

Diplomatura en



FullStack Developer

JavaScript
Gestión de Errores, Promesas y Asincronía

Introducción a la Asincronía

La asincronía en programación se refiere a la ejecución de operaciones sin bloquear el flujo principal del programa. Permite que otras tareas continúen mientras una operación está en progreso.

Ejemplos de Operaciones Asíncronas:

- **Peticiones HTTP:** Solicitar datos de un servidor remoto.
- **Temporizadores:** Uso de `setTimeout` y `setInterval` para ejecutar código después de un intervalo de tiempo.
- **Eventos:** Responder a eventos de usuario como clics, movimientos del ratón o teclas presionadas.

Funciones de orden superior

Las funciones de orden superior en JavaScript son funciones que pueden tomar otras funciones como argumentos o que devuelven una función como su resultado. Este concepto es fundamental en la programación funcional y permite técnicas como la composición de funciones y el uso extensivo de funciones de callback.

La definición de funciones de orden superior es un concepto clave para profundizar en métodos avanzados de manipulación de arrays y objetos en JavaScript. Estas funciones son esenciales para entender y aplicar métodos de array como map, filter, reduce y otros, que son fundamentales para trabajar con colecciones de datos de manera expresiva y eficiente. Además, las funciones de orden superior son cruciales en la manipulación y gestión de objetos, ya que permiten operaciones tales como la transformación y la combinación de propiedades de objetos de manera funcional. Esta comprensión facilita el desarrollo de patrones de código más limpios, modulares y reutilizables.

Los métodos de array en JavaScript son herramientas esenciales para trabajar con colecciones de datos. Permiten una manipulación eficiente y expresiva de arrays, facilitando la ejecución de tareas comunes como iteración, búsqueda, filtrado, transformación y agregación de datos.

map: Este método se utiliza para crear un nuevo array con los resultados de la llamada a una función proporcionada aplicada a cada elemento del array original.

Ejemplo:

```
const numeros = [1, 2, 3, 4];  
const cuadrados = numeros.map(num => num * num);  
El array cuadrados es ahora [1, 4, 9, 16].
```

filter: Este método crea un nuevo array con todos los elementos que cumplan con la condición implementada por la función dada.

Ejemplo:

```
const numeros = [1, 2, 3, 4];  
const pares = numeros.filter(num => num % 2 === 0);  
El array pares es ahora [2, 4].
```

reduce: Este método aplica una función a un acumulador y a cada valor del array (de izquierda a derecha) para reducirlo a un único valor.

Ejemplo:

```
const numeros = [1, 2, 3, 4];  
const suma = numeros.reduce((acumulador, valorActual) => acumulador + valorActual,  
0);
```

El valor de suma es ahora 10.

forEach: Este método ejecuta una función dada una vez por cada elemento del array.

Ejemplo:

```
const numeros = [1, 2, 3, 4];  
numeros.forEach(num => console.log(num));
```

Se imprimirá 1, 2, 3 y 4 en la consola.

some: Este método comprueba si al menos un elemento del array cumple con la condición implementada por la función proporcionada.

Ejemplo:

```
const numeros = [1, 2, 3, 4];  
const tienelImpares = numeros.some(num => num % 2 !== 0);  
// tienelImpares es true porque hay números impares en el array.
```

every: Este método verifica si todos los elementos del array satisfacen la condición dada por la función proporcionada.

Ejemplo:

```
const numeros = [2, 4, 6, 8];  
const todosPares = numeros.every(num => num % 2 === 0);  
// todosPares es true porque todos los elementos son pares.
```

find: Este método devuelve el valor del primer elemento del array que cumple la función de prueba proporcionada.

Ejemplo:

```
const numeros = [3, 6, 8, 10];  
const primerParMayorQueCinco = numeros.find(num => num % 2 === 0 && num > 5);  
// primerParMayorQueCinco es 6, el primer número par mayor que 5.
```

Los objetos son colecciones de propiedades, y hay varios métodos incorporados en el lenguaje que permiten manipular estos objetos de manera efectiva. Aquí hay algunos métodos esenciales para trabajar con objetos

Object.keys(): Este método devuelve un array de las propiedades propias y enumerables de un objeto. Es útil para iterar sobre las claves de un objeto o para obtener su cantidad.

```
const objeto = { a: 1, b: 2, c: 3 };  
const claves = Object.keys(objeto);  
// claves es ["a", "b", "c"]
```

Object.values(): Similar a Object.keys(), este método devuelve un array de los valores de las propiedades enumerables de un objeto.

```
const objeto = { a: 1, b: 2, c: 3 };  
const valores = Object.values(objeto);  
// valores es [1, 2, 3]
```


Object.entries(): Este método devuelve un array de pares [clave, valor] propios de un objeto, lo que es útil para iterar sobre un objeto y tener acceso tanto a la clave como al valor.

```
const objeto = { a: 1, b: 2, c: 3 };
const entradas = Object.entries(objeto);
// entradas es [["a", 1], ["b", 2], ["c", 3]]
```

Object.assign(): Se utiliza para copiar las propiedades de uno o más objetos fuente a un objeto destino. Es útil para combinar objetos o clonar un objeto con propiedades enumerables y propias.

```
const objeto1 = { a: 1 };
const objeto2 = { b: 2 };
const combinado = Object.assign(objeto1, objeto2);
// combinado es { a: 1, b: 2 }
```

Object.freeze(): Este método congela un objeto, lo que significa que previene que nuevas propiedades sean añadidas a él, y hace todas las propiedades existentes no configurables.

```
const objeto = { a: 1 };  
Object.freeze(objeto);  
objeto.b = 2; // No tiene efecto ya que el objeto está congelado  
// objeto es { a: 1 }
```

Callbacks

Un callback es una función como cualquier otra, la diferencia está en que ésta se pasa como parámetro (argumento) para poder ser utilizado por otra función.

Permite que entonces las funciones ejecuten operaciones adicionales dentro de sí mismas.

Cuando pasamos un callback, lo hacemos porque no siempre sabemos qué queremos que se ejecute en cada caso de nuestra función.

Algunos ejemplos donde utilizamos callbacks son:

- El método `forEach`
- El método `map` o `filter`

JS callbacks1.js > ...

```
1 //Utilizaremos este arreglo de prueba
2 let valoresOriginales = [1,2,3,4,5];
3 //Estamos acostumbrados a leer una función map de la siguiente forma:
4 let nuevosValores = valoresOriginales.map(x=>x+1); //nuevosValores tendrá: [2,3,4,5,6]
5 /**
6  * Sin embargo, lo que metemos dentro de la función map es una función (flecha, más específicamente), que indica que se suma en 1 el valor
7  * del número que esté en el arreglo.
8  * ¿Siempre tenemos que sumar 1? ¡No! Nosotros podemos meter la operación que queramos, ¡y map la ejecutará de manera interna!
9  */
10 let otrosValores = valoresOriginales.map(x=>x*2); //otrosValores tendrá: [2,4,6,8,10]
11 let masValores = valoresOriginales.map(x=>"a"); //masValores tendrá: ["a","a","a","a","a"]
12 /**
13  * Notamos que, no importa cuánto cambie la función que esté metiendo dentro de map, map está hecho para RECIBIR UNA FUNCIÓN COMO PARÁMETRO
14  * y poder ejecutarla cuando lo considere pertinente. Ahora. Si estructuramos el callback por fuera.
15  */
16 const funcionCallback = (valor) => { //Función que evalúa si el valor del arreglo es un número par
17     if(valor%2===0){
18         return valor
19     }
20     else{
21         return "no es par"
22     }
23 }
24 const evaluacionDePares = valoresOriginales.map(funcionCallback); //Estoy pasando la función completa como argumento de la función map
25 console.log(evaluacionDePares) // el resultado será: ["no es par",2,"no es par",4,"no es par"];
26
```

Algunas convenciones

- El callback siempre es el último parámetro.
- El callback suele ser una función que recibe dos parámetros.
- La función llama al callback al terminar de ejecutar todas sus operaciones.
- Si la operación fue exitosa, la función llamará al callback pasando null como primer parámetro y si generó algún resultado este se pasará como segundo parámetro.
- Si la operación resultó en un error, la función llamará al callback pasando el error obtenido como primer parámetro.

Desde el lado del callback, estas funciones deberán saber cómo manejar los parámetros. Por este motivo, nos encontraremos muy a menudo con esta estructura.

```
const ejemploCallback = (error, resultado) => {  
  if (error) {  
    // hacer algo con el error!  
  } else {  
    // hacer algo con el resultado!  
  }  
};
```

Promesas

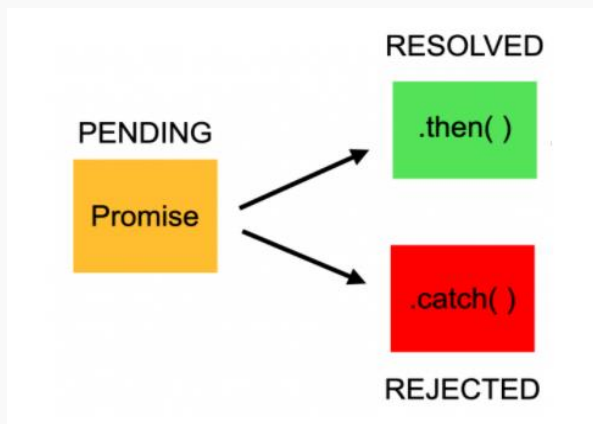
Es un objeto especial que nos permitirá encapsular una operación, la cual reaccionará a dos posibles situaciones dentro de una promesa:

- ¿Qué debería hacer si la promesa se cumple?
- ¿Qué debería hacer si la promesa no se cumple?

Al prometerse algo, es una promesa en estado pendiente (**pending**), no sabemos cuándo se resolverá esa promesa. Sin embargo, cuando llega el momento, se nos notifica si la promesa se cumplió (**Fulfilled**, también lo encontramos como Resolved) o tal vez, a pesar del tiempo, al final nos notifiquen que la promesa no pudo cumplirse, se rechazó (**Rejected**).

Ahora que entendimos que hay dos formas de resolver una promesa (**Resolved/Fulfilled o Rejected**), debemos aprender cómo utilizar estos dos estados.

- Ejecutaremos la función que acabamos de crear para que se ejecute la promesa.
- Utilizaremos el operador **.then** para recibir el caso en el que la promesa Sí se haya cumplido
- Utilizaremos el operador **.catch** para recibir el caso en el que la promesa No se haya cumplido.



setInterval y setTimeout

`setTimeout` es una función que permite ejecutar una función después de un período de tiempo específico.

`setTimeout(función, tiempo);`

- **función:** La función que se ejecutará después del tiempo especificado.
- **tiempo:** El tiempo en milisegundos después del cual se ejecutará la función.

```
setTimeout(() => {  
  console.log('Esto se ejecuta después de 2 segundos');  
}, 2000);
```

Manejo de Errores

Los errores son inevitables en la programación y deben ser gestionados adecuadamente para evitar que afecten la ejecución del programa. El manejo de errores permite capturar y responder a situaciones excepcionales.

Sintaxis para Crear un Error:

```
const error = new Error('Esto es un error personalizado');  
console.log(error.message); // Salida: Esto es un error personalizado
```

Lanzar Errores con **throw**

- La declaración **throw** se utiliza para lanzar una excepción.
- Cuando se lanza un error, la ejecución del código actual se detiene y el control se transfiere al primer bloque **catch**.

Bloques try...catch:

- `try...catch` se usa para manejar errores de manera sincrónica.
- El bloque `try` contiene el código que puede causar un error.
- El bloque `catch` maneja el error lanzado en el bloque `try`.

```
try {  
    // Código que puede lanzar un error  
    let result = riskyOperation();  
    console.log(result);  
} catch (error) {  
    // Manejo del error  
    console.error('Error capturado:', error.message);  
}
```

```
function checkNumber(number) {  
    if (number > 10) {  
        throw new Error('El número es mayor que 10');  
    }  
    return 'El número es aceptable';  
}  
  
try {  
    console.log(checkNumber(15));  
} catch (error) {  
    console.error('Error capturado:', error.message);  
}
```

async / await

Funciones Asíncronas (async):

- Una función marcada con async siempre devuelve una promesa.
- Si la función devuelve un valor, JavaScript lo envuelve en una promesa resuelta automáticamente.
- Si la función lanza una excepción, la promesa devuelta es rechazada con ese error.

La Palabra Clave await:

- await solo puede ser usada dentro de una función async.
- await pausa la ejecución de la función async y espera la resolución de la promesa.
- Cuando la promesa se resuelve, await devuelve el valor resultante.
- Si la promesa es rechazada, await lanza el error, que puede ser capturado con un bloque try...catch.

Ejemplo

```
function randomOperation() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const randomNumber = Math.floor(Math.random() * 10);
      console.log(`Número aleatorio generado: ${randomNumber}`);

      if (randomNumber < 5) {
        resolve(`Éxito: el número ${randomNumber} es menor que 5`);
      } else {
        reject(new Error(`Fallo: el número ${randomNumber} es 5 o
mayor`));
      }
    }, 2000);
  });
}

async function executeRandomOperation() {
  try {
    const message = await randomOperation();
    console.log('Resultado:', message);
  } catch (error) {
    console.error('Error capturado:', error.message);
  }
}
```

```
randomOperation()  
  .then((message) => {  
    console.log(message);  
  })  
  .catch((error) => {  
    console.error(error.message);  
  });
```

Así se vería las posibles respuestas que devuelve esta promesa

Resolved

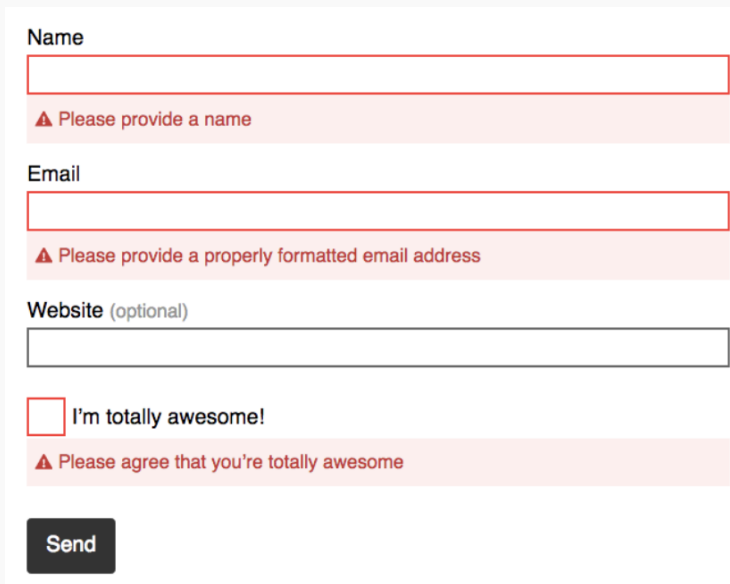
```
Número aleatorio generado: 1  
Éxito: el número 1 es menor que 5
```

Rejected




```
Número aleatorio generado: 6  
❗ ▶ Fallo: el número 6 es 5 o mayor
```

Ejercicio

Vamos a trabajar con un formulario y manejo de errores, en este ejercicio debemos devolver un texto según donde corresponda el error. Imagen de ejemplo



Formulario de ejemplo con los siguientes campos y mensajes de error:

- Name:** Campo de texto vacío. Mensaje de error:  Please provide a name
- Email:** Campo de texto vacío. Mensaje de error:  Please provide a properly formatted email address
- Website (optional):** Campo de texto vacío.
- Checkboxes:** ☐ I'm totally awesome! Mensaje de error:  Please agree that you're totally awesome
- Botón:** Send