25. Async / Await

El uso de `async` y `await` es una mejora sintáctica sobre las **promesas** que permite escribir código asincrónico de manera más clara, legible y fácil de mantener. Con `async/await`, el código parece síncrono (línea por línea), pero sigue siendo completamente asincrónico.

¿Qué son `async` y `await`?

1. `async`:
   - Declara una función como **asincrónica**.
   - Una función marcada con `async` siempre devuelve una **promesa**.
2. `await`:
   - Pausa la ejecución de una función `async` hasta que una promesa se resuelva.
   - Solo puede usarse dentro de funciones declaradas con `async`.

Sintaxis Básica

Ejemplo Simple:

```
async function obtenerDatos() {
  const datos = await descargarDatos(); // Espera a que la promesa se resuelva
  console.log(datos);
}

function descargarDatos() {
  return new Promise((resolve) => {
    setTimeout(() => resolve({ id: 1, nombre: "Ana" }), 2000);
  });
}

obtenerDatos();
```

Salida (después de 2 segundos):

```
{ id: 1, nombre: "Ana" }
```

Ventajas de `async/await`

1. **Legibilidad**:
   - Elimina la necesidad de encadenar múltiples `.then()` y hace que el código sea más lineal.
2. **Manejo de Errores**:

- Permite usar `try...catch` para manejar errores, similar al manejo de excepciones en código síncrono.

3. **Compatibilidad**:
   - Compatible con todas las promesas existentes.

Manejo de Errores con `try...catch`

Cuando usas `async/await`, puedes capturar errores usando un bloque `try...catch`.

Ejemplo:

```
async function procesarDatos() {
  try {
    const datos = await descargarDatos();
    console.log("Datos descargados:", datos);

    if (!datos.nombre) {
      throw new Error("Los datos no contienen un nombre");
    }

    console.log("Nombre:", datos.nombre);
  } catch (error) {
    console.error("Error:", error.message);
  }
}

function descargarDatos() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const error = true; // Simulamos un error
      if (error) {
        reject("Error al descargar datos");
      } else {
        resolve({ id: 1, nombre: "Ana" });
      }
    }, 2000);
  });
}

procesarDatos();
```

Salida:

```
Error: Error al descargar datos
```

Encadenamiento con `async/await`

Puedes usar `await` para esperar varias promesas en secuencia sin anidarlas.

Ejemplo:

```
async function cargarRecursos() {
  try {
    const recurso1 = await descargarRecurso("<https://api.com/recurso1>");
    console.log("Recurso 1 cargado:", recurso1);

    const recurso2 = await descargarRecurso("<https://api.com/recurso2>");
    console.log("Recurso 2 cargado:", recurso2);
  } catch (error) {
    console.error("Error al cargar recursos:", error);
  }
}

function descargarRecurso(url) {
  return new Promise((resolve) => {
    setTimeout(() => resolve(`Datos de ${url}`), 1000);
  });
}

cargarRecursos();
```

Salida:

```
Recurso 1 cargado: Datos de <https://api.com/recurso1>
Recurso 2 cargado: Datos de <https://api.com/recurso2>
```

Ejecución Paralela con `Promise.all`

Si deseas ejecutar varias promesas en paralelo, puedes combinar `Promise.all` con `await`.

Ejemplo:

```
async function cargarRecursosParalelos() {
  try {
    const [recurso1, recurso2] = await Promise.all([
      descargarRecurso("<https://api.com/recurso1>"),
      descargarRecurso("<https://api.com/recurso2>")
    ]);

    console.log("Recurso 1:", recurso1);
    console.log("Recurso 2:", recurso2);
  } catch (error) {
    console.error("Error al cargar recursos:", error);
  }
}
```

```
cargarRecursosParalelos();
```

Salida:

```
Recurso 1: Datos de <https://api.com/recurso1>
Recurso 2: Datos de <https://api.com/recurso2>
```

Comparación: Promesas vs `async/await`

Característica	Promesas	`async/await`
Legibilidad	Menos legible con `.then()`	Más legible y estructurado
Manejo de Errores	`.catch()`	`try...catch`
Encadenamiento	`.then()`	Código plano
Ejecución Paralela	`Promise.all`	Combinable con `Promise.all`

Buenas Prácticas

1. **Usa `try...catch`**:
   Siempre maneja los errores en funciones `async` para evitar interrupciones inesperadas.
2. **Evita Bloquear el Event Loop**:
   Aunque `await` pausa la función, no bloquea el hilo principal. Otras tareas pueden seguir ejecutándose mientras esperas.
3. **Combina con Promesas**:
   Usa `Promise.all` para ejecutar múltiples operaciones en paralelo cuando sea posible.

Ejemplo Completo

Escenario: Descargar y Procesar Datos

```
async function main() {
  try {
    const usuario = await descargarUsuario();
    console.log("Usuario:", usuario);

    const posts = await descargarPosts(usuario.id);
    console.log("Posts del usuario:", posts);

    const comentarios = await Promise.all(
      posts.map((post) => descargarComentarios(post.id))
    );
    console.log("Comentarios de los posts:", comentarios);
```

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

function descargarUsuario() {
  return new Promise((resolve) => {
    setTimeout(() => resolve({ id: 1, nombre: "Ana" }), 1000);
  });
}

function descargarPosts(userId) {
  return new Promise((resolve) => {
    setTimeout(() => resolve([{ id: 1, titulo: "Post 1" }, { id: 2, titulo: "Post 2" }]), 1500);
  });
}

function descargarComentarios(postId) {
  return new Promise((resolve) => {
    setTimeout(() => resolve(`Comentarios del post ${postId}`), 1000);
  });
}

main();
```

Salida:

```
Usuario: { id: 1, nombre: "Ana" }
Posts del usuario: [{ id: 1, titulo: "Post 1" }, { id: 2, titulo: "Post 2" }]
Comentarios de los posts: ["Comentarios del post 1", "Comentarios del post 2"]
```
