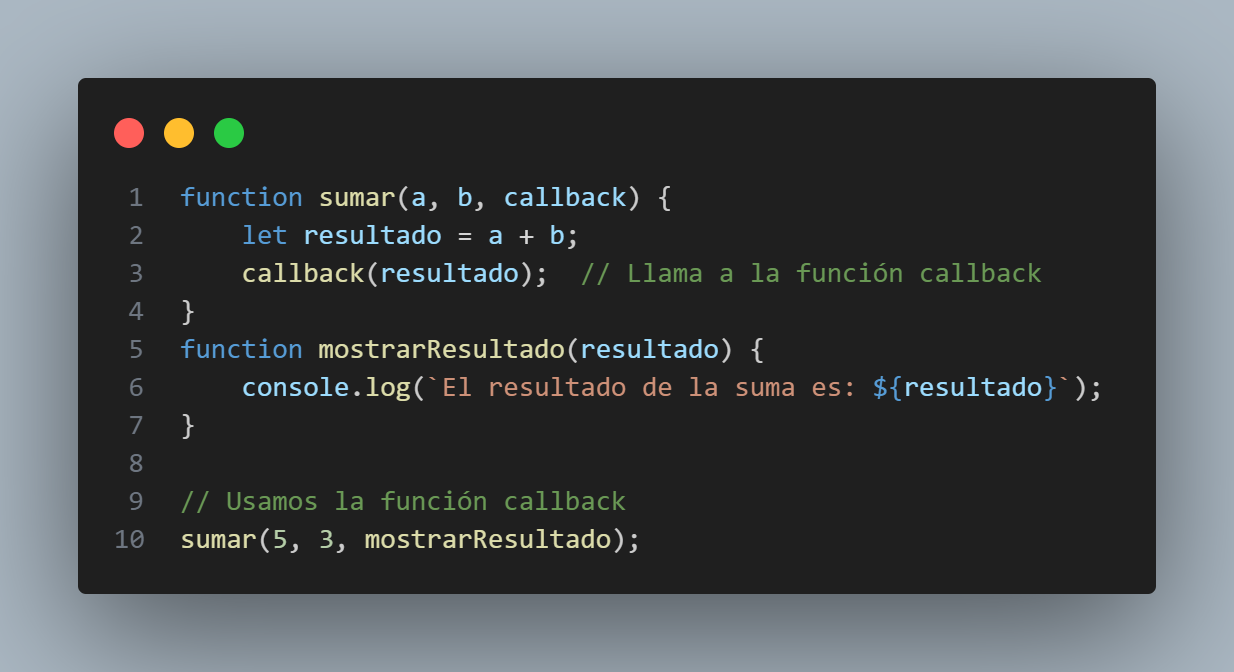
**Funciones Callback**

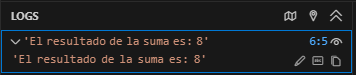
Una **función callback** es una **función que se pasa como argumento a otra función**, y esa función es **ejecutada dentro de la función externa** en un momento posterior. En otras palabras, una función callback es una forma de decirle a una función que **haga algo más** después de que se complete una tarea.

Las callbacks son fundamentales cuando se trata de **ejecutar código después de que ocurra un evento** o se complete una operación asincrónica.

#### Ejemplo:

Imagina que tienes una función sumar que necesita realizar una operación asincrónica y luego ejecutar una acción al completar esa operación. La función callback permite ejecutar esa acción una vez que la tarea asincrónica haya terminado.





**Explicación:**

* En este ejemplo, sumar es una función que toma dos números y una **función callback** como parámetros.
* Después de hacer la suma, sumar llama a la función mostrarResultado pasándole el resultado de la operación.
* La función mostrarResultado se ejecuta cuando se le pasa el resultado y muestra ese resultado en la consola.

### Uso de funciones Callback en situaciones asincrónicas

Las funciones callback son particularmente útiles cuando se trabajan con operaciones asincrónicas. Veamos un ejemplo práctico donde usamos una función callback para simular una operación asincrónica.

#### Ejemplo de manejo de eventos con Callback:

Supongamos que tenemos un botón en una página web, y queremos ejecutar algo cuando el usuario haga clic en ese botón. El clic es un evento asincrónico, y necesitamos una función callback que se ejecute cuando ese evento ocurra.



**Explicación:**

* En este ejemplo, addEventListener es una función que toma dos parámetros:
  1. El tipo de evento ("click").
  2. La función que debe ejecutarse cuando se dispare ese evento, que en este caso es la función manejarClic.
* manejarClic es una función **callback** que se ejecutará cuando el usuario haga clic en el botón.

### ¿Por qué las Callbacks pueden ser problemáticas?

Aunque las funciones callback son poderosas, **pueden llevar a ciertos problemas**, especialmente cuando las operaciones son complejas y dependen de varias tareas asincrónicas. Uno de los problemas más comunes con las callbacks es el fenómeno conocido como **Callback Hell** (Infierno de Callbacks).

### ****Callback Hell (Infierno de Callbacks)****

El **Callback Hell** ocurre cuando se tienen muchas funciones anidadas dentro de otras funciones. Esto puede hacer que el código sea difícil de leer, mantener y depurar.

Existen varias maneras de evitar el **Callback Hell**. Las dos principales soluciones son:

1. **Usar funciones auxiliares** para hacer el código más modular y legible.
2. **Usar Promesas** o la sintaxis async/await (en versiones modernas de JavaScript), lo que simplifica mucho el manejo de operaciones asincrónicas y hace que el código sea más fácil de leer.

### setTimeout()

La función **setTimeout()** permite ejecutar una función después de un retraso especificado en milisegundos. Este es un temporizador único, lo que significa que **solo se ejecutará una vez** después de que haya pasado el tiempo de espera.

#### Sintaxis:

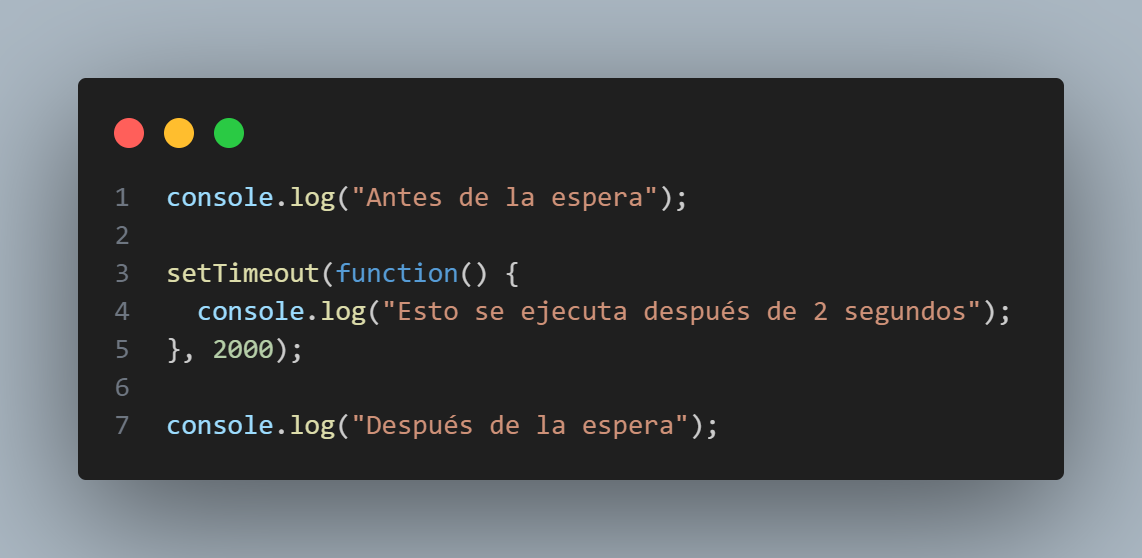
setTimeout(función, tiempoEnMilisegundos);

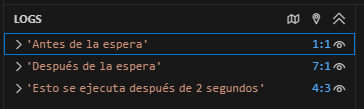
 **función**: La función o el código que deseas ejecutar después de que haya pasado el tiempo de espera.

 **tiempoEnMilisegundos**: El tiempo de retraso en milisegundos (1 segundo = 1000 milisegundos).

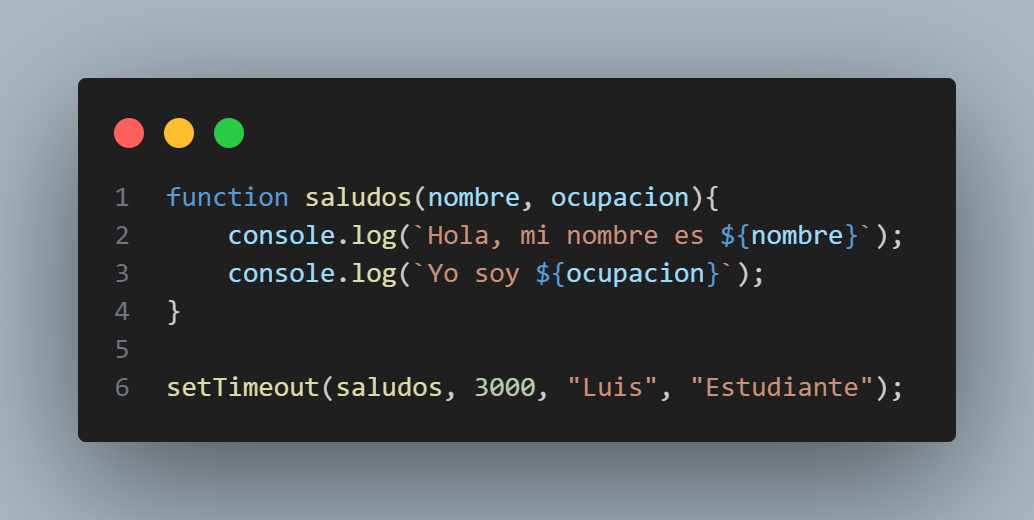
Es posible **cancelar** un setTimeout() antes de que se ejecute utilizando clearTimeout(). Esto es útil cuando tienes una condición para detener el temporizador antes de que ocurra la ejecución de la función.

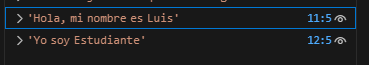
Ejemplo:





Otra forma de usar setTimeOut() es primero declarar una función y luego llamarla usando setTimeOut(nombreFuncion, tiempo, parametros)





**Explicación:**

* La función se ejecuta después de 3 segundos

Promesas

Las **promesas** en JavaScript son un mecanismo esencial para manejar la asincronía de manera más estructurada y legible que usando callbacks. Son una forma de representar el resultado de una operación asincrónica que puede completarse exitosamente o fallar en el futuro.

### setInterval()

La función **setInterval()** es similar a setTimeout(), pero con una diferencia clave: **se ejecuta de manera repetitiva** en intervalos de tiempo especificados en milisegundos. Es decir, después de que se haya ejecutado la función una vez, vuelve a ejecutarse después del tiempo indicado y continuará haciéndolo indefinidamente hasta que se detenga manualmente.

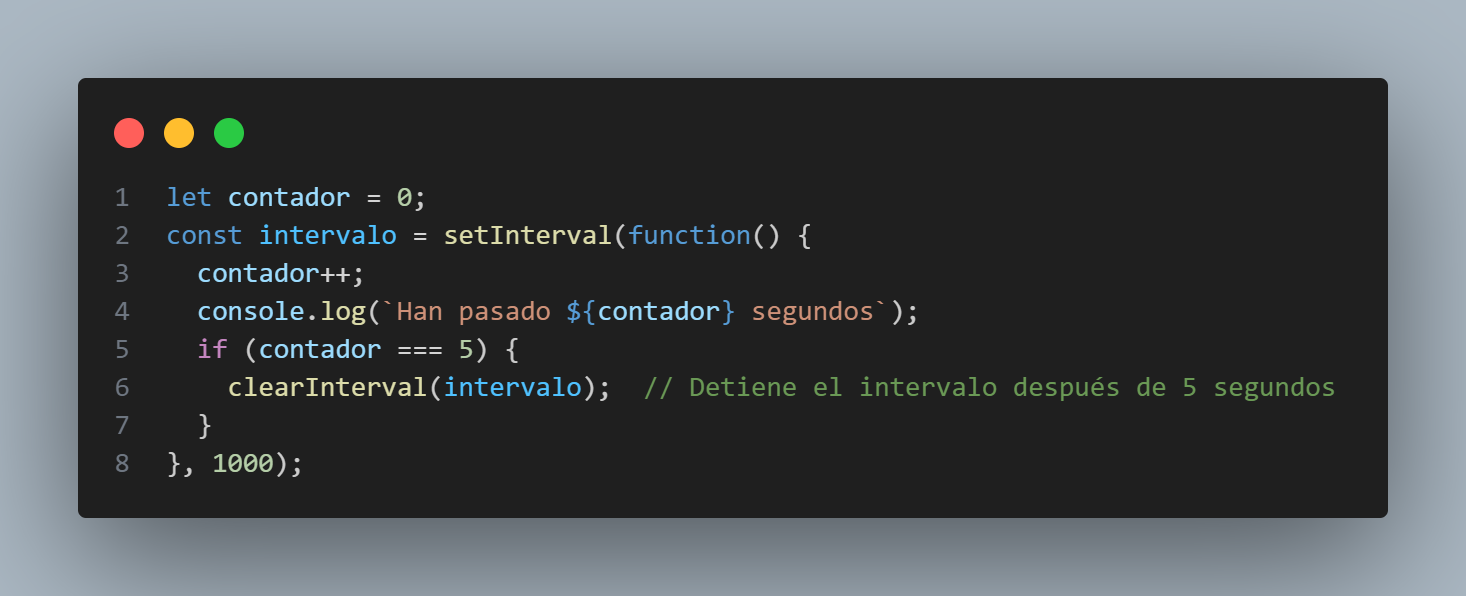
Sintaxis:

setInterval(función, intervalo);

 **función**: La función que deseas ejecutar repetidamente.

 **intervalo**: El intervalo de tiempo (en milisegundos) entre cada ejecución de la función.

Ejemplo:



**Explicación:**

* setInterval ejecuta la función cada 1000 milisegundos (1 segundo) e incrementa un contador, imprimiendo el tiempo transcurrido.
* Después de 5 segundos, se detiene el intervalo usando clearInterval.

**¿Qué es una Promesa?**

Una **promesa** en JavaScript es un objeto que representa el eventual resultado de una operación asincrónica. Es un "contrato" entre el código que genera la promesa (el productor) y el código que la maneja (el consumidor). Este contrato tiene tres estados posibles:

1. **Pendiente (Pending)**: El estado inicial de la promesa, donde la operación asincrónica aún no ha terminado.
2. **Cumplida (Fulfilled)**: La operación asincrónica ha terminado correctamente, y la promesa tiene un valor resultante que se puede usar.
3. **Rechazada (Rejected)**: La operación asincrónica falló, y la promesa tiene un motivo (un error o una razón) que explica por qué falló.

La promesa proporciona un mecanismo para manejar estos tres estados de manera eficiente.

**Creación de una Promesa**

Para crear una promesa en JavaScript, usamos el constructor Promise, que toma una función llamada **executor** como argumento. Esta función recibe dos parámetros:

* **resolve**: Una función que se invoca cuando la operación asincrónica se completa con éxito. Su argumento será el valor con el que se resuelve la promesa.
* **reject**: Una función que se invoca cuando la operación asincrónica falla. Su argumento es la razón del rechazo (por ejemplo, un error).

**Consumiendo Promesas**

Una vez que creamos una promesa, el siguiente paso es **manejar su resolución** o **rechazo** usando los métodos .then() y .catch().

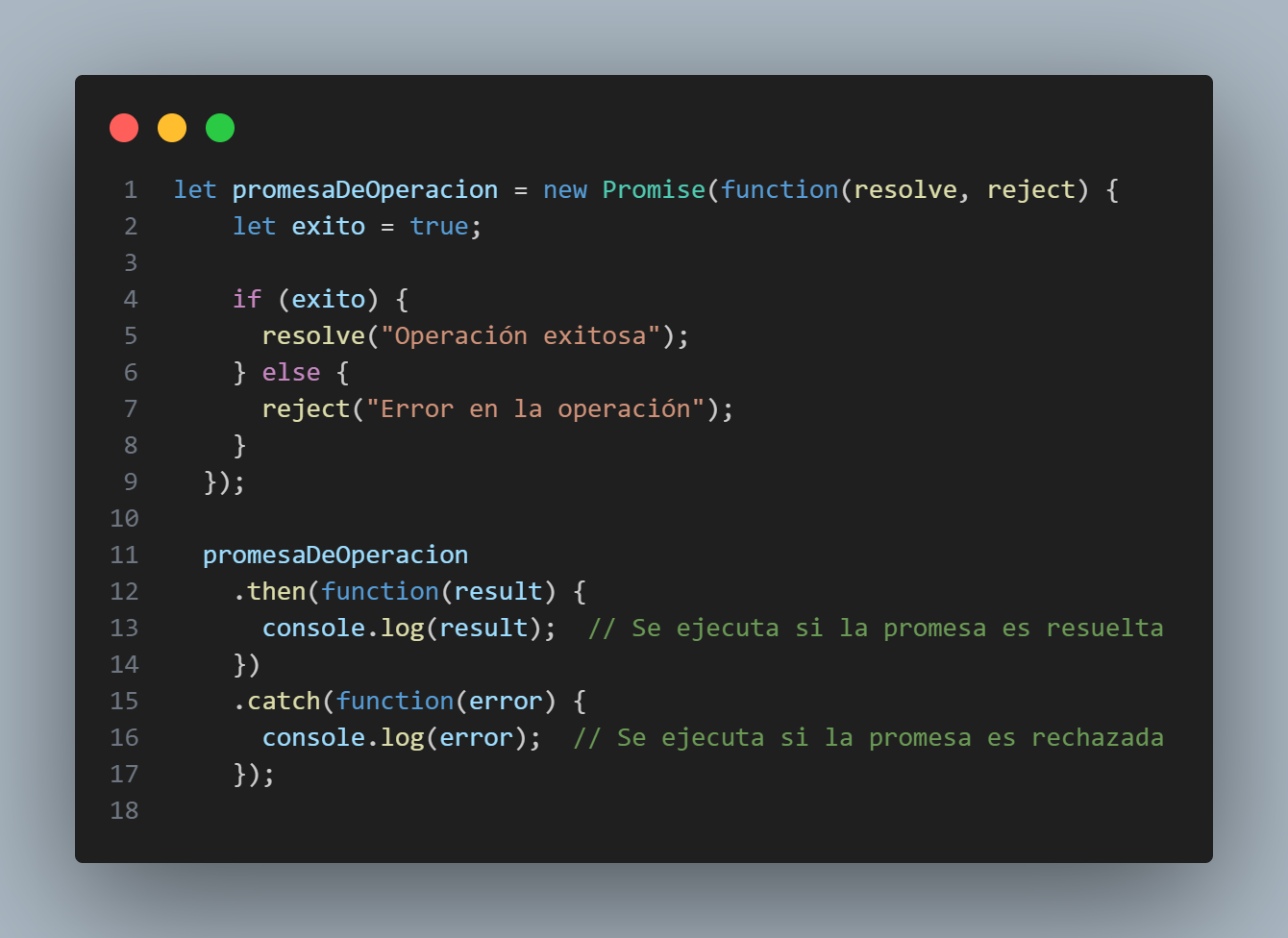
**Método .then()**

* El método .then() se utiliza para manejar el caso en que la promesa se resuelve exitosamente.
* Recibe dos funciones como argumentos:
  + La primera función es llamada cuando la promesa se resuelve (cumplida), y recibe el valor con el que la promesa fue resuelta.
  + La segunda función es llamada cuando la promesa se rechaza, y recibe el motivo del rechazo.

**Método .catch()**

* El método .catch() es una forma simplificada de manejar un rechazo de la promesa.
* Es equivalente a pasar solo la función de rechazo a .then(), pero se utiliza para mejorar la legibilidad y hacer más claro que se trata del manejo de errores.

Ejemplo básico:





Ejemplo Practico:

Simular el proceso de un pago, que puede tardar un poco. La promesa se resolverá si el pago fue exitoso o se rechazará si hubo un error.





### Promesas Combinadas

JavaScript proporciona métodos adicionales para trabajar con múltiples promesas al mismo tiempo.

Promise.all()

Promise.all() toma un arreglo de promesas y retorna una nueva promesa que se resuelve cuando **todas las promesas** en el arreglo se resuelven. Si alguna promesa es rechazada, la promesa retornada por Promise.all() también se rechaza.

#### Ejemplo de Promise.all():





En este ejemplo:

* Ambas promesas se resuelven después de 1 y 2 segundos, respectivamente.
* Promise.all() espera a que ambas promesas se resuelvan y luego devuelve un arreglo con sus resultados.

**Promise.race()**

Promise.race() toma un arreglo de promesas y retorna una nueva promesa que se resuelve con el resultado de la **primera promesa** que se resuelva (ya sea cumplida o rechazada).





En este ejemplo:

* La segunda promesa se resuelve antes que la primera, por lo que Promise.race() devolverá "Segundo valor".

Procesos **async** y **await**

**async** y **await** son características de JavaScript que simplifican la escritura y comprensión del código asincrónico. Ambas fueron introducidas en **2017** y están diseñadas para trabajar con **Promesas**, facilitando la lectura y evitando los problemas comunes de la programación asincrónica, como el "callback hell" o infierno de los callbacks.

A continuación, te proporcionaré una explicación detallada de cómo funcionan **async** y **await**, junto con ejemplos prácticos y comparaciones con otras técnicas asincrónicas.

### 1. ****¿Qué es**** async****?****

El **modificador async** se usa para declarar una función como **asincrónica**. Esto significa que dentro de esa función podemos utilizar **await** para esperar a que una promesa se resuelva, y el código se ejecutará de manera secuencial, como si fuera sincrónico, pero sin bloquear el hilo principal.

Cuando declaramos una función como async, automáticamente esta función **devuelve una promesa**. Si dentro de la función hay valores retornados, **ese valor será envuelto en una promesa** de manera implícita.

#### Sintaxis de async:

async function nombreDeLaFuncion() { }

**¿Qué es await?**

El **operador await** solo puede ser utilizado dentro de funciones async y se utiliza para esperar a que una promesa se resuelva o rechace. await hace que el código **se "detenga" hasta que la promesa se resuelva**, pero no bloquea el hilo principal de ejecución de JavaScript, lo que significa que no bloquea la interfaz de usuario ni el procesamiento de otros eventos mientras espera.

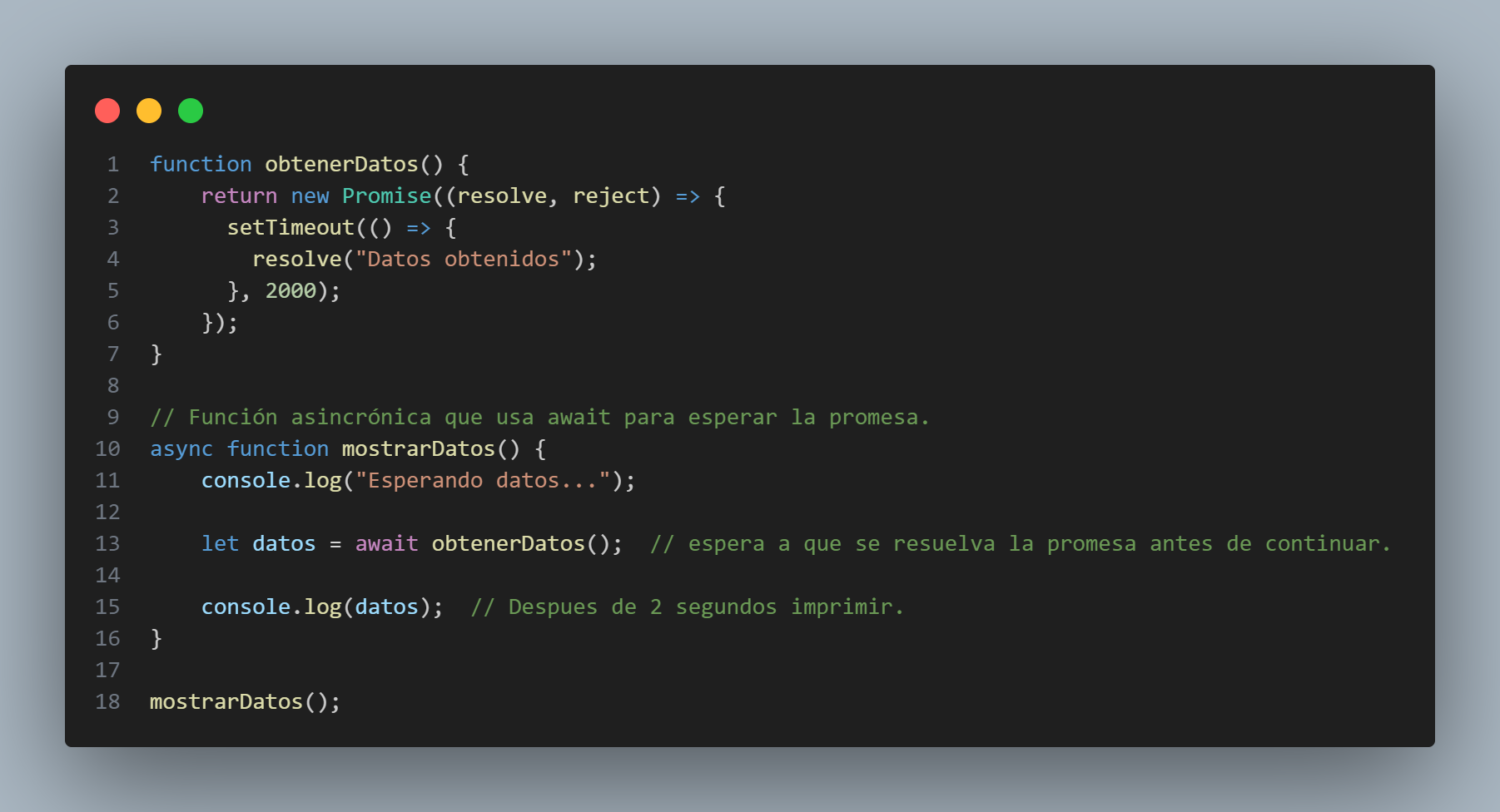
Cuando se usa await con una promesa:

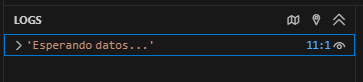
* Si la promesa se resuelve, **devuelve el valor resuelto**.
* Si la promesa se rechaza, **lanza un error**, que puede ser manejado con try...catch.

**Sintaxis de await:**

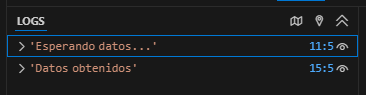
let resultado = await promesa;

Cuando se combina async con await, el flujo del código se vuelve mucho más claro y fácil de leer que con los métodos tradicionales basados en callbacks o promesas encadenadas. En lugar de anidar múltiples promesas o callbacks, async/await permite escribir código asincrónico de una manera similar a la programación sincrónica.





Despues de los ds segundos:



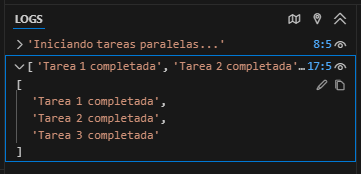
**Explicación**:

1. La función obtenerDatos() devuelve una promesa que se resuelve después de 2 segundos.
2. La función mostrarDatos() está marcada como async, lo que significa que puede usar await dentro de ella.
3. Cuando await obtenerDatos() es ejecutado, **la ejecución de mostrarDatos() se "detiene" en ese punto** hasta que la promesa devuelta por obtenerDatos() se resuelva.
4. Después de que la promesa se resuelve, el valor obtenido (en este caso, "Datos obtenidos") se asigna a la variable datos y se imprime.

Aunque async y await hacen que el código asincrónico se vea más secuencial, **a veces necesitamos ejecutar varias promesas de manera paralela**. En lugar de esperar una por una, podemos ejecutar varias promesas al mismo tiempo usando Promise.all().

#### Ejemplo con Promise.all:





**Explicación**:

1. En este ejemplo, las tres tareas (esperar 1, 2 y 1.5 segundos) se ejecutan en paralelo usando Promise.all().
2. await Promise.all([...]) espera a que **todas las promesas** se resuelvan y devuelve un array con los resultados de cada tarea.
3. El código sigue siendo sencillo de entender y se aprovecha de la ejecución paralela.

**¿Por qué usar async/await en lugar de promesas o callbacks?**

**Beneficios:**

1. **Código más limpio y legible**: async/await permite escribir código que parece sincrónico, pero mantiene el rendimiento asincrónico de JavaScript.
2. **Manejo de errores más claro**: Con try...catch, el manejo de errores en operaciones asincrónicas se vuelve mucho más simple y comprensible.
3. **Evita el Callback Hell**: En lugar de anidar promesas o callbacks, el código es más plano y secuencial.

**Comparación con promesas:**

Aunque las promesas también proporcionan una forma efectiva de manejar operaciones asincrónicas, **async/await** es más fácil de leer, escribir y entender, especialmente cuando tienes varias operaciones asincrónicas secuenciales.

**Comparación con callbacks:**

* Las callbacks pueden ser complicadas cuando tienes múltiples tareas asincrónicas, y las promesas ayudan a evitar este problema.
* async/await lleva este concepto un paso más allá, permitiendo un flujo de código que parece secuencial pero es asincrónico.