## **Ejemplo 01: Dos procesos con setTimeOut aislados**

Ejemplo 02: Mismo ejemplo que el anterior, pero una la invoca a la otra dentro

Ejemplo 03: Del cuerpo de la función

## **El Event Loop en JavaScript**

### ****¿Qué es el Event Loop?****

El **event loop** es el mecanismo fundamental que permite a JavaScript manejar tareas asincrónicas, como temporizadores, eventos del DOM, promesas y otras operaciones no bloqueantes. JavaScript es de naturaleza **single-threaded**, lo que significa que solo tiene un hilo de ejecución, por lo tanto, el **event loop** es esencial para permitir que las tareas asíncronas no bloqueen la ejecución del código.

### ****¿Cómo funciona el Event Loop?****

El event loop es el encargado de:

1. **Ejecutar el código sincrónico**.
2. **Escuchar** las tareas asincrónicas que se encuentran en la **cola de tareas (task queue)**.
3. Cuando el hilo principal está libre, el **event loop** toma las tareas pendientes en la cola y las ejecuta.

### ****Ciclo del Event Loop****:

1. **Stack de ejecución**: Aquí es donde se ejecutan las operaciones sincrónicas.
2. **Task Queue (Cola de tareas)**: Aquí es donde se almacenan las funciones asincrónicas listas para ejecutarse, como las callbacks de setTimeout, setInterval, eventos del DOM, y resoluciones de promesas.
3. **Event Loop**: Monitorea continuamente el stack de ejecución. Si el stack está vacío, toma tareas de la **task queue** y las ejecuta.

### ****¿Qué tareas entran al Event Loop?****

1. **Callbacks de temporizadores** (setTimeout, setInterval): Estas tareas son puestas en la **cola de tareas** después de que el temporizador expira y el stack de ejecución está vacío.
2. **Eventos del DOM**: Los eventos como click, scroll, input también se colocan en la cola de tareas y se ejecutan cuando el hilo está libre.
3. **Promesas**: Cuando una promesa se resuelve o se rechaza, la función que maneja el resultado se coloca en la cola de tareas.
4. **AJAX y Fetch**: Las respuestas de peticiones asíncronas se manejan de manera similar, cuando los datos están disponibles, las tareas se colocan en la cola de tareas.

### ****¿Qué NO entra en el Event Loop?****

1. **Código sincrónico**: Todo el código que se ejecuta de inmediato bloquea el hilo principal y no entra en el event loop. Ejemplos son bucles for, funciones sincrónicas largas, o llamadas recursivas.
2. **Operaciones que bloquean el hilo**: Cualquier operación pesada o que consuma mucho tiempo en el stack de ejecución bloquea el flujo del event loop.

### ****Relación entre Sincronismo y el Event Loop****:

El código sincrónico siempre se ejecuta primero, bloqueando el hilo principal. Las tareas asincrónicas solo se ejecutan cuando el event loop detecta que el stack de ejecución está vacío. Por eso, aunque agendes tareas con setTimeout o setInterval, estas no podrán ejecutarse hasta que el código sincrónico finalice.

# **Sincronismo y Asincronismo en JavaScript**

## **1. Introducción a Sincronismo y Asincronismo**

En JavaScript, el **sincronismo** y el **asincronismo** se refieren a cómo se manejan las tareas o instrucciones en el tiempo. Es importante comprender cómo funcionan las operaciones en cada uno de estos paradigmas para escribir código eficiente.

### ****1.1 Instrucciones Sincrónicas****

Las **instrucciones sincrónicas** son aquellas que se ejecutan **de forma secuencial**, es decir, el código se ejecuta línea por línea, y una instrucción no se ejecuta hasta que la anterior haya terminado. Durante las operaciones sincrónicas, el hilo principal queda bloqueado hasta que la tarea se completa.

#### **Ejemplos de operaciones sincrónicas:**

* **Operaciones matemáticas sencillas:**

js

Copiar código

let resultado = 2 + 3; // Se ejecuta de inmediato

console.log(resultado); // Muestra el resultado inmediatamente

* **Acceso sincrónico al DOM:**

js

Copiar código

const elemento = document.querySelector("div"); // El DOM se accede de inmediato

* **Bucles (for, while, etc.):**

js

Copiar código

for (let i = 0; i < 100; i++) {

console.log(i); // Cada iteración debe completarse antes de continuar

}

### ****1.2 Instrucciones Asincrónicas****

Las **instrucciones asincrónicas** permiten que JavaScript ejecute tareas de larga duración sin bloquear el hilo principal. Mientras una tarea asincrónica se ejecuta, el código sigue avanzando, y cuando la tarea asincrónica se completa, una **callback**, **promesa**, o **async/await** maneja el resultado.

#### **Ejemplos de operaciones naturalmente asincrónicas:**

* **Temporizadores (setTimeout, setInterval):**

js

Copiar código

setTimeout(() => {

console.log("Esta línea se ejecuta después de 2 segundos");

}, 2000);

* **Solicitudes HTTP (fetch o XMLHttpRequest):**

js

Copiar código

fetch('https://api.example.com/data')

.then(response => response.json())

.then(data => console.log(data)); // Esto ocurre después de recibir la respuesta

* **Eventos del DOM (click, input, etc.):**

js

Copiar código

document.querySelector("button").addEventListener("click", () => {

console.log("Botón clickeado");

});

* **Operaciones con Promise y async/await:**

js

Copiar código

async function obtenerDatos() {

const respuesta = await fetch('https://api.example.com/data');

const datos = await respuesta.json();

console.log(datos);

}

obtenerDatos();

### ****1.3 Problema del Sincronismo:****

* En JavaScript, si ejecutas tareas **sincrónicas** que requieren mucho tiempo, como un bucle o procesamiento de grandes volúmenes de datos, **bloquearás el hilo principal**, lo que impedirá que el navegador responda a otras interacciones (como eventos del usuario).

### ****1.4 Necesidad del Asincronismo:****

* **Asincronismo** es clave para tareas como:
  + Solicitudes HTTP (fetch de datos de un servidor).
  + Manipulación de archivos grandes o cálculo de operaciones pesadas.
  + Eventos de entrada del usuario.
  + Animaciones que no bloqueen el flujo normal de la aplicación.

## **2. Técnicas para manejar el Asincronismo en JavaScript**

### ****2.1 Callbacks****

Una **callback** es una función que se pasa como argumento a otra función para ser ejecutada una vez que la tarea principal haya terminado. Es una de las técnicas más antiguas para manejar el asincronismo en JavaScript.

#### **Ejemplo de callback asincrónica:**

js

Copiar código

function procesoAsincrono(callback) {

setTimeout(() => {

console.log("Proceso asincrónico completado");

callback();

}, 2000);

}

console.log("Inicio");

procesoAsincrono(() => {

console.log("Callback ejecutada");

});

console.log("Fin");

**Resultado esperado**:

Copiar código

Inicio

Fin

Proceso asincrónico completado

Callback ejecutada

#### **Problema: Callback Hell**

Las callbacks anidadas pueden hacer que el código sea difícil de leer y mantener, generando lo que se conoce como **Callback Hell**.

js

Copiar código

primeraFuncion(() => {

segundaFuncion(() => {

terceraFuncion(() => {

cuartaFuncion(() => {

// Más y más anidaciones...

});

});

});

});

### ****2.2 Promesas (Promises)****

Las **promesas** proporcionan una forma más estructurada y manejable de trabajar con código asincrónico. Una promesa representa un valor que estará disponible ahora, en el futuro, o nunca.

#### **Sintaxis básica de una Promesa:**

js

Copiar código

const promesa = new Promise((resolve, reject) => {

setTimeout(() => {

const exito = true;

if (exito) {

resolve("La promesa se resolvió correctamente");

} else {

reject("Hubo un error en la promesa");

}

}, 2000);

});

promesa

.then((resultado) => {

console.log(resultado); // Manejo de éxito

})

.catch((error) => {

console.log(error); // Manejo de error

});

#### **Ventajas de las Promesas:**

* Son encadenables usando .then(), lo que permite ejecutar tareas de forma secuencial sin caer en el **Callback Hell**.

#### **Encadenamiento de promesas:**

js

Copiar código

primeraPromesa()

.then((resultado1) => {

return segundaPromesa(resultado1);

})

.then((resultado2) => {

return terceraPromesa(resultado2);

})

.then((resultado3) => {

console.log("Proceso completado", resultado3);

});

### ****2.3 Async y Await****

**async** y **await** son una sintaxis moderna para trabajar con promesas, que hace que el código asincrónico sea más fácil de leer y escribir. **async** convierte una función en una función que devuelve una promesa, y **await** pausa la ejecución hasta que la promesa se resuelva.

#### **Ejemplo con** async **y** await**:**

js

Copiar código

async function obtenerResultado() {

console.log("Inicio");

const resultado = await new Promise((resolve) => {

setTimeout(() => {

resolve("Proceso completado");

}, 2000);

});

console.log(resultado); // Espera a que la promesa se resuelva

console.log("Fin");

}

obtenerResultado();

**Resultado esperado**:

Copiar código

Inicio

Proceso completado

Fin

#### **Ventajas de** async/await**:**

* Hace que el código asincrónico sea más legible y fácil de seguir, como si fuera código sincrónico.
* Simplifica el manejo de errores utilizando bloques **try/catch**.

#### **Manejo de errores con** async/await**:**

js

Copiar código

async function obtenerDatos() {

try {

const respuesta = await fetch('https://api.example.com/data');

const datos = await respuesta.json();

console.log(datos);

} catch (error) {

console.error("Error:", error);

}

}

## **3. Comparativa entre Callbacks, Promesas y** async/await

| **Característica** | **Callbacks** | **Promesas** | **async/await** |
| --- | --- | --- | --- |
| **Complejidad** | Alta con varias anidaciones | Moderada con encadenamiento | Baja, más clara y secuencial |
| **Legibilidad** | Difícil de leer | Legible, pero con .then() | Muy legible, similar al código sincrónico |
| **Manejo de errores** | Manual | .catch() | Bloques try/catch |
| **Uso moderno** | En desuso por su complejidad | Amplio uso | Preferido en JavaScript moderno |

## **4. Conclusión**

* **Instrucciones sincrónicas** se ejecutan de manera secuencial, bloqueando el hilo principal.
* **Instrucciones asincrónicas** permiten ejecutar tareas sin bloquear el hilo, usando técnicas como callbacks, promesas y async/await.
* **Callbacks** fueron el enfoque original, pero las **promesas** y **async/await** son las técnicas preferidas hoy en día por su simplicidad y claridad.

Exactamente, **JavaScript es naturalmente sincrónico**, lo que significa que ejecuta una línea de código a la vez, de manera secuencial, bloqueando el hilo principal hasta que cada tarea se complete. Sin embargo, **para lograr asincronismo** y permitir que se ejecuten tareas de larga duración sin bloquear la interfaz o el flujo del programa, se utilizan técnicas como:

* **Promesas (Promise)**: Para manejar tareas asincrónicas de una manera más estructurada, evitando el **Callback Hell**.
* **async y await**: Una forma más moderna y legible de trabajar con promesas, haciendo que el código asincrónico se vea y se comporte como si fuera sincrónico.
* **Temporizadores (setTimeout() y setInterval())**: Para ejecutar tareas después de un cierto período de tiempo, liberando el hilo principal mientras tanto.

**Conclusión:**

JavaScript es naturalmente **sincrónico**. Para manejar tareas asincrónicas y evitar el bloqueo del hilo principal, utilizamos técnicas como **promesas**, **async/await**, y **temporizadores** (setTimeout()/setInterval()), que nos permiten ejecutar tareas en segundo plano o en un futuro cercano sin detener la ejecución del resto del código.

Estas herramientas hacen que JavaScript pueda manejar tareas complejas, como solicitudes a servidores o procesos intensivos, sin comprometer la interacción del usuario y la eficiencia del sistema.