Callbacks, Promesas y Asincronismo en JavaScript

Repaso de los callbacks

- Como ya hemos visto, los callbacks son funciones dentro de otras funciones .
- Estas funciones llamadas callbacks se pasan como parámetro a otra función, para que esta función ejecute dicho callback.

```
function hacerAlgo(callback) {
    console.log("Haciendo algo...");
    callback();
}

function mensajeFinal() {
    console.log("¡Chau!");
}

hacerAlgo(mensajeFinal);
```

Uso de los callbacks

- Los callbacks nos ayudan a no repetir código, y nos puede ayudar a mejorar el nivel de abstracción y lectura del código.
- Suelen utilizarse para realizar acciones luego de haber ejecutado una función.
- Los callbacks son síncronos
- Esto significa que están coordinados con el tiempo.
- La sincronía se refiere a la ejecución de un solo proceso de manera simultanea. Javascript es síncrono

Asincronismo

- Supongamos que queremos ejecutar primero una función, y luego otra función.

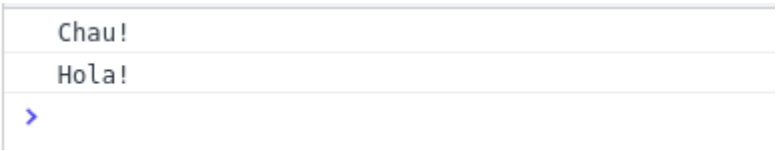
```
function uno() {
    setTimeout(function() {
        console.log("Hola!");
    }, 3000)
}

function dos() {
    console.log("Chau!");
}

uno();
dos();
```

- La primer función demorará mas tiempo que la segunda (por ejemplo, porque hará una petición a un servidor), en este caso lo simulamos con un Timer

¿En qué orden se ejecuta el código?



- Vemos que el programa no se ejecuta como yo quiero
- Primero se ejecuta dos(), y luego uno()
- Primero imprime “Chau!” y luego “Hola!”
- Nosotros queríamos que ocurriera lo contrario.

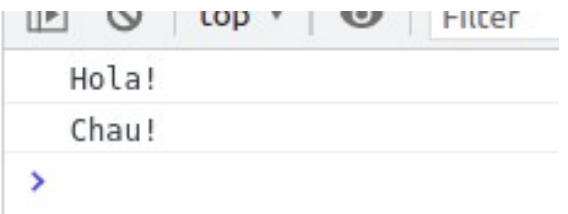
¿Cómo lo solucionamos?

- Podemos solucionar esto con callbacks de la siguiente forma:

```
function uno(callback) {
    setTimeout(function() {
        console.log("Hola!");
        callback();
    }, 3000)
}

function dos() {
    console.log("Chau!");
}

uno(dos);
```



¿Porqué ocurre esto?

- Esto ocurre porque JavaScript es asíncrono.
- Javascript usa un modelo asíncrono y no bloqueante, con un loop de eventos implementado en un sólo hilo, (single thread)
- Esto significa que JavaScript no esperará a que un proceso termine y se quedará bloqueado, si no que pasará al siguiente código, sin esperar que el código que “demora” termine su ejecución
- Vale mencionar que no todo el código que escribamos tardará en ejecutarse. En general esto ocurrirá en algunos casos particulares, como cuando hacemos una petición a una API.

¿Cómo controlamos la asincronía?

- Podemos hacerlo de tres formas:

- 1. Usando ***callbacks*** (ya lo vimos)
- 2. Usando **Promesas**
- 3. Usando **async / await**

Introducción a las Promesas

- Una promesa (Promise) es un objeto que representa el estado de una operación asíncrona.
- Tiene tres estados posibles:
  - Pendiente
  - Resuelta
  - Rechazada

- `const p = new Promise((resolve, reject) => { })`

Promesas

- Con resolve resolvemos la promesa
- Con reject rechazamos la promesa

```
function hola() {
    return new Promise((resolve)=> {
        setTimeout(() => {
            resolve(console.log("Hola"))
        }, 3000)
    })
}

function chau() {
    console.log("Chau!")
}

hola().then(() => {
    chau();
});
```

Métodos then y catch del objeto Promise

- Con then manejaremos el estado terminado (o resuelto) de nuestra promesa (el resolve)

- Con catch manejaremos el estado rechazado de nuestra promesa (el reject)

```
hola()
.then(() => {
    chau();
})
.catch(error => console.error(error))
```

Ejemplo completo
Usando promesas y then/catch

```
function hola() {
    return new Promise((resolve)=> {
        setTimeout(() => {
            resolve(console.log("Hola"))
        }, 3000)
    })
}

function chau() {
    console.log("Chau!")
}

hola()
.then(() => {
    chau();
})
.catch(error => console.error(error))
```

Otro ejemplo de Promesas

- Analicemos el siguiente código...

```
const promise1 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("Datos obtenidos del servidor. (Promesa 1)");
    }, 3000)

    setTimeout(() => {
        reject("Error: NO se pudieron obtener lo datos (Promesa 1)")
    }, 2000)

});
```

- ¿Qué se imprimirá por consola?

```
promise1
.then(data => console.log(data))
.catch(error => console.error(error))
```

Async / Await:
Funciones Asincronas

- Las promesas pueden llegar a ser muy largas a medida que se requieran más y más métodos .then().
- Las funciones asíncronas surgen para simplificar el manejo de las promesas.
- La palabra async declara una función como asíncrona e indica que una promesa será automáticamente devuelta.
- La palabra await debe ser usada siempre dentro de una función declarada como async y esperará de forma asíncrona y no bloqueante a que una promesa se resuelva o rechace.

Ejemplo de función asincrona

- Nuestro código quedaría así, usando async/await

```
function a() {
    return new Promise((resolve)=> {
        setTimeout(() => {
            resolve(console.log("Hola!"))
        }, 3000)
    })
}

function b() {
    console.log("Chau!");
}

async function ejecutar() {
    await a();
    b();
}
ejecutar()
```

Otro ejemplo de async / await

```
const promise = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("Datos obtenidos del servidor. (Promesa 1)");
    }, 3000)

    setTimeout(() => {
        reject("Error: NO se pudieron obtener lo datos (Promesa 1)")
    }, 1000)

});

async function obtenerDatos() {
    try {
        const data = await promise;
        console.log(data);
    } catch (error) {
        console.error(error);
    }
}

obtenerDatos();
```