<codoacodo/>

Agencia de
Aprendizaje
a lo largo
de la vida

Desarrollo Fullstack

BA

Les damos la bienvenida

Vamos a comenzar a grabar la clase

Clase 18

Javascript para la Web

- DOM Parte II

Clase 19

Asincronismo en Javascript

- ¿Qué es?
- Call Stack
- Event Loop
- Promesas
- Async/Await

Clase 20

Solicitando info desde Javascript

- Fetch - Axios
- Template Strings
- Local Storage

JAVASCRIPT

Asincronismo

¿Qué es el Asincronismo?

JavaScript dispone de un solo hilo de ejecución (single thread), por lo que es fácil que se **generen bloqueos**.

Cuando una operación es bloqueante, no puede ejecutarse más de una tarea al mismo tiempo o en paralelo.

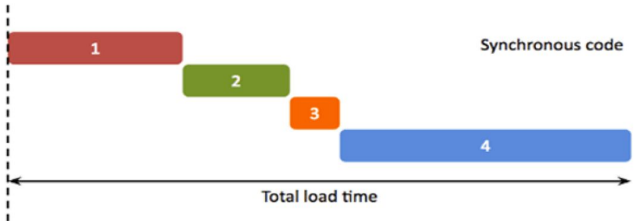
Javascript posee la capacidad de manejar procesos bloqueantes.

Ejecución Síncrona

Nuestro programa se ejecuta de forma lineal.

Si un proceso tarda en ejecutarse, nuestro programa se verá demorado hasta que ese proceso termine.

Al tener naturaleza bloqueante, cualquier proceso que no termine puede romper nuestro programa.

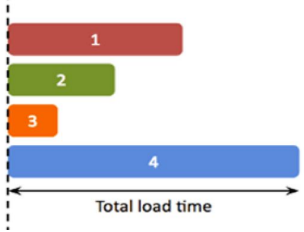


Ejecución Asíncrona

La ejecución de nuestro programa es dinámica.

Cuando definimos un proceso asíncrono, el programa sigue su camino hasta que es avisado que un proceso anterior ha finalizado.

Capturamos el resultado y lo utilizamos, sin bloquear la ejecución de nuestro código.



Async code

Asincronía en Javascript

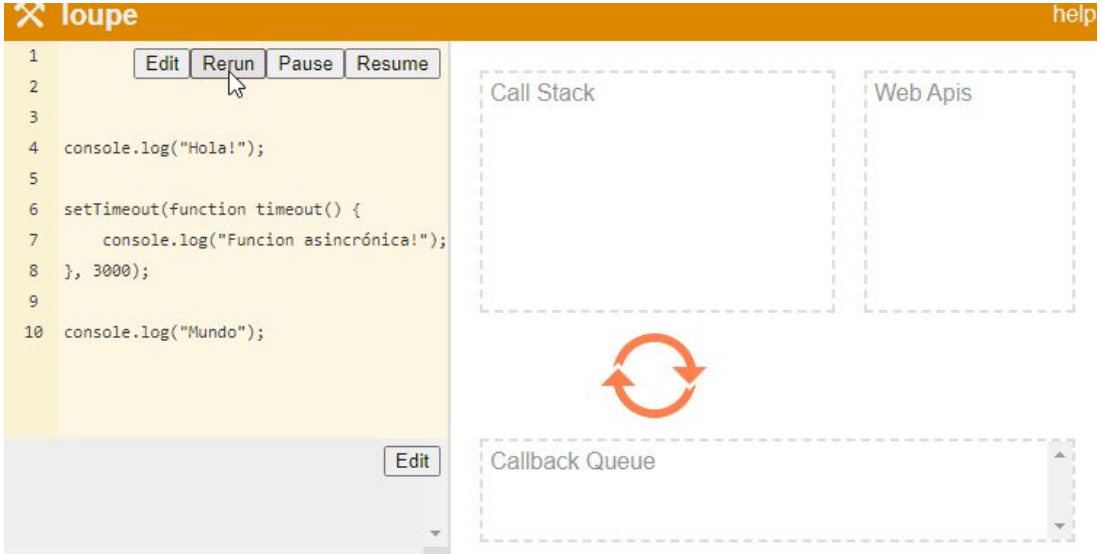
CallStack

Apila las tareas a ejecutar a medida que va leyendo el código.

Si una tarea posee otra tarea dentro estas entran en orden ascendente y se resuelven en orden descendente.

Event Loop

Recibe tareas asíncronas, las envía a la cola de tareas (callback queue) y las devuelve al callstack una vez resueltas.



Generemos asincronismo con un poco de ayuda...

setTimeOut()

Es una función nativa que nos permite generar un delay (retraso) en la ejecución de un proceso de nuestro programa.

setTimeOut()

```
setTimeout(callback, ms);
```

Recibe como primer parámetro una función a ejecutar y como segundo, el tiempo que debe transcurrir para que eso suceda expresado en milisegundos.

```
setTimeout(() => {
    console.log("Hola Mundo");
}, 5000);
```

También podemos utilizar clearTimeOut() para cancelar el retraso producido por esta función.

```
let evento = setTimeout(() => {
    console.log("Hola Mundo");
}, 5000);

clearTimeOut(evento);
```

setInterval()

Similar a la anterior, pero en este caso nos permite ejecutar una función cada determinado bloque de tiempo.

setInterval()

```
setInterval(callback, ms);
```

Recibe como primer parámetro una función a ejecutar y como segundo, cada cuanto tiempo debe repetirse su ejecución.

```
setInterval(() => {
    console.log("Hola Mundo");
}, 1000);
```

***imprime “Hola mundo” por consola cada 1 segundo**

Utilizando clearInterval() cortamos la ejecución del proceso.

```
let bucle = setInterval(() => {
    console.log("Hola Mundo");
}, 1000);

clearInterval(bucle);
```

Pasemos a lo bueno, como manejar procesos asíncronos.

Promesas

Si bien la forma primitiva de manejar procesos asíncronos en Javascript son los callbacks, hoy en día ya no se utilizan como tal, sino que en su lugar se encuentran las promesas.

Una promesa es algo que, en principio pensamos que se cumplirá, pero en el futuro pueden ocurrir varias cosas.

Estados de las Promesas

Pending

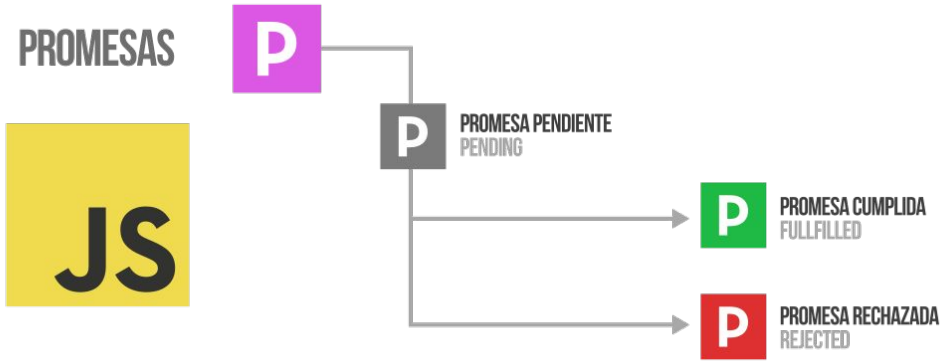
La promesa se queda en un estado incierto indefinidamente (promesa pendiente)

Fullfilled

La promesa se cumple por ende, queda resuelta y devuelve el resultado.

Rejected

La promesa no se cumple, lo que significa que fue rechazada y arrojará un error.



Creando una Promesa

Si tenemos una función donde sabemos que la respuesta de su ejecución puede no ser inmediata, necesitamos retornar una promesa.

```
function esperar() {
    return new Promise((resolve, reject) => {
        setTimeout(() => resolve("Hola Mundo"), 4000);
    });
}
```

En este caso, como nuestra función demora 4 segundos en devolver el texto, es necesario crear una promesa.

Si intentamos ejecutar nuestra función e imprimir el valor por consola, nos encontraremos con un problema.

```
console.log('1');
console.log(esperar());
console.log('3');
```

```
1
Promise { <pending> }
3
```

Esto sucede porque el console.log() intenta imprimir el resultado de esperar() de inmediato, en lugar de esperar los 4 segundos.

Manejo de Resultados

Para poder manejar el resultado de una promesa debemos hacer uso de 3 métodos especiales.

Ellos son:

.then(): que recibe como parámetro el resultado de nuestra promesa.

.catch(): que recibe como parámetro el error en caso que sea rechazada.

.finally(): recibe un callback que se ejecutará aunque la promesa falle o se resuelva.

Para tomar el resultado del ejemplo anterior usemos .then()

```
console.log('1');

esperar()
.then(res => console.log(res));

console.log('3');
```

Lo que nos da por resultado en consola:

```
1
3
Hola Mundo
```

Manejo de Errores

Cambiemos nuestro ejemplo para que la promesa resulte rechazada:

```
function esperar(condicion) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (condicion) {
                resolve("Hola Mundo");
            } else {
                reject('Ohh hubo un error');
            }
        }, 2000);
    });
}
```

Simulamos el resultado de la promesa mediante una condición.

Si la condición es verdadera:

```
esperar(true)
.then(res => console.log(res))
.catch(err => console.error(err));
```

2 segundos más tarde:

```
Hola Mundo
```

En cambio si es falsa:

```
esperar(false)
.then(res => console.log(res))
.catch(err => console.error(err));
```

```
Ohh hubo un error
```

Finalmente

Mediante el mismo ejemplo, probemos que sucede si concatenamos un .finally() a nuestra ejecución:

```
function esperar(condicion) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (condicion) {
                resolve("Hola Mundo");
            } else {
                reject('Ohh hubo un error');
            }
        }, 2000);
    });
}
```

```
esperar(true)
.then(res => console.log(res))
.catch(err => console.error(err))
.finally(() => console.log('Proceso finalizado'));
```

```
Hola Mundo
Proceso finalizado
```

Async/Await

Esta es una nueva sintaxis que nos permite trabajar con nuestras promesas evitando seguir una metodología tan estructurada.

Para eso utilizaremos las palabras reservadas async y await.

Manejo de errores con try/catch

Antes de ver **async/await** es importante conocer el bloque **try/catch** para el manejo de errores en nuestro código.

Si bien, este bloque no es exclusivo del asincronismo se utiliza muy seguido en estos casos.

Con **try/catch** le vamos a indicar a nuestro programa que debe “intentar” ejecutar el código dentro del try pero en caso de fallar debe devolver la respuesta dentro del catch.

```
function manejoError(condition) {
    try {
        if (condition) {
            console.log("Bien hecho, el código funciona!");
        } else {
            throw new Error('Algo salió mal');
        }
    } catch (err) {
        console.error(err);
    }
}

manejoError(false);
```

En este caso si condición es verdadera imprime el mensaje por consola, de lo contrario arroja un error capturado por el catch y devuelto en el console.error();

Async/Await

Bien, ahora usemos el ejemplo anterior pero con un proceso asíncrono.

```
const manejoError = async (condition) => {
    try {
        const resultado = await esperar(condition);
        console.log(resultado);
    } catch (err) {
        console.error(err);
    }
}
```

Nuestra función esperar() devuelve una promesa, por lo tanto mediante las palabras reservadas **async/await** reemplazamos el uso de los métodos **.then()** y **.catch()**.

y el finally? 🤔

```
const manejoError = async (condition) => {
    try {
        const resultado = await esperar(condition);
        console.log(resultado);
    } catch (err) {
        console.error(err);
    } finally {
        console.log('Proceso finalizado')
    }
}
```

Como vemos, también podemos agregar un finally en nuestro bloque try/catch

```
manejoError(true);
```

```
Hola Mundo
Proceso finalizado
```

Conclusión

Trabajar con promesas nos permite manipular los procesos asíncronos de nuestro código.

Siempre que tengamos un proceso que devuelva una promesa podemos utilizar las siguientes 2 variantes para manipular su resultado:

then/catch/finally

```
esperar(true)
.then(res => console.log(res))
.catch(err => console.error(err))
.finally(() => console.log('Proceso finalizado'));
```

async/await + try/catch

```
const manejoError = async (condition) => {
    try {
        const resultado = await esperar(condition);
        console.log(resultado);
    } catch (err) {
        console.error(err);
    } finally {
        console.log('Proceso finalizado')
    }
}
```

La próxima clase, veremos cómo aplicar estos conceptos a casos reales.

No te olvides de dar el presente

Recordá:

- Revisar la Cartelera de Novedades.
- Hacer tus consultas en el Foro.

Todo en el Aula Virtual.

Gracias