Lic. Gabriel Chaldu
Laboratorio 4

Callbacks

Los callbacks en JavaScript son funciones que se pasan como argumentos a otras funciones y se ejecutan después de que se complete una tarea o evento particular. En esencia, son una forma de manejar la asincronía en JavaScript y permiten controlar el flujo de ejecución de manera más flexible.

En JavaScript, muchas operaciones son asíncronas, lo que significa que no se ejecutan de manera secuencial. Por ejemplo, solicitudes a servidores, lectura/escritura de archivos y manipulación de bases de datos son operaciones que pueden llevar tiempo en completarse. En lugar de bloquear la ejecución del programa mientras se espera a que estas operaciones terminen, se utilizan callbacks para indicar qué hacer cuando la operación asíncrona se complete.

Aquí hay un ejemplo básico de cómo se podría usar un callback en JavaScript:

```
//operacion es un callback
const operaciones = (n1, n2, operacion) => {
  return operacion(n1, n2);
};
//llamada
let resultado = operaciones(
  10, //n1
  12, //n2
  ( num1, num2 ) => {//callback
    return num1 + num2;
  }
);

console.log("La suma es = ", resultado);
```

Los callbacks pueden volverse complicados y propensos a errores cuando se usan en situaciones más complejas, lo que llevó al surgimiento de otras soluciones más elegantes, como las Promesas y las funciones asíncronas (async/await), que proporcionan un mejor control y legibilidad en el manejo de la asincronía en JavaScript. Estas soluciones posteriores están diseñadas para ayudar a manejar la complejidad y los problemas potenciales asociados con los callbacks anidados.

Lic. Gabriel Chaldu
Laboratorio 4

Ejemplo de Callback

Tenemos en arreglo de empleados

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

Definimos la función

```
const getEmpleadoId = (id, callback) => {
    const empleado = empleados.find((e) => e.id === id);

    (empleado)
        ? callback(null, empleado)
        : callback (“El empleado con id: ${id} no existe”)
}
```

Llamada a la función

```
getEmpleadoId(id, (err, e) => {
    if(err){
        console.log('ERROR')
        return console.log(err)
    }
    console.log(e)
})
```

Lic. Gabriel Chaldu
Laboratorio 4

Ejemplo de Callback Hell

Agregamos el arreglo de salario de los empleados:

```
const salarios = [
    { id: 1, salario: 1000 },
    { id: 2, salario: 2000 },
]
```

Buscamos un salario por su id

```
const getSalarioId = (id, callback) => {
    const salario = salarios.find((e) => e.id === id)
    if(salario)
    {
        callback(null, salario)
    }else{
        callback (“El empleado con id: ${id} no TIENE SALARIO”);
    }
}
```

Callback hell

```
let id=10;
getEmpleadoId(id, (err, e) => {
    if(err){
        console.log('ERROR')
        return console.log(err)
    }
    getSalarioId(id, (err, s) => {
        if(err){
            console.log('ERROR')
            return console.log(err)
        }
        console.log(“El empleado/a ${e.nombre} tiene un salalario de
${s.salario}”);

    })
})
```

Lic. Gabriel Chaldu
Laboratorio 4

Callback Hell - Teoria

El "callback hell" (también conocido como "Pyramid of Doom" o "Callback Pyramid") es un término que se utiliza en la programación para describir una situación en la que tienes múltiples callbacks anidados dentro de otros callbacks, lo que resulta en un código difícil de leer y mantener. Este fenómeno generalmente ocurre en lenguajes de programación asincrónica, como JavaScript, donde las operaciones asincrónicas se manejan mediante callbacks.

Cuando se tienen muchas operaciones asincrónicas que dependen unas de otras o que deben ejecutarse en un orden específico, es común que los callbacks se aniden uno dentro de otro. Esto puede hacer que el código sea difícil de entender, seguir y depurar, ya que la estructura de anidamiento puede volverse muy profunda y confusa.

Ejemplo simplificado de "callback hell" en JavaScript:

```
const hacerAlgo = (callback1) => {
    hacerOtraCosa((resultado1) => {
        hacerMásCosas(resultado1, (resultado2) => {
            hacerAúnMás(resultado2, (resultado3) => {
                // Y así sucesivamente...
            });
        });
    });
}
```

Como puedes ver, los callbacks se anidan uno dentro de otro, y a medida que aumenta la complejidad de las operaciones asincrónicas, la estructura se vuelve aún más confusa. Esto puede dificultar la comprensión del flujo de ejecución y aumentar la posibilidad de cometer errores.

Para mitigar el callback hell, se han introducido conceptos como Promesas y async/await en JavaScript, que proporcionan una forma más legible y estructurada de manejar operaciones asincrónicas, reduciendo la necesidad de anidar callbacks de manera excesiva y mejorando la legibilidad del código.

Callback vs Funciones encapsuladas

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

1. **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. Los callbacks permiten manejar el flujo de control de manera flexible y pueden ser útiles cuando necesitas realizar acciones después de que se complete una operación.

Ejemplo:

```
function obtenerDatosDelServidor(callback) {
    // Simulamos una solicitud AJAX asincrónica
    setTimeout(() => {
        const datos = { mensaje: 'Datos del servidor recibidos' };
        callback(datos);
    }, 1000);
}

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

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

2. **Llamada a otra función común:** 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.

Ejemplo

```
function realizarCalculos() {
    const resultado = 5 + 3;
    mostrarResultado(resultado);
}

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.

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.