12. Manejo de Errores

El manejo de errores es una parte fundamental del desarrollo de software. Permite anticipar y gestionar situaciones inesperadas o excepcionales que pueden interrumpir el flujo normal de un programa. En JavaScript, las estructuras `try...catch...finally` y la palabra clave `throw` son herramientas esenciales para capturar, manejar y lanzar errores de manera controlada.

---

1. Bloque `try...catch`

El bloque `try...catch` se utiliza para probar un bloque de código en busca de errores. Si ocurre un error dentro del bloque `try`, el control pasa al bloque `catch`, donde puedes manejar el error.

Sintaxis:

```
try {
    // Código que puede generar un error
} catch (error) {
    // Código para manejar el error
}
```

- `try`: Contiene el código que podría fallar.
- `catch`: Captura y maneja el error si ocurre.

Ejemplo:

```
try {
    let resultado = 10 / 0;
    if (!isFinite(resultado)) {
        throw new Error("El resultado no es un número finito.");
    }
    console.log(resultado);
} catch (error) {
    console.log(`Error capturado: ${error.message}`);
}
// Salida: "Error capturado: El resultado no es un número finito."
```

Este ejemplo de manejo de errores en Javascript no diferencia suficientemente la parte en la que se gestiona un error procedente de la ejecución de un código y la parte en la que se genera el error debido al cálculo incorrecto de la división.

En muchas ocasiones **el programador que ha diseñado el lanzamiento del error es otro diferente del que obtiene el error lanzado**, por ejemplo si son el autor de una librería y su usuario.

1. Código de la "librería" (quien lanza errores):

```
function dividirNumeros(a, b) {
    // Validación de entrada
    if (typeof a !== 'number' || typeof b !== 'number') {
```

```
        throw new TypeError("Ambos argumentos deben ser números");
    }

    // Lógica de negocio
    if (b === 0) {
        throw new Error("División por cero no permitida");
    }

    const resultado = a / b;

    if (!isFinite(resultado)) {
        throw new Error("Resultado no es un número finito");
    }

    return resultado;
}
```

2. Código del "usuario" (quien maneja errores):

```
try {
    const resultado = dividirNumeros(10, 0);
    console.log(`Resultado: ${resultado}`);
} catch (error) {
    console.error(`Ocurrió un error: ${error.message}`);
    // Podríamos hacer más cosas como:
    // - Registrar el error
    // - Mostrar mensaje al usuario
    // - Intentar recuperación alternativa
}
```

1. **Separación de responsabilidades:**
   - La función `dividirNumeros()` (librería) se encarga de:
     - Validar entradas
     - Detectar condiciones de error
     - Lanzar errores descriptivos
   - El bloque `try/catch` (usuario) se encarga de:
     - Capturar cualquier error lanzado
     - Gestionar la situación excepcional
2. **Flujo de ejecución:**
   - Cuando el usuario llama a `dividirNumeros(10, 0)`
   - La función detecta división por cero → lanza error
   - El error salta inmediatamente al bloque `catch`
   - El código después del `throw` no se ejecuta

2. Bloque `finally`

El bloque `finally` es opcional y se ejecuta después de `try` o `catch`, sin importar si ocurrió un error o no. Es útil para realizar tareas de limpieza, como cerrar archivos o liberar recursos.

Sintaxis:

```
try {
    // Código que puede generar un error
} catch (error) {
    // Código para manejar el error
} finally {
    // Código que siempre se ejecuta
}
```

Ejemplo:

```
try {
    console.log("Intentando dividir...");
    let resultado = 10 / 0;
    console.log(resultado);
} catch (error) {
    console.log(`Error capturado: ${error.message}`);
} finally {
    console.log("Este bloque siempre se ejecuta.");
}
// Salida:
// Intentando dividir...
// Este bloque siempre se ejecuta.
```

---

3. Tipos de Errores Comunes

JavaScript proporciona varios tipos de errores predefinidos que puedes usar para describir diferentes situaciones:

- `Error`: Error genérico.
- `SyntaxError`: Error de sintaxis.
- `ReferenceError`: Acceso a una variable no definida.
- `TypeError`: Operación inválida para el tipo de dato.
- `RangeError`: Valor fuera del rango válido.
- `EvalError`: Error relacionado con la función `eval()` (poco común).

Ejemplo:

```
try {
    let numero = "cinco";
```

```
    if (isNaN(numero)) {
        throw new TypeError("El valor no es un número.");
    }
    console.log(numero * 2);
} catch (error) {
    console.log(`Error: ${error.name}: ${error.message}`);
}
// Salida: "Error: TypeError: El valor no es un número."
```

4. Creación de Errores Personalizados

Puedes crear tus propios tipos de errores extendiendo la clase `Error`. Esto es útil para manejar errores específicos de tu aplicación.

Ejemplo:

```
class MiErrorPersonalizado extends Error {
    constructor(mensaje) {
        super(mensaje);
        this.name = "MiErrorPersonalizado";
    }
}

try {
    throw new MiErrorPersonalizado("Esto es un error personalizado.");
} catch (error) {
    console.log(`${error.name}: ${error.message}`);
}
// Salida: "MiErrorPersonalizado: Esto es un error personalizado."
```

5. Buenas Prácticas para el Manejo de Errores

1. **Capturar Errores Específicos**:
   Usa bloques `try...catch` solo donde sea necesario, y maneja errores específicos en lugar de capturar todos los errores de manera genérica.
2. **Proporcionar Información Clara**:
   Asegúrate de que los mensajes de error sean claros y útiles para facilitar la depuración.
3. **Usar `finally` para Liberar Recursos**:
   Siempre libera recursos (como cerrar archivos o conexiones) en el bloque `finally`.
4. **Evitar Silenciar Errores**:
   No ignores los errores capturados sin manejarlos adecuadamente, ya que esto puede ocultar problemas importantes.