Youtic's
Lic. Gabriel Chaldú

Diferencia entre Callback y una función síncrona

La diferencia principal entre usar un callback dentro de una función y llamar a otra función síncrona dentro otra función en JavaScript es cómo manejas la ejecución de código asincrónico o dependiente de eventos.

Callback dentro de una función: Cuando utilizas un callback dentro de una función, estás pasando una función como argumento a otra función para que se ejecute en un momento específico, como en respuesta a un evento o una operación asincrónica. Por lo tanto, cuando usas un callback dentro de una función, estás permitiendo que la función principal continúe su ejecución sin interrupciones mientras se espera la finalización de una tarea asincrónica o un evento. El callback se ejecutará en un momento posterior cuando se cumplan ciertas condiciones. Esto es especialmente útil en situaciones donde no sabes cuándo ocurrirá el evento que desencadenará la acción que quieres realizar.

Por ejemplo, considera una función que realiza una solicitud AJAX para obtener datos de un servidor y luego llama a un callback cuando se completan los datos:

Ejemplo:

```
function obtenerDatosDelServidor(callback) {
    // Simulamos una solicitud AJAX asincrónica
    setTimeout( () => {
        const datos = { mensaje: 'Datos del servidor recibidos' };
        callback(datos);
    }, 2000);
}

function procesarDatos(datos) {
    console.log('Procesando datos:', datos.mensaje);
}

obtenerDatosDelServidor(procesarDatos);
console.log('Solicitud AJAX en progreso...');
```

2. **Llamada a otra función síncrona**: Si llamas directamente a otra función común dentro de una función, estás ejecutando esa función de inmediato y **bloqueando el flujo de ejecución** hasta que se complete. Esto puede ser útil cuando deseas simplemente encapsular una funcionalidad en una función separada para mejorar la legibilidad y modularidad del código.

Por ejemplo, considera una función que realiza una serie de cálculos y luego llama a una función separada para mostrar los resultados:

```
function realizarCalculos() {
    const resultado = 5 + 3;
    mostrarResultado(resultado);
    console.log('Se ejecuta ultimo')
}

function mostrarResultado(valor) {
    console.log('El resultado es:', valor);
}

realizarCalculos();
```

Los callbacks son útiles para el manejo de código asincrónico o eventos, permitiendo que una función se ejecute en respuesta a una acción específica. Llamar a otra función común simplemente encapsula una funcionalidad, y esa función se ejecuta inmediatamente dentro de la función principal. La elección entre ambos enfoques depende de la lógica y el flujo de tu programa.

Sí, en términos generales, los callbacks suelen utilizarse en situaciones asincrónicas, mientras que las funciones normales se utilizan en situaciones sincrónicas. A continuación, aclararé esta distinción:

1. Callbacks y Asincronía:

- ✓ Los callbacks son comúnmente utilizados para manejar operaciones asincrónicas, como solicitudes AJAX, lectura/escritura de archivos, temporizadores, eventos del usuario, entre otros.
- ✓ Permiten que la función principal continúe ejecutándose mientras se espera que ocurra algún evento asincrónico.
- ✓ El callback se ejecutará en un momento posterior cuando se complete la operación asincrónica o cuando se active el evento.
- ✓ Ayudan a evitar el bloqueo del hilo principal de JavaScript, lo que permite que la aplicación siga siendo receptiva durante operaciones que pueden llevar tiempo.

2. Funciones Síncronas:

- ✓ Las funciones normales se utilizan en situaciones sincrónicas, donde el código se ejecuta secuencialmente y bloquea la ejecución hasta que se complete la función.
- ✓ Son adecuadas para tareas que no involucran demoras significativas o eventos asincrónicos.
- ✓ La ejecución de una función normal no continúa hasta que la función haya terminado su trabajo.

Entonces, sí, puedes generalizar que los callbacks se utilizan en situaciones asincrónicas, mientras que las funciones normales se utilizan en situaciones sincrónicas. Sin embargo, ten en cuenta que JavaScript es un lenguaje flexible, y las líneas entre sincronía y asincronía pueden difuminarse en escenarios más complejos, especialmente con el uso de promesas, async/await y otros conceptos de programación asincrónica más modernos.

Promesas | async y await

1. Promesas (Promises):

- Las promesas en JavaScript son asincrónicas, lo que significa que permiten que el código continúe ejecutándose sin bloquearse mientras se espera que se resuelva una tarea asíncrona.

- Las promesas son no bloqueantes, lo que significa que el código no se detiene para esperar a que se resuelva una promesa. En cambio, se ejecuta una función de retorno de llamada (callback) cuando la promesa se resuelve (o se rechaza), permitiendo así que otras tareas continúen.

Ejemplo básico de Promise

```
const empleados = [
    { id: 1, nombre: 'Maria' },
    { id: 2, nombre: 'Carla' },
    { id: 3, nombre: 'Fernando' }
];

const getEmpleadoId = (id) => {
    return new Promise ((resolve, reject) => {
        const empleado = empleados.find((e) => e.id === id);
        (empleado)
            ? resolve(empleado)
            : reject(`El empleado con id: ${id} no existe`)
    })
}
```

```
getEmpleadoId2(2)
    .then((e)=> console.log(e))
    .catch((error) => console.log(error))
```

2. **async/await**:

- async y await son características de JavaScript que hacen que el código asincrónico sea más fácil de leer y entender.

- async se utiliza para marcar una función como asincrónica, lo que significa que puede contener operaciones asincrónicas dentro de ella.

- await se usa dentro de una función async para esperar a que una promesa se resuelva. Sin embargo, a diferencia de ser "bloqueante", await pausa la ejecución de la función async pero no bloquea todo el hilo de ejecución. **Otras tareas que no dependen de la resolución de esa promesa aún pueden ejecutarse.**

Entonces, para aclarar, async/await no es bloqueante en el sentido de que permite que otras tareas se ejecuten mientras se espera una operación asincrónica. Esto es útil para evitar bloqueos en la interfaz de usuario en aplicaciones web, por ejemplo. En resumen:

- Las promesas son asincrónicas y no bloqueantes.

- async/await es una forma más legible de trabajar con promesas y tampoco bloquea completamente el hilo de ejecución.

fetch()

En JavaScript, fetch es una función que se utiliza para realizar solicitudes de red (por ejemplo, solicitudes HTTP) desde un navegador web o desde un entorno de ejecución de JavaScript compatible con la especificación Fetch API. Fetch se utiliza comúnmente para recuperar datos de servidores web o para enviar datos a un servidor. Es una forma moderna de realizar solicitudes de red en comparación con las técnicas más antiguas, como XMLHttpRequest.

Aquí tienes una breve descripción de cómo se usa fetch y para qué sirve:

1. Realizar una solicitud GET:

```
const url = 'https://picsum.photos/list'


const getApi = async(url) => {

    try{
        const resultado = await fetch(url);
        const data = await resultado.json();
        console.log(data)
    }catch(error){
        console.log(`Error grave!!!: ${error}`)
    }
}

document.addEventListener('DOMContentLoaded', getApi(url))
```

2. Realizar una solicitud POST:

```
const nuevoCliente = async (cliente) => {
    try {
            await fetch(
                    url,
                {
                    method: 'POST',
                    body: JSON.stringify(cliente),
                    headers: {'Content-Type': 'application/json'}
                }
            );
            window.location.href = 'index.html';
    } catch (error) {
        console.log(error)
    }
}
```

3. Realizar otras operaciones como PUT, DELETE, etc., es similar a la solicitud POST, simplemente cambias el método en la configuración.

fetch devuelve una promesa que resuelve en un objeto Response, que contiene la respuesta del servidor. Puedes usar métodos como .json(), .text(), o .blob() en el objeto Response para obtener los datos en el formato deseado.

fetch es una función importante en JavaScript que se utiliza para realizar solicitudes de red y recuperar datos de servidores web. Es ampliamente utilizado en aplicaciones web modernas para interactuar con servicios web y API.