Bienvenides a fundamentos de programacion, capitulo 7!

En esta ocasión, vamos a ver el concepto de asincronismo. Primero vamos a tratar de entender el concepto en sí, y luego vamos a ver como se aplica en Javascript.

Primero que nada: Sincronismo

Para entender que es el **asincronismo**, primero veamos que es el **sincronismo**.

En la programación, hablamos de sincronismo si las instrucciones de nuestro algoritmo se ejecutan una tras otra de forma secuencial, siempre esperando que la instrucción anterior se termine para poder empezar a ejecutar la siguiente instrucción. Basicamente lo que venimos haciendo :P

```
instruccion1;
instruccion2;
instruccion3;
```

Si consideramos que este algoritmo es sincrónico, entonces podemos asegurar que la **instrucción2** se va a ejecutar luego que se termine la **Instrucción1**, y recién cuando se termina la **Instrucción2**, se va a ejecutar la **Instrucción3**. Nada muy loco.

Ahora si: Asincronismo

Como ya se deben imaginar, hablamos de asincronismo cuando hay instrucciones en nuestro algoritmo que no necesitan esperar que termine la instrucción anterior para ejecutarse.

¿Pero para qué sirve esto?

Cocinemos unas galletitas

```
Agregar huevos;
Mezclar;
Agregar manteca;
Agregar azucar;
Mezclar;
Hornear;
```

En este caso, nos conviene que todas nuestras instrucciones sean **sincrónicas**, sino nuestra cocina sería un desastre.

Ahora cocinemos unos fideos con salsa

```
Hervir agua;
Cocinar fideos;
Calentar Salsa;
```

En este caso, podríamos ejecutar todas las instrucciones de forma **sincrónica**, pero para empezar a calentar la salsa, deberíamos esperar que los fideos se terminen de cocinar, osea no estariamos aprovechando bien el tiempo.

Podríamos entonces, empezar a calentar la salsa antes de que se terminen de cocinar los fideos. Para eso, tenemos que hacer que nuestra instrucción `Cocinar fideos;` sea **asincrónica** para que la siguiente instrucción no tenga que esperar.

```
Hervir agua;
Cocinar fideos; (instrucción asincrónica)
Calentar Salsa;
```

Hasta ahora, las instrucciones de javascript que vimos se ejecutan muy rápidamente, por ejemplo: declarar variables, modifcar valores de variables, agregar estructuras de control como if, for o while. Pero también existen operaciones que son lentas, por ejemplo: escribir datos en disco duro, hacer una consulta a una base de datos o hacer una petición HTTP. Estas operaciones que son más lentas, se ejecutan de forma asincrónica para no bloquear la ejecución de nuestro programa mientras se realizan.

Asincronísmo en Javascript

Javascript permite ejecutar código asincrónicamente a través de las **Promesas**. Para entender mejor que son las promesas veamos unos ejemplos con funciones.

```
function operacionRapida() {
    // Devuelve directamente un valor
    return 10
}
```

Como nuestra función **operacionRapida** nos devuelve directamente un valor, es sincrónica, por lo que si llamamos a la funciónde la siguiente forma:

```
const resultado = operacionRapida();
console.log(resultado);
```

Eso va a imprimir en consola el número 10, porque inmediatamente luego de ejecutar la función **operacionRapida** ya tenemos disponible el resultado.

Por otro lado, si nuestra función realiza una **operación lenta**, podemos devolver una promesa de la siguiente forma

```
function operacionLenta() {
    // Devolver una promesa que
    // EN ALGUN MOMENTO EN EL FUTURO
    // nos va a devolver un 10
    return new Promise(...) // Ya vamos a ver esto en más detalle
}
```

En este caso, si llamamos a nuestra función lenta como hicimos antes:

```
const resultado = operacionLenta();
console.log(resultado);
```

Ahora en la consola no vamos a ver un **10** porque el resultado no es un valor directamente, sino que es una PROMESA que en ALGUN MOMENTO nos va a devolver un valor. Entonces en consola vamos a ver "Promise"

¿Y qué hacemos con una Promise?

Si la función que estamos llamando nos devuelva una promesa en vez de un resultado inmediato, no podemos operar con el resultado en la siguiente instrucción, pero podemos llamar a los métodos **then** y **catch** de la promesa.

```
promesa.then(callbackSiSeCumple).catch(callbackSiNoSeCumple)
```

Si la promesa se cumple, es decir que sale todo bien, se va ejecutar la función de callback que le pasemos por parametro al método **then**. Si algo llegara a fallar en la promesa y esta al final no se cumple, se va a ejecutar la función de callback que mandemos por parametro al método **catch**.

Veamos un ejemplo

Supongamos que tenemos la función **obtenerNumeroAsync**. Debido a que es una operación asincrónica, no retorna el resultado inmediatamente sino que retorna una promesa. Sabemos también que si la promesa se cumple, nustra callback de "exito" es llamada con el resultado final como parametro, es decir el número que queremos obtener. Si la promesa falla, nuestra callback de "fallo" se llama con el error producido.

```
function callbackDeExito(resultadoFinal) {
    console.log("El número obtenido es: " + resultadoFinal)
}

function callbackDeFallo(errorOcurrido) {
    console.log("Lo sentimos, ocurrió un error: " + errorOcurrido)
}

const promesa = obtenerNumeroAsync()
promesa.then(callbackDeExito).catch(callbackDeFallo)
```

Como crear una promesa nosotros mismos

Lo más común es que nosotros ejecutemos funciones que devuelven promesas, pero a veces nosotros mismos tenemos que crear dichas funciones, por eso es bueno también saber como se crea una promesa.

```
function funcionEjecutora(resolve, reject) {
    try {
        let resultado = 10;
        resolve(resultado)
    } catch (error) {
        reject(error)
    }
}

new Promise(funcionEjecutora)
```

Como vemos, cuando hacemos `new Promise()` tenemos que enviar por parametro una "función ejecutora", la cual va a ser llamada internamente por la Promise y le va a mandar por parametro 2 funciones: **resolve** y **reject**. La función resolve la llamamos cuando nuestra promesa se cumple, y la función reject cuando no se cumple.

Otro ejemplo, la función **obtenerNumeroAsync** que usamos antes:

```
function obtenerNumeroAsync() {
    return new Promise(function(resolve, reject) {
        try {
            setTimeout(() => {
                resolve(10)
            }, 2000);
        } catch (error) {
            reject(error)
        }
    })
}
```

async / await

A veces, cuando nuestro código involucra muchas operaciones asincrónicas, se nos puede armar un lio the **.then()** anidados, lo cual da como resultado código dificil de entender y de mantener. Es por eso que javascript permite otra forma de lidiar con promesas, evitandonos el uso de los métodos **then** y **catch**, y es con el **async** y **await**. Veamos un ejemplos:

```
async function programa() {
    const resultado = await obtenerNumeroAsync()
}

programa();
```

Básicamente agregamos **await** a la instrucción que devuelve la **Promise** y eso hace que el resto de la ejecución espere a que la promesa se resuelva y devuelva un resultado. Ese resultado lo podemos asignar directamente como en el ejemplo.

Importante: Cuando usamos **await** tenemos que marcar la función en la que estamos como **async** para decirle a javascript "esta función ejecuta código asincrónico".

¿Y qué pasa con el .catch() ?

Si usamos **async / await** tenemos que usar **try / catch**

Por ejemplo:

```
async function programa() {
    try {
        const resultado = await obtenerNumeroAsync()
    } catch(error) {
        console.log("Lo sentimos, ocurrió un error: " + error);
    }
}

programa()
```

Ejercicio 1 - a

Reescribir la siguiente función tal que retorne una promesa que se cumpla luego de 3 segundos (usar setTimeout), devolviendo el mismo resultado que la función dada.

```
function dividirNumeros(dividendo, divisor) {
    return dividendo / divisor;
}
```

Ejercicio 1 - b

Usando **then** y **catch** completar la función "programa" dada para llamar a **dividirNumeros** e imprimir el resultado o imprimir un mensaje de error si ocurriera. Para provocar un error pueden enviar 0 como divisor.

```
function programa() {
    // Tu código acá
}

programa();
```

Ejercicio 1 - c

Reescribir la función **programa** del insiso anterior para que use **async** y **await**