<!--Estudio Meet-->

Callback y Promise {

<Por="Grupo de meet"/>

}

Función

Callback {

Una función callback es aquella que es pasada como argumento a otra función para que sea "llamada de nuevo" (call back) en un momento posterior.

Una función que acepta otras funciones como argumentos es llamada función de orden-superior (High-Order), y contiene la lógica para determinar cuándo se ejecuta la función callback.

```
/* *
*  Una vez entendido esto, vamos a profundizar un poco con las
*  funciones callbacks utilizadas para realizar tareas
*  asíncronas.
*  */
```

}

Formas de uso para Callback {

- Como argumento de una función.
- Definir una función anónima como callback.
- Callback con parámetros si es necesario.
- Callbacks en funciones de Array.
- Callbacks en método de temporización.
- Callbacks en Event Listener

El callback con sus formas de usar e implementar nos permite un mayor control de flujo.

}

Callbacks {

Ventajas

- Manejo de operaciones asíncronas: Permiten manejar operaciones asíncronas de manera más eficiente. Esto es especialmente útil en entornos como JavaScript, donde se realizan tareas como solicitudes HTTP, lecturas de archivos, etc.

- No bloquean el hilo principal: Al usar callbacks en operaciones asíncronas, se evita bloquear el hilo principal de ejecución, lo que permite que otras operaciones continúen mientras se espera la respuesta de una operación.

- Flexibilidad: Los callbacks permiten definir comportamientos que se ejecutarán después de que se complete una operación, lo que brinda flexibilidad en la lógica de programación.

- Reusabilidad: Los callbacks pueden ser funciones genéricas que se pueden reutilizar en diferentes contextos, lo que promueve una programación más modular y mantenible.

Desventajas

- Callback hell: En situaciones donde se anidan múltiples callbacks, puede surgir un patrón conocido como "callback hell" o "infierno de callbacks". Esto puede dificultar la comprensión y mantenimiento del código debido a la anidación excesiva de funciones.

- Dificultad para el manejo de errores: El manejo de errores en callbacks puede volverse complejo, especialmente cuando hay múltiples niveles de anidación. Puede conducir a situaciones donde los errores se propagan de manera inesperada o son difíciles de rastrear.

- Legibilidad y mantenibilidad: Un código con múltiples callbacks puede ser menos legible y más difícil de mantener, especialmente si no se maneja cuidadosamente la estructura y organización del código.

- Puede generar dependencias temporales: En algunos casos, el uso excesivo de callbacks puede generar dependencias temporales complejas, lo que dificulta la comprensión del flujo de datos y la depuración de problemas.

}

Ejemplos {

```js
const fs = require('fs');

// Función para leer un archivo
fs.readFile('archivo.txt', 'utf8', (error, datos) => {
  if (error) {
    console.error('Error al leer el archivo:', error);
    return;
  }
  console.log('Contenido del archivo:', datos);
});
```

```
/* *
*  Por cierto, tu no estás obligado a usar la palabra "callback" como el
*  nombre de tu argumento, Javascript solo necesita saber que es el nombre
*  correcto del argumento.
*  */
```

```
1  console.log("Inicio");
2
3  // setTimeout ejecutará la función de callback después de 2 segundos
4  setTimeout(() => {
5    console.log("Han pasado 2 segundos");
6  }, 2000);
7
8  console.log("Fin");
```

// ----------

```
1  const boton = document.getElementById('boton');
2
3  // Agregar un event listener que se activa cuando se hace clic en el botón
4  boton.addEventListener('click', () => {
5    console.log("¡Se hizo clic en el botón!");
6  });
7
```

}

Ejemplo callback {

```
1   function primero() {
2       console.log('primero');
3   }
4   function segundo() {
5       console.log('segundo');
6   }
7   primero();
8   segundo();
```

```
function primero() {
    setTimeout(function () {
        console.log('primero');
    }, 3000);
}
function segundo() {
    console.log('segundo');
}
primero();
segundo();
```

Promise{

Que es y sus funciones

- Las promesas (promises) son un concepto fundamental en la programación asíncrona, especialmente en JavaScript. Permiten manejar la ejecución de tareas que se completan en un futuro incierto, de forma ordenada y segura.

- Se crea utilizando el constructor Promise y recibe una función asíncrona como parámetro. Esta función, llamada "ejecutor", se ejecuta inmediatamente y tiene dos argumentos:

  1. resolve
  2. reject

}

RESOLVE

```
const myPromise = new Promise((resolve,
reject) => {
  // Simular una operación asíncrona
exitosa
  setTimeout(() => {
    resolve("Datos recuperados con
éxito");
  }, 2000);
});

myPromise.then((result) => {
  console.log("Resultado:", result);
});
```

REJECT

```
const myPromise = new Promise((resolve,
reject) => {
  // Simular una operación asíncrona
fallida
  setTimeout(() => {
    reject(new Error("Error al recuperar
datos"));
  }, 2000);
});

myPromise.catch((error) => {
  console.error("Error:", error.message);
});
```

Características

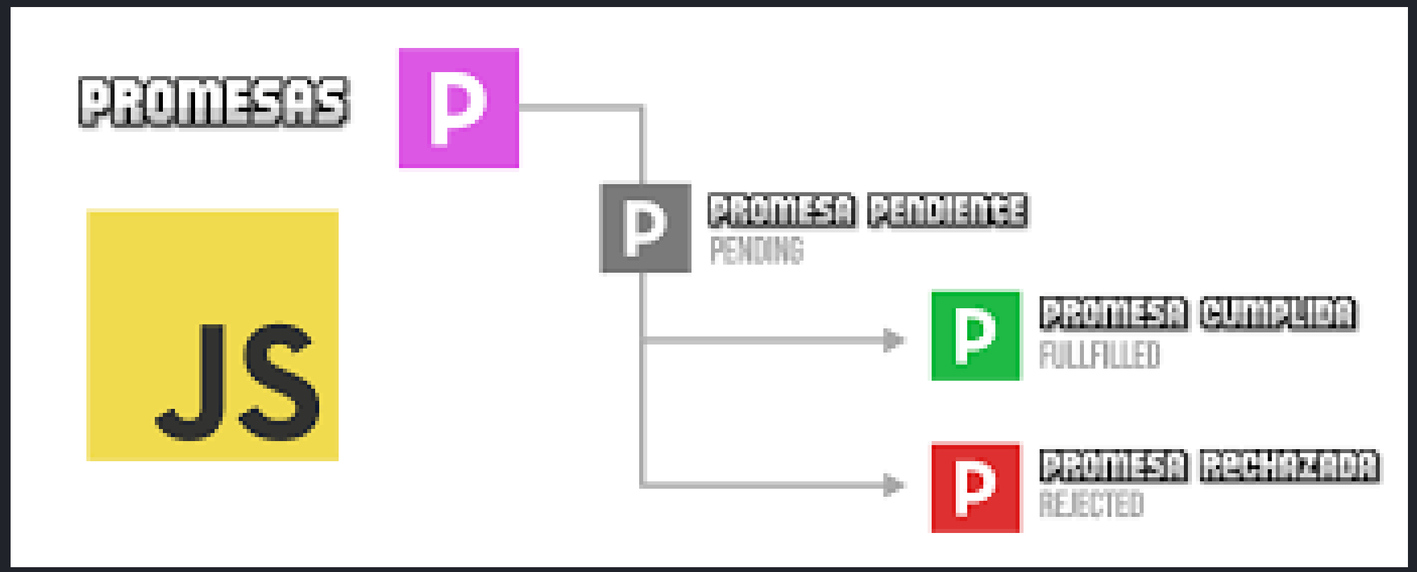
Una promesa puede estar en uno de estos tres estados:

Pending: Estado inicial, no completada ni rechazada.

Fulfilled: Significa que la operación completó con éxito.

Rejected: Significa que la operación falló.

Inmutabilidad: Una vez que una promesa se resuelve (cumplida o rechazada), su estado no puede cambiar. El valor o el motivo del rechazo también permanecen inmutables.Encadenamiento: Las promesas pueden encadenarse con .then() y .catch(), lo que permite realizar operaciones asíncronas secuenciales de manera más limpia y organizada.



Promise{

Ventajas

Manejo del flujo asíncrono óptimo
Maneja operaciones asíncronas de manera más estructurada y legible

Gestión de errores:
Se usa el método .catch() para capturar errores que ocurren en cualquier parte de la cadena de promesas.

Encadenamiento de operaciones:
El resultado de una promesa puede ser pasado como entrada a la siguiente promesa en la cadena utilizando el método .then().

Legibilidad y mantenibilidad:
Proporcionan una estructura clara para manejar resultados exitosos y errores

Manejo de paralelismo:
Se puede utilizar constructores como Promise.all() y Promise.race() para manejar múltiples promesas al mismo tiempo.

}

Promise{

Desventajas

Anidamiento complicado
El encadenamiento de muchas promesas puede llevar a ser complicado de leer. Esto puede llevar a "pirámide de la muerte" o "callback hell"

Mayor complejidad inicial:
l concepto de promesas puede resultar inicialmente confuso y difícil de entender en comparación con enfoques más simples como el uso de callbacks.

Estado no mutable:
Una vez que una promesa se resuelve o se rechaza, su estado no puede cambiar.

Falta de soporte nativo para cancelación
Las promesas no tienen soporte nativo para la cancelación, lo que puede dificultar la gestión de tareas asíncronas que necesitan ser canceladas o abortadas antes de completarse.

}

Ejemplo promise{

```
// Función que simula la búsqueda de un usuario por ID

function buscarUsuarioPorId(idUsuario) {
  return new Promise((resolve, reject) => {
    // Simular una petición asíncrona al servidor
    setTimeout(() => {
      const usuarios = [
        { id: 1, nombre: "Juan Pérez", email: "juan.perez@example.com" },
        { id: 2, nombre: "Ana Gómez", email: "ana.gomez@example.com" },
        { id: 3, nombre: "Pedro López", email: "pedro.lopez@example.com" },
      ];

      const usuarioEncontrado = usuarios.find((usuario) => usuario.id === idUsuario);

      if (usuarioEncontrado) {
        resolve(usuarioEncontrado);
      } else {
        reject(new Error("Usuario no encontrado"));
      }
    }, 2000); // Simular tiempo de espera de la petición
  });
}

// ID del usuario a buscar
const idUsuarioABuscar = 2;

// Llamada a la función de búsqueda de usuario
buscarUsuarioPorId(idUsuarioABuscar)
  .then((usuario) => {
    console.log("Usuario encontrado:");
    console.log(`Nombre: ${usuario.nombre}`);
    console.log(`Email: ${usuario.email}`);
  })
  .catch((error) => {
    console.error("Error:", error.message);
  });
```

Diferencias entre Callbacks y Promises

- **Composición:**

Callbacks: Pueden llevar a un código muy anidado y difícil de leer, conocido como "callback hell".
Promises: Permiten encadenar operaciones de manera más organizada con .then() y .catch(), haciendo el código más limpio y fácil de seguir.

- **Manejo de errores:**

Callbacks: Requieren que manejes los errores en cada callback individual, lo que puede ser repetitivo y propenso a errores.
Promises: Centralizan el manejo de errores en un solo bloque usando .catch(), simplificando la gestión de errores y mejorando la legibilidad.

- **Control de flujo:**

Callbacks: Coordinar múltiples tareas dependientes puede ser complicado y desordenado.
Promises: Ofrecen métodos como Promise.all(), que simplifican el manejo de varias operaciones asíncronas al mismo tiempo.