<codoacodo/>

Agencia de Aprendizaje a lo largo de la vida

FULL STACK FRONTEND
Clase 19

Javascript 7

BA

Fundamentos del asincronismo en Javascript (Parte I)

Les damos la bienvenida

Vamos a comenzar a grabar la clase

Clase 18

DOM y Eventos

- Manipulación del DOM.
- Definición, alcance y su importancia..
- Eventos en JS.
- Eventos. ¿Qué son, para qué sirven y cuáles son los más comunes?
- Escuchar un evento sobre el DOM.

Clase 19

Asincronismo (Parte I)

- Introducción
- Sincronismo y Asincronismo
- Promesas
- Funciones del constructor y métodos.

Clase 20

Asincronismo (Parte II)

- Introducción
- Callbacks
- async / await
- Manejo de errores con try/catch en funciones asincronas

¿Qué es el sincronismo?

En JavaScript, el sincronismo se refiere a la ejecución ***secuencial de código***, donde cada línea de código espera a que la línea anterior se haya completado antes de ejecutarse.

Este modelo de ejecución ***es lineal y bloqueante.*** Significa que si una línea de código toma tiempo para completarse por algún motivo, toda la ejecución del programa se detiene hasta que esa línea termine.

Dos características de este modelo son:

El ***Bloqueo:*** El modelo sincrónico puede conducir a problemas de bloqueo de la interfaz de usuario, especialmente en aplicaciones web donde una tarea larga puede hacer que la página no responda.

La ***Previsibilidad:*** el código sincrónico es generalmente más fácil de leer, escribir y depurar debido a su naturaleza secuencial y predecible.

Ejemplo sincrónico

En este ejemplo, cada operación se completa antes de que la siguiente comience. Aquí, imprimiremos mensajes en la consola uno tras otro.

```
console.log('Primero: Este mensaje se muestra primero.');
console.log('Segundo: Este mensaje se muestra después, sin demora.');
```

En este ejemplo, "Primero" se imprime primero, seguido inmediatamente por "Segundo".

No hay retraso entre las operaciones: todo ocurre en un orden estricto y secuencial.

Otro ejemplo sincrónico

```
console.log('Inicio del proceso');

let suma = 0;
for (let i = 0; i < 1000000; i++) {
    suma += i;
}

console.log('Suma completa: ', suma);
console.log('Fin del proceso');
```

En este código, el mensaje "Fin del proceso" no se imprimirá hasta que el bucle haya terminado de calcular la suma, demostrando el comportamiento bloqueante y secuencial del código sincrónico.

¿Qué es el asincrónismo?

Es la capacidad de ejecutar ciertas operaciones en ***segundo plano*** y continuar realizando otras tareas ***sin tener que esperar a que la operación asincrónica termine.***

En JavaScript, esto es esencial debido a que es un lenguaje de programación de un solo hilo, lo que significa que puede ejecutar un solo bloque de código a la vez. El asincronismo permite que JavaScript maneje tareas como cargar datos, procesar archivos grandes, o realizar solicitudes de ***red sin bloquear la interfaz de usuario o detener otras operaciones***. Utiliza funciones como ***callbacks, promesas y async/await*** para gestionar comportamientos asíncronos, permitiendo que las aplicaciones sean más rápidas y responsivas.

Asincronismo en JavaScript

Imagina que estás organizando una gran fiesta y has decidido enviar invitaciones por correo electrónico a todos tus amigos. La tarea de enviar cada invitación es similar a una operación que lleva tiempo, como cargar datos o hacer una solicitud a una base de datos.

En este ejemplo, JavaScript actúa como el organizador de la fiesta. Sin asincronismo, JavaScript tendría que enviar cada invitación uno por uno, esperando que el correo electrónico de un amigo se envíe antes de poder empezar a enviar el siguiente. Esto significa que, mientras espera, no podría hacer nada más, como planificar la música o la comida de la fiesta, lo que no es eficiente.

Asincronismo en JavaScript

Ahora, gracias al asincronismo en JavaScript, la situación es muy diferente:

Inicio del Proceso: JavaScript comienza el proceso de enviar las invitaciones. Para cada amigo, inicia el envío de un correo electrónico.

Asincronismo en Acción: Mientras los correos electrónicos se están enviando (algo que toma tiempo y se hace a través de la red), JavaScript no necesita esperar que cada uno se complete. En lugar de eso, puede seguir trabajando en otros preparativos para la fiesta, como elegir la música o planear los juegos.

Asincronismo en JavaScript

Manejo de Respuestas: Cada vez que una invitación se envía correctamente, JavaScript recibe una notificación (esto se puede manejar con promesas o callbacks en un entorno de programación real). Si algún correo electrónico falla, puede manejar esos casos individualmente, quizás intentando enviar la invitación nuevamente o eligiendo un método alternativo de contacto.

Continúa con otras Tareas: Mientras las respuestas de las invitaciones llegan de manera asincrónica, JavaScript no se ha detenido ni un segundo. Ha estado libre para hacer todas las otras tareas necesarias, asegurándose de que la fiesta sea un éxito.

Ejemplo asincrónico

Para el ejemplo asíncrono, usaremos setTimeout para introducir un retraso antes de imprimir el segundo mensaje, mostrando cómo JavaScript puede manejar operaciones que no se completan inmediatamente:

```
console.log('Primero: Este mensaje se muestra primero.');

setTimeout(() => {
  console.log('Tercero: Este mensaje se muestra después de 2 segundos.');
}, 2000);

console.log('Segundo: Este mensaje se muestra mientras esperamos el
temporizador.');
```

¿Qué son las promesas?

Son un mecanismo para manejar operaciones asíncronas. Permiten manejar el eventual resultado de una operación asincrónica, ya sea un éxito o un fallo.

Las promesas representan un valor que puede no estar disponible aún, pero que eventualmente se resolverá o rechazará, permitiendo que el código continúe ejecutándose en paralelo sin bloquear la ejecución principal.

¿Para qué se utilizan las Promesas?
Simplifican el manejo de secuencias de operaciones asíncronas y el manejo del flujo de ejecución en escenarios donde se depende de datos o eventos que tardan tiempo en completarse, como:

- Solicitudes de red (p.ej., AJAX/fetch).
- Lecturas de archivos en aplicaciones basadas en Node.js.
- Operaciones de bases de datos.
- Operaciones que requieran esperar por un evento o respuesta antes de continuar.

Características de las promesas:

Asíncronas: Permiten que el código continúe ejecutándose mientras la operación completa su ejecución.

Evitan el Callback Hell: Proporcionan una alternativa más limpia y manejable a los callbacks anidados.

Encadenamiento: Las promesas pueden ser encadenadas, lo que permite realizar varias operaciones asíncronas secuenciales de manera más legible.

Cómo crear una promesa:

```
const promesa = new Promise((resolve, reject) => {
    // Simulamos un valor para que la condición se cumpla (o no)
    const x = 42;
    // Operación asincrónica simulada
    setTimeout(() => {
        if (x === 42) {
            resolve("¡Éxito!"); // La promesa se resuelve exitosamente
        } else {
            reject("Fallo."); // La promesa falla y se rechaza
        }
    }, 2000);
});
```

new Promise(executor): El constructor Promise se utiliza para crear una nueva promesa. Recibe una función ejecutora como argumento, que a su vez recibe dos funciones: resolve y reject.

executor: Es la función que se ejecuta inmediatamente por el constructor Promise, antes incluso de que el constructor Promise devuelva el objeto promesa recién creado. Esta función generalmente inicia algún trabajo asíncrono y luego, dependiendo del resultado de ese trabajo, llama a **resolve** o a **reject**.

Funciones dentro del Constructor

```
const promesa = new Promise((resolve, reject) => {
    // Simulamos un valor para que la condición se cumpla (o no)
    const x = 42;
    // Operación asincrónica simulada
    setTimeout(() => {
        if (x === 42) {
            resolve("¡Éxito!"); // La promesa se resuelve exitosamente
        } else {
            reject("Fallo."); // La promesa falla y se rechaza
        }
    }, 2000);
});
```

resolve(value): Esta función se llama con un valor cuando la operación asíncrona se completa con éxito. Al llamar a **resolve**, la promesa **pasa de estar en estado "pendiente" a "cumplida".**
El valor pasado a **resolve** es lo que se maneja con el método **.then** de la promesa.

reject(reason): Esta función se llama con una razón o error cuando la operación asíncrona falla o se encuentra con un error. Al llamar a reject, la **promesa pasa de estar en estado "pendiente" a "rechazada".** La razón pasada a reject es lo que se maneja con el método .catch de la promesa.

Métodos de la promesa

```
promesa.then((valor) => {
    console.log(valor); // Se ejecuta si la promesa se resuelve exitosamente
}).catch((error) => {
    console.error(error); // Se ejecuta si la promesa se rechaza
}).finally(() => {
    console.log('Operación completada.'); // Se ejecuta al finalizar la resolución
});
```

.then(): Este método se ejecuta si la promesa se resuelve exitosamente. Valor recibe el valor que fue pasado a resolve cuando la promesa fue cumplida. En este caso, imprime el valor resuelto de la promesa a la consola.

.catch(): Este método gestiona los errores, se ejecuta si la promesa se rechaza. Error recibe el error o la razón que fue pasada a reject cuando la promesa fue rechazada. En este caso Imprime el error a la consola. Este es un mecanismo clave para el manejo de errores en el flujo asíncrono.

.finally(): se ejecuta al final del manejo de la promesa, independientemente de si fue resuelta o rechazada. No recibe ningún argumento, y su propósito es realizar acciones de limpieza o finalización que se deben ejecutar después de que las promesas se han completado.

Ejemplo de Uso de Promesas con then, catch, y finally

```
function obtenerDatosDeAPI() {
    return new Promise((resolve, reject) => {
        // Simulando una llamada a API con setTimeout
        setTimeout(() => {
            // Supongamos que esta es una operación que puede fallar aleatoriamente
            const fallo = Math.random() > 0.5; // 50% de probabilidad de fallo
            if (fallo) {
                reject("Error: No se pudo obtener los datos.");
            } else {
                resolve("Datos obtenidos exitosamente.");
            }
        }, 1000); // Simulamos un retraso de 1 segundo
    });
}
```

Ejemplo de Uso de Promesas con then, catch, y finally

```
// Usando la función que devuelve la promesa
obtenerDatosDeAPI()
    .then(datos => {
        // Se ejecuta si la promesa se resuelve exitosamente
        console.log(datos);
    })
    .catch(error => {
        // Se ejecuta si la promesa se rechaza
        console.error(error);
    })
    .finally(() => {
        // Se ejecuta independientemente del resultado anterior
        console.log("Operación completada.");
    });
```

Material extra

Artículos de interés

Material de lectura:

- Promesas en JS: https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Promise
- Javascript.info – Promesas y async/awayt: https://es.javascript.info/promise-basics

Videos:

- ¿Cómo funcionan las Promises y Async/Await en JavaScript? : https://www.youtube.com/watch?v=rKK1q7nFt7M
- Callbacks VS Promises en JavaScript. ¡Entiende las diferencias y la importancia de cada una!: https://www.youtube.com/watch?v=frm0CHyeSbE

No te olvides de dar el presente

Recordá:

- Revisar la Cartelera de Novedades.
- Hacer tus consultas en el Foro.
- Realizar los Ejercicios obligatorios.

Todo en el Aula Virtual.

Muchas gracias por tu atención.

Nos vemos pronto