Taller: Realice los siguientes ejemplos, investigue temas que están en el taller, en la siguiente clase, lo subirá a un Drive para su respectiva evidencia.

Ejercicios de funciones en javascript

1. **Función para calcular el área de un círculo**:

JavaScript

```
function calcularAreaCirculo(radio) {
  return Math.PI * Math.pow(radio, 2);
}
console.log(calcularAreaCirculo(5)); // Salida: 78.53981633974483
```

2. **Función para verificar si un número es par o impar**:

JavaScript

```
function esPar(numero) {
  return numero % 2 === 0;
}
console.log(esPar(4)); // Salida: true
console.log(esPar(7)); // Salida: false
```

3. **Función para convertir una cadena a mayúsculas**:

JavaScript

```
function convertirMayusculas(cadena) {
  return cadena.toUpperCase();
}
console.log(convertirMayusculas("hola mundo")); // Salida: HOLA MUNDO
```

4. **Función para encontrar el número máximo en un array**:

JavaScript

```
function encontrarMaximo(array) {
  return Math.max(...array);
}
console.log(encontrarMaximo([1, 5, 3, 9, 2])); // Salida: 9
```

5. **Función para contar las vocales en una cadena**:

JavaScript

```
function contarVocales(cadena) {
```

```
    let contador = 0;
    const vocales = "aeiouAEIOU";
    for (let char of cadena) {
        if (vocales.includes(char)) {
            contador++;
        }
    }
    return contador;
}
console.log(contarVocales("Hola Mundo")); // Salida: 4
```

EXPLICACIÓN funciones flecha en JavaScript.

Las funciones flecha, introducidas en ECMAScript 6 (ES6), son una forma más concisa de escribir funciones en JavaScript. Aquí tienes una explicación detallada:

Sintaxis Básica

La sintaxis de una función flecha es más compacta que la de una función tradicional. Aquí tienes un ejemplo básico:

JavaScript

```
// Función tradicional
function suma(a, b) {
  return a + b;
}

// Función flecha
const suma = (a, b) => a + b;
```

Características Principales

1. **Sintaxis Concisa**:
    - Si la función tiene un solo parámetro, puedes omitir los paréntesis:

JavaScript

```
const cuadrado = x => x * x;
```

    - Si la función no tiene parámetros, debes usar paréntesis vacíos:

JavaScript

```
const saludar = () => console.log("Hola");
```

2. **Retorno Implícito**:
    - Si el cuerpo de la función es una sola expresión, el valor de esa expresión se devuelve automáticamente sin necesidad de usar la palabra clave return:

JavaScript

```
const multiplicar = (a, b) => a * b;
```

3. **No tienen su propio** this:
    - Las funciones flecha no tienen su propio contexto this. En su lugar, heredan el this del contexto en el que fueron definidas. Esto es útil para evitar problemas comunes con el this en funciones tradicionales.

JavaScript

```
function Persona() {
  this.edad = 0;


  setInterval(() => {
    this.edad++; // `this` se refiere a la instancia de Persona
    console.log(this.edad);
  }, 1000);
}


const p = new Persona();
```

4. **No tienen** arguments:
    - Las funciones flecha no tienen su propio objeto arguments. Si necesitas acceder a los argumentos, puedes usar el operador rest ...:

JavaScript

```
const sumarTodo = (...numeros) => {
  return numeros.reduce((acumulador, actual) => acumulador + actual, 0);
};
console.log(sumarTodo(1, 2, 3, 4)); // Salida: 10
```

Ejemplo Completo

Aquí tienes un ejemplo más completo que muestra varias características de las funciones flecha:

JavaScript

```
const estudiantes = [
  { nombre: 'Ana', nota: 8 },
  { nombre: 'Luis', nota: 6 },
  { nombre: 'Carlos', nota: 9 }
];
```

```
// Usando una función flecha para filtrar estudiantes con nota mayor o igual a 7
const aprobados = estudiantes.filter(estudiante => estudiante.nota >= 7);

console.log(aprobados);
// Salida: [ { nombre: 'Ana', nota: 8 }, { nombre: 'Carlos', nota: 9 } ]
```

Espero que esto te haya aclarado cómo funcionan las funciones flecha en JavaScript.

Ejercicios de funciones y alcance en javascript

Ejercicios que te ayudarán a practicar funciones y el alcance (scope) en JavaScript:

1. **Función para calcular el factorial de un número (usando recursión)**:

JavaScript

```
function factorial(n) {
  if (n === 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
console.log(factorial(5)); // Salida: 120
```

2. **Función para verificar si una cadena es un palíndromo**:

JavaScript

```
function esPalindromo(cadena) {
  const cadenaLimpia = cadena.replace(/[^a-zA-Z0-9]/g, '').toLowerCase();
  const cadenaReversa = cadenaLimpia.split('').reverse().join('');
  return cadenaLimpia === cadenaReversa;
}
console.log(esPalindromo("A man, a plan, a canal, Panama")); // Salida: true
```

3. **Función para encontrar el segundo número más grande en un array**:

JavaScript

```
function segundoMayor(array) {
  let max = -Infinity, segundoMax = -Infinity;
  for (let num of array) {
    if (num > max) {
      segundoMax = max;
      max = num;
    } else if (num > segundoMax && num < max) {
```

```
            segundoMax = num;
        }
    }
    return segundoMax;
}
console.log(segundoMayor([1, 3, 4, 5, 0, 2])); // Salida: 4
```

4. **Función para contar la frecuencia de cada carácter en una cadena**:

JavaScript

```
function contarFrecuencia(cadena) {
    const frecuencia = {};
    for (let char of cadena) {
        if (frecuencia[char]) {
            frecuencia[char]++;
        } else {
            frecuencia[char] = 1;
        }
    }
    return frecuencia;
}
console.log(contarFrecuencia("hello world"));
// Salida: { h: 1, e: 1, l: 3, o: 2, ' ': 1, w: 1, r: 1, d: 1 }
```

5. **Ejercicio sobre alcance (scope) y cierre (closure)**:

JavaScript

```
function crearContador() {
    let contador = 0;
    return function() {
        contador++;
        return contador;
    };
```

```
}


const contador1 = crearContador();

console.log(contador1()); // Salida: 1

console.log(contador1()); // Salida: 2


const contador2 = crearContador();

console.log(contador2()); // Salida: 1

console.log(contador1()); // Salida: 3
```

Estos ejercicios cubren una variedad de conceptos, desde recursión y manipulación de cadenas hasta el uso de cierres y el manejo del alcance de variables

Ejercicios para practicar el manejo de JSON en JavaScript:

1. **Convertir un objeto JavaScript a una cadena JSON**:

JavaScript

```
const persona = {
  nombre: "Juan",
  edad: 30,
  ciudad: "Madrid"
};

const jsonString = JSON.stringify(persona);
console.log(jsonString);
// Salida: {"nombre":"Juan","edad":30,"ciudad":"Madrid"}
```

2. **Convertir una cadena JSON a un objeto JavaScript**:

JavaScript

```
const jsonString = '{"nombre":"Ana","edad":25,"ciudad":"Barcelona"}';

const objeto = JSON.parse(jsonString);
console.log(objeto);
// Salida: { nombre: 'Ana', edad: 25, ciudad: 'Barcelona' }
```

3. **Acceder a datos en un objeto JSON**:

JavaScript

```
const jsonString = '{"nombre":"Carlos","edad":28,"ciudad":"Valencia"}';
const objeto = JSON.parse(jsonString);

console.log(objeto.nombre); // Salida: Carlos
console.log(objeto.edad);   // Salida: 28
console.log(objeto.ciudad); // Salida: Valencia
```

4. **Modificar datos en un objeto JSON y convertirlo de nuevo a una cadena JSON**:

JavaScript

```
const jsonString = '{"nombre":"Laura","edad":22,"ciudad":"Sevilla"}';

const objeto = JSON.parse(jsonString);


objeto.edad = 23; // Modificar la edad

const nuevoJsonString = JSON.stringify(objeto);

console.log(nuevoJsonString);

// Salida: {"nombre":"Laura","edad":23,"ciudad":"Sevilla"}
```

5. **Recorrer un array de objetos JSON**:

JavaScript

```
const jsonString =
'[{"nombre":"Pedro","edad":35},{"nombre":"Lucía","edad":29},{"nombre":"Miguel","edad":40}]';

const array = JSON.parse(jsonString);


array.forEach(persona => {

  console.log(`Nombre: ${persona.nombre}, Edad: ${persona.edad}`);

});

// Salida:

// Nombre: Pedro, Edad: 35

// Nombre: Lucía, Edad: 29

// Nombre: Miguel, Edad: 40
```

Estos ejercicios te ayudarán a familiarizarte con la conversión entre objetos JavaScript y cadenas JSON, así como a manipular y acceder a datos en formato JSON

Ejercicios avanzados con manipulación de objetos JSON.

Practicar la manipulación de objetos JSON en JavaScript:

1. **Filtrar un array de objetos JSON basado en una condición**:

JavaScript

```
const empleados = [
  { nombre: "Ana", edad: 28, departamento: "Ventas" },
  { nombre: "Luis", edad: 35, departamento: "Marketing" },
  { nombre: "Carlos", edad: 32, departamento: "Ventas" },
  { nombre: "Marta", edad: 25, departamento: "Desarrollo" }
];


const ventas = empleados.filter(empleado => empleado.departamento === "Ventas");
console.log(ventas);
// Salida: [{ nombre: "Ana", edad: 28, departamento: "Ventas" }, { nombre: "Carlos", edad: 32, departamento: "Ventas" }]
```

2. **Agrupar un array de objetos JSON por una propiedad**:

JavaScript

```
const productos = [
  { nombre: "Laptop", categoria: "Electrónica" },
  { nombre: "Camiseta", categoria: "Ropa" },
  { nombre: "Teléfono", categoria: "Electrónica" },
  { nombre: "Pantalones", categoria: "Ropa" }
];


const agrupadosPorCategoria = productos.reduce((grupo, producto) => {
  const { categoria } = producto;
  grupo[categoria] = grupo[categoria] || [];
  grupo[categoria].push(producto);
```

```
  return grupo;
}, {});

console.log(agrupadosPorCategoria);
// Salida: { Electrónica: [{ nombre: "Laptop" }, { nombre: "Teléfono" }], Ropa: [{ nombre: "Camiseta" }, { nombre: "Pantalones" }] }
```

3. **Transformar un array de objetos JSON**:

JavaScript

```
const estudiantes = [
  { nombre: "Juan", calificacion: 85 },
  { nombre: "María", calificacion: 92 },
  { nombre: "Pedro", calificacion: 78 }
];

const estudiantesConEstado = estudiantes.map(estudiante => ({
  ...estudiante,
  estado: estudiante.calificacion >= 80 ? "Aprobado" : "Reprobado"
}));

console.log(estudiantesConEstado);
// Salida: [{ nombre: "Juan", calificacion: 85, estado: "Aprobado" }, { nombre: "María", calificacion: 92, estado: "Aprobado" }, { nombre: "Pedro", calificacion: 78, estado: "Reprobado" }]
```

4. **Combinar dos arrays de objetos JSON basados en una propiedad común**:

JavaScript

```
const usuarios = [
  { id: 1, nombre: "Ana" },
  { id: 2, nombre: "Luis" },
  { id: 3, nombre: "Carlos" }
```

```
];


const pedidos = [
  { idUsuario: 1, producto: "Laptop" },
  { idUsuario: 2, producto: "Teléfono" },
  { idUsuario: 1, producto: "Camiseta" }
];


const usuariosConPedidos = usuarios.map(usuario => ({
  ...usuario,
  pedidos: pedidos.filter(pedido => pedido.idUsuario === usuario.id)
}));


console.log(usuariosConPedidos);
// Salida: [{ id: 1, nombre: "Ana", pedidos: [{ idUsuario: 1, producto: "Laptop" }, { idUsuario: 1, producto: "Camiseta" }] }, { id: 2, nombre: "Luis", pedidos: [{ idUsuario: 2, producto: "Teléfono" }] }, { id: 3, nombre: "Carlos", pedidos: [] }]
```

5. **Ordenar un array de objetos JSON por una propiedad**:

JavaScript

```
const libros = [
  { titulo: "El Quijote", autor: "Miguel de Cervantes", año: 1605 },
  { titulo: "Cien Años de Soledad", autor: "Gabriel García Márquez", año: 1967 },
  { titulo: "Don Juan Tenorio", autor: "José Zorrilla", año: 1844 }
];


const librosOrdenados = libros.sort((a, b) => a.año - b.año);
console.log(librosOrdenados);
```

// Salida: [{ titulo: "El Quijote", autor: "Miguel de Cervantes", año: 1605 }, { titulo: "Don Juan Tenorio", autor: "José Zorrilla", año: 1844 }, { titulo: "Cien Años de Soledad", autor: "Gabriel García Márquez", año: 1967 }]

Estos ejercicios te ayudarán a profundizar en la manipulación de objetos JSON, incluyendo filtrado, agrupación, transformación, combinación y ordenación

Ejercicios de Fetch API

Primero, vamos a entender qué es una API y cómo funciona.

¿Qué es una API?

Una API (Interfaz de Programación de Aplicaciones) es un conjunto de reglas y protocolos que permite a diferentes aplicaciones comunicarse entre sí. Las APIs definen métodos y datos que las aplicaciones pueden usar para interactuar con otros servicios, sistemas o aplicaciones. Por ejemplo, cuando usas una aplicación en tu teléfono para ver el clima, esa aplicación está utilizando una API para obtener los datos meteorológicos de un servidor.

¿Cómo funciona una API?

Las APIs funcionan mediante solicitudes y respuestas. Una aplicación (el cliente) envía una solicitud a otra aplicación (el servidor) a través de la API. El servidor procesa la solicitud y envía una respuesta con los datos solicitados. Este proceso generalmente se realiza a través de HTTP/HTTPS.

Fetch API en JavaScript

La Fetch API es una interfaz moderna que permite realizar solicitudes HTTP asíncronas en JavaScript. Es una alternativa más poderosa y flexible que XMLHttpRequest. La Fetch API devuelve promesas, lo que facilita el manejo de operaciones asíncronas.

Ejercicios con Fetch API

1. **Realizar una solicitud GET simple**:

JavaScript

```
// URL de la API
const url = 'https://jsonplaceholder.typicode.com/posts/1';


// Realizar la solicitud GET
fetch(url)
  .then(response => response.json()) // Convertir la respuesta a JSON
  .then(data => console.log(data))    // Mostrar los datos en la consola
  .catch(error => console.error('Error:', error)); // Manejar errores
```

Explicación: Este ejercicio realiza una solicitud GET a una API de prueba y muestra los datos obtenidos en la consola. La respuesta se convierte a JSON usando response.json().

2. **Enviar datos con una solicitud POST**:

JavaScript

```
const url = 'https://jsonplaceholder.typicode.com/posts';
const data = {
  title: 'foo',
  body: 'bar',
  userId: 1
};

fetch(url, {
  method: 'POST', // Método HTTP
  headers: {
    'Content-Type': 'application/json' // Tipo de contenido
  },
  body: JSON.stringify(data) // Convertir datos a JSON
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Error:', error));
```

Explicación: Este ejercicio envía datos a la API usando una solicitud POST. Los datos se convierten a JSON y se incluyen en el cuerpo de la solicitud.

3. **Manejar errores en una solicitud Fetch**:

JavaScript

```
const url = 'https://jsonplaceholder.typicode.com/invalid-url';

fetch(url)
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok ' + response.statusText);
    }
```

```
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Explicación: Este ejercicio muestra cómo manejar errores en una solicitud Fetch. Si la respuesta no es ok, se lanza un error que se captura en el bloque catch.

4. **Actualizar datos con una solicitud PUT**:

JavaScript

```
const url = 'https://jsonplaceholder.typicode.com/posts/1';
const data = {
  id: 1,
  title: 'foo',
  body: 'bar',
  userId: 1
};

fetch(url, {
  method: 'PUT', // Método HTTP
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(data)
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Error:', error));
```

Explicación: Este ejercicio actualiza un recurso existente usando una solicitud PUT. Los datos actualizados se envían en el cuerpo de la solicitud.

5. **Eliminar un recurso con una solicitud DELETE**:

JavaScript

```
const url = 'https://jsonplaceholder.typicode.com/posts/1';


fetch(url, {
   method: 'DELETE' // Método HTTP
})
.then(response => {
   if (response.ok) {
      console.log('Recurso eliminado');
   } else {
      console.error('Error al eliminar el recurso');
   }
})
.catch(error => console.error('Error:', error));
```

Explicación: Este ejercicio elimina un recurso usando una solicitud DELETE. Se verifica si la respuesta es ok para confirmar la eliminación.

Estos ejercicios te ayudarán a comprender cómo realizar operaciones básicas de CRUD (Crear, Leer, Actualizar, Eliminar) usando la Fetch API en JavaScript.

Ejercicios de promesas y de callbacks en javascript.

Ejercicios de Promesas en JavaScript

1. **Crear una promesa simple**:

JavaScript

```
const promesaSimple = new Promise((resolve, reject) => {
  const exito = true;
  if (exito) {
    resolve("¡Promesa cumplida!");
  } else {
    reject("Promesa rechazada.");
  }
});


promesaSimple
  .then(mensaje => console.log(mensaje))
  .catch(error => console.error(error));
```

Explicación: Aquí creamos una promesa que se resuelve si exito es true y se rechaza si es false. Usamos .then para manejar el caso de éxito y .catch para manejar el caso de error.

2. **Encadenar promesas**:

JavaScript

```
const promesa1 = new Promise((resolve) => {
  setTimeout(() => resolve("Promesa 1 cumplida"), 1000);
});


const promesa2 = new Promise((resolve) => {
  setTimeout(() => resolve("Promesa 2 cumplida"), 2000);
});
```

```
promesa1
  .then(mensaje => {
    console.log(mensaje);
    return promesa2;
  })
  .then(mensaje => console.log(mensaje))
  .catch(error => console.error(error));
```

Explicación: Este ejercicio muestra cómo encadenar promesas. La segunda promesa se ejecuta después de que la primera se resuelve.

3. **Manejo de múltiples promesas con** Promise.all:

JavaScript

```
const promesaA = Promise.resolve("Promesa A cumplida");
const promesaB = Promise.resolve("Promesa B cumplida");
const promesaC = Promise.resolve("Promesa C cumplida");

Promise.all([promesaA, promesaB, promesaC])
  .then(resultados => console.log(resultados))
  .catch(error => console.error(error));
```

Explicación: Promise.all se usa para ejecutar múltiples promesas en paralelo y esperar a que todas se resuelvan. Si alguna promesa se rechaza, Promise.all también se rechaza.

4. **Manejo de la primera promesa resuelta con** Promise.race:

JavaScript

```
const promesaX = new Promise((resolve) => setTimeout(() => resolve("Promesa X cumplida"), 1000));
const promesaY = new Promise((resolve) => setTimeout(() => resolve("Promesa Y cumplida"), 2000));

Promise.race([promesaX, promesaY])
```

```
  .then(resultado => console.log(resultado))
  .catch(error => console.error(error));
```

Explicación: Promise.race devuelve la primera promesa que se resuelve o rechaza. En este caso, promesaX se resuelve primero.

5. **Encadenar promesas con valores intermedios**:

JavaScript

```
const sumar = (a, b) => new Promise((resolve) => resolve(a + b));
const multiplicar = (a, b) => new Promise((resolve) => resolve(a * b));

sumar(2, 3)
  .then(resultado => {
    console.log(`Suma: ${resultado}`);
    return multiplicar(resultado, 2);
  })
  .then(resultado => console.log(`Multiplicación: ${resultado}`))
  .catch(error => console.error(error));
```

Explicación: Este ejercicio muestra cómo pasar valores intermedios entre promesas encadenadas. Primero sumamos dos números y luego multiplicamos el resultado.

Ejercicios de Callbacks en JavaScript

1. **Callback simple**:

JavaScript

```
function saludar(nombre, callback) {
  console.log(`Hola, ${nombre}`);
  callback();
}

function despedirse() {
  console.log("Adiós");
```

```
}

saludar("Juan", despedirse);
```

Explicación: Aquí definimos una función saludar que toma un nombre y un callback. Después de saludar, se llama al callback despedirse.

2. **Callback con parámetros**:

JavaScript

```
function operacion(a, b, callback) {
  const resultado = a + b;
  callback(resultado);
}

function mostrarResultado(resultado) {
  console.log(`El resultado es: ${resultado}`);
}

operacion(5, 3, mostrarResultado);
```

Explicación: En este ejercicio, la función operacion realiza una suma y pasa el resultado al callback mostrarResultado.

3. **Callback anidado**:

JavaScript

```
function primero(callback) {
  setTimeout(() => {
    console.log("Primero");
    callback();
  }, 1000);
}
```

```
function segundo(callback) {
  setTimeout(() => {
    console.log("Segundo");
    callback();
  }, 1000);
}

function tercero() {
  console.log("Tercero");
}

primero(() => {
  segundo(tercero);
});
```

Explicación: Este ejercicio muestra cómo anidar callbacks. primero se ejecuta, luego segundo y finalmente tercero.

4. **Manejo de errores con callbacks**:

JavaScript

```
function dividir(a, b, callback) {
  if (b === 0) {
    callback(new Error("No se puede dividir por cero"), null);
  } else {
    callback(null, a / b);
  }
}

dividir(10, 2, (error, resultado) => {
  if (error) {
```

```
    console.error(error.message);
  } else {
    console.log(`El resultado es: ${resultado}`);
  }
});
```

Explicación: Aquí manejamos errores en una operación de división. Si b es 0, se pasa un error al callback; de lo contrario, se pasa el resultado.

5. **Callback para procesar un array**:

JavaScript

```
function procesarArray(array, callback) {
  const resultado = array.map(callback);
  console.log(resultado);
}

function duplicar(numero) {
  return numero * 2;
}

procesarArray([1, 2, 3, 4], duplicar);
```

Explicación: Este ejercicio muestra cómo usar un callback para procesar cada elemento de un array. La función duplicar se aplica a cada elemento del array.

Estos ejercicios te ayudarán a comprender mejor cómo trabajar con promesas y callbacks en JavaScript.

Diferencias entre callbacks y promesas.

Callbacks

Definición: Un callback es una función que se pasa como argumento a otra función y se ejecuta después de que se completa una operación.

Ventajas:

- **Simplicidad**: Son fáciles de entender y usar para operaciones simples.
- **Compatibilidad**: Funcionan en todos los entornos de JavaScript, incluso en versiones más antiguas.

Desventajas:

- **Callback Hell**: Cuando se anidan múltiples callbacks, el código puede volverse difícil de leer y mantener. Esto se conoce como “callback hell” o “pyramid of doom”.
- **Manejo de Errores**: El manejo de errores puede ser complicado y propenso a errores, ya que cada callback debe manejar sus propios errores.

Ejemplo:

JavaScript

```
function obtenerDatos(url, callback) {
  setTimeout(() => {
    const datos = { nombre: "Juan", edad: 30 };
    callback(null, datos);
  }, 1000);
}


obtenerDatos('https://api.ejemplo.com', (error, datos) => {
  if (error) {
    console.error('Error:', error);
  } else {
    console.log('Datos:', datos);
  }
});
```

Promesas

Definición: Una promesa es un objeto que representa la eventual finalización (o falla) de una operación asíncrona y su valor resultante.

Ventajas:

- **Encadenamiento**: Permiten encadenar operaciones asíncronas de manera más limpia y legible usando .then().
- **Manejo de Errores**: Facilitan el manejo de errores, ya que los errores se pueden capturar en un solo lugar con .catch().
- **Evitan el Callback Hell**: Hacen que el código sea más plano y fácil de seguir.

Desventajas:

- **Complejidad Inicial**: Pueden ser más difíciles de entender para principiantes en comparación con los callbacks.
- **Compatibilidad**: No son compatibles con versiones muy antiguas de JavaScript sin polyfills.

Ejemplo:

JavaScript

```
function obtenerDatos(url) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const datos = { nombre: "Juan", edad: 30 };
      resolve(datos);
    }, 1000);
  });
}


obtenerDatos('https://api.ejemplo.com')
  .then(datos => {
    console.log('Datos:', datos);
  })
  .catch(error => {
    console.error('Error:', error);
```

```
});
```

Comparación

1. **Legibilidad**:
    - **Callbacks**: Pueden volverse difíciles de leer y mantener cuando se anidan múltiples callbacks.
    - **Promesas**: Ofrecen una sintaxis más limpia y manejable para encadenar operaciones asíncronas.
2. **Manejo de Errores**:
    - **Callbacks**: Cada callback debe manejar sus propios errores, lo que puede ser propenso a errores.
    - **Promesas**: Permiten manejar errores en un solo lugar con .catch().
3. **Encadenamiento**:
    - **Callbacks**: El encadenamiento de múltiples operaciones asíncronas puede llevar al "callback hell".
    - **Promesas**: Facilitan el encadenamiento de operaciones asíncronas de manera más limpia.
4. **Compatibilidad**:
    - **Callbacks**: Compatibles con todas las versiones de JavaScript.
    - **Promesas**: Necesitan polyfills para versiones muy antiguas de JavaScript.

Conclusión

Las promesas son generalmente preferibles para manejar operaciones asíncronas debido a su legibilidad y facilidad de manejo de errores. Sin embargo, los callbacks siguen siendo útiles y necesarios en ciertos contextos, especialmente cuando se trabaja con APIs más antiguas o código legado.

Entender mejor las diferencias.

Ejemplo 1: Llamadas anidadas

Con Callbacks:

JavaScript

```
function obtenerUsuario(id, callback) {
   setTimeout(() => {
      console.log('Usuario obtenido');
      callback(null, { id: id, nombre: 'Juan' });
   }, 1000);
}

function obtenerPedidos(usuarioId, callback) {
   setTimeout(() => {
      console.log('Pedidos obtenidos');
      callback(null, [{ id: 1, producto: 'Laptop' }, { id: 2, producto: 'Teléfono' }]);
   }, 1000);
}

obtenerUsuario(1, (error, usuario) => {
   if (error) {
      console.error(error);
   } else {
      obtenerPedidos(usuario.id, (error, pedidos) => {
         if (error) {
            console.error(error);
         } else {
            console.log('Pedidos:', pedidos);
         }
      });
```

```
    }
});
```

Con Promesas:

JavaScript

```
function obtenerUsuario(id) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log('Usuario obtenido');
            resolve({ id: id, nombre: 'Juan' });
        }, 1000);
    });
}

function obtenerPedidos(usuarioId) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log('Pedidos obtenidos');
            resolve([{ id: 1, producto: 'Laptop' }, { id: 2, producto: 'Teléfono' }]);
        }, 1000);
    });
}

obtenerUsuario(1)
    .then(usuario => {
        return obtenerPedidos(usuario.id);
    })
    .then(pedidos => {
        console.log('Pedidos:', pedidos);
```

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

Ejemplo 2: Manejo de errores

Con Callbacks:

JavaScript

```
function dividir(a, b, callback) {
  if (b === 0) {
    callback(new Error('No se puede dividir por cero'), null);
  } else {
    callback(null, a / b);
  }
}

dividir(10, 0, (error, resultado) => {
  if (error) {
    console.error(error.message);
  } else {
    console.log('Resultado:', resultado);
  }
});
```

Con Promesas:

JavaScript

```
function dividir(a, b) {
  return new Promise((resolve, reject) => {
    if (b === 0) {
```

```
            reject(new Error('No se puede dividir por cero'));
        } else {
            resolve(a / b);
        }
    });
}

dividir(10, 0)
    .then(resultado => {
        console.log('Resultado:', resultado);
    })
    .catch(error => {
        console.error(error.message);
    });
```

Ejemplo 3: Operaciones paralelas

Con Callbacks:

JavaScript

```
function obtenerDatos1(callback) {
    setTimeout(() => {
        callback(null, 'Datos 1');
    }, 1000);
}

function obtenerDatos2(callback) {
    setTimeout(() => {
        callback(null, 'Datos 2');
    }, 2000);
}
```

```
obtenerDatos1((error, datos1) => {
  if (error) {
    console.error(error);
  } else {
    console.log(datos1);
    obtenerDatos2((error, datos2) => {
      if (error) {
        console.error(error);
      } else {
        console.log(datos2);
      }
    });
  }
});
```

Con Promesas:

JavaScript

```
function obtenerDatos1() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve('Datos 1');
    }, 1000);
  });
}

function obtenerDatos2() {
  return new Promise((resolve) => {
    setTimeout(() => {
```

```
        resolve('Datos 2');
    }, 2000);
  });
}

Promise.all([obtenerDatos1(), obtenerDatos2()])
  .then(resultados => {
    console.log(resultados[0]); // Datos 1
    console.log(resultados[1]); // Datos 2
  })
  .catch(error => {
    console.error(error);
  });
```

Resumen de las diferencias

1. **Legibilidad**:
    - **Callbacks**: Pueden volverse difíciles de leer y mantener cuando se anidan múltiples callbacks.
    - **Promesas**: Ofrecen una sintaxis más limpia y manejable para encadenar operaciones asíncronas.
2. **Manejo de Errores**:
    - **Callbacks**: Cada callback debe manejar sus propios errores, lo que puede ser propenso a errores.
    - **Promesas**: Permiten manejar errores en un solo lugar con .catch().
3. **Encadenamiento**:
    - **Callbacks**: El encadenamiento de múltiples operaciones asíncronas puede llevar al “callback hell”.
    - **Promesas**: Facilitan el encadenamiento de operaciones asíncronas de manera más limpia.