IES CLAUDIO MOYANO 1919 2019
Junta de Castilla y León
Consejería de Educación
Unión Europea
Fondo Social Europeo
"El FSE invierte en tu futuro"
GOBIERNO DE ESPAÑA
MINISTERIO DE EDUCACIÓN, CULTURA Y DEPORTE
Programa cofinanciado por el Ministerio de Educación, Cultura y Deporte y por el Fondo Social Europeo

DWEC – Javascript Web Cliente.

JavaScript 01 – Sintaxis (IV)

Manejo de errores

Si sucede un error en nuestro código el programa dejará de ejecutarse, por lo que el usuario tendrá la sensación de que no hace nada (el error sólo se muestra en la consola y el usuario no suele abrirla nunca). Para evitarlo debemos intentar capturar los posibles errores de nuestro código antes de que se produzcan.

Pero hay una construcción sintáctica **try...catch** que nos permite "atrapar" errores para que el script pueda, en lugar de morir, hacer algo más razonable.

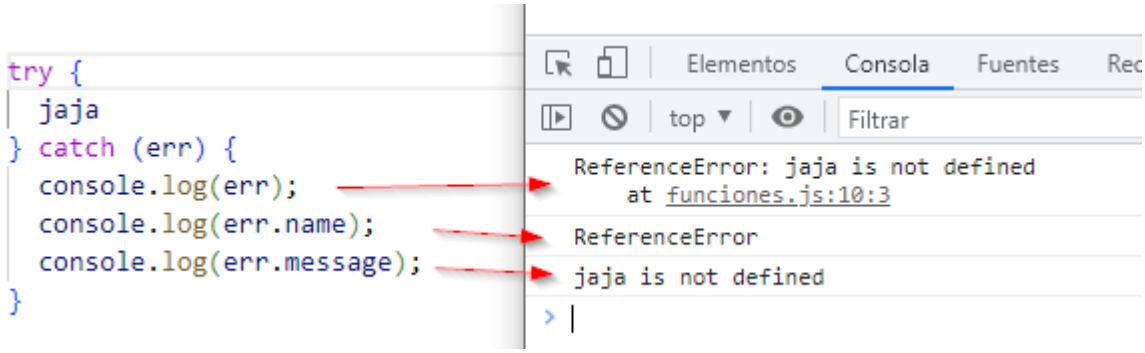
```
try {
  // código...
} catch (err) {
  // manipulación de error
}
```

Dentro del bloque *try* ponemos el código que queremos proteger y cualquier error producido en él será pasado al bloque *catch* donde es tratado. Opcionalmente podemos tener al final un bloque *finally* que se ejecuta tanto si se produce un error como si no.

Funciona así:

- Primero, se ejecuta el código en `try {...}`.
- Si no hubo errores, se ignora `catch (err)`: la ejecución llega al final de `try` y continúa, omitiendo `catch`.
- Si se produce un error, la ejecución de `try` se detiene y el control fluye al comienzo de `catch (err)`.
- La variable `err` (podemos usar cualquier nombre para ella) contendrá un objeto de error con detalles sobre lo que sucedió.

El parámetro que recibe *catch* es un objeto con las propiedades *name*, que indica el tipo de error (*SyntaxError*, *RangeError*, … o el genérico *Error*), y *message*, que indica el texto del error producido.



Para que `try..catch` funcione, el código debe ser ejecutable. En otras palabras, debería ser JavaScript válido.

No funcionará si el código es sintácticamente incorrecto, por ejemplo, si hay llaves sin cerrar.

El motor de JavaScript primero lee el código y luego lo ejecuta. Los errores que ocurren en la fase de lectura se denominan errores de "tiempo de análisis" y son irrecuperables (desde dentro de ese código). Eso es porque el motor no puede entender el código.

Entonces, `try...catch` solo puede manejar errores que ocurren en un código válido. Dichos errores se denominan "errores de tiempo de ejecución" o, a veces, "excepciones".

En ocasiones podemos querer que nuestro código genere un error. Esto evita que tengamos que comprobar si el valor devuelto por una función es el adecuado o es un código de error.

Ejemplo:

Tenemos una función para retirar dinero de una cuenta que recibe:

- el saldo de la cuenta
- y la cantidad de dinero a retirar

La función devuelve:

- el nuevo saldo, pero si no hay suficiente saldo no debería restar nada sino mostrar un mensaje al usuario.

Sin gestión de errores haríamos:

```
function retirar(saldo, cantidad) {
  if (saldo < cantidad) {
    return false;
  }
  return saldo - cantidad;
}

let saldo = 30;
cantidad = 200;
let resultado = retirar(saldo, cantidad);

if (resultado === false) {
  console.log("Saldo insuficiente");
} else {
  saldo = resultado;
}
```

Se trata de un código poco claro que podemos mejorar lanzando un error en la función. Para ello se utiliza la instrucción `throw`:

```
if (saldo < cantidad) {
    throw 'Saldo insuficiente'
}
```

Por defecto al lanzar un error, este será de clase *Error* pero (el código anterior es equivalente a `throw new Error('Valor no válido')`) aunque podemos lanzarlo de cualquier otra clase (`throw new RangeError('Saldo insuficiente')`) o personalizarlo.

Siempre que vayamos a ejecutar código que pueda generar un error debemos ponerlo dentro de un bloque *try.*

Por lo que la llamada a la función que contiene el código anterior debería estar dentro de un *try*. El código del ejemplo anterior quedaría:

```
function retirar(saldo, cantidad) {
  if (saldo < cantidad) {
    throw "Saldo insuficiente";
  }
  return saldo - cantidad;
}
// Siempre debemos llamar a esa función desde un bloque _try_

let saldo = 30;
let importe = 200;

try {
  saldo = console.log(`Nuevo saldo: ${retirar(saldo, importe)}`);
} catch (err) {
  console.log(err); // muestra "Saldo Insuficiente"
}

try {
  saldo = console.log(`Nuevo saldo: ${retirar(200, 5)}`); //Muestra "Nuevo Saldo: 195"
} catch (err) {
  console.log(err);
}
```

Podemos ver en detalle cómo funcionan en la página de MDN web docs de Mozilla.

Try…catch trabaja sincrónicamente

Si ocurre una excepción en el código “programado”, como en `setTimeout`, entonces `try..catch` no lo detectará.

Más información en: https://es.javascript.info/try-catch

EJERCICIO: Escribe una función que devuelva la nota media de un array de notas. Si el array está vacío debe avisar del error. Prueba la función con try ctach:

Buenas prácticas

Javascript nos permite hacer muchas cosas que otros lenguajes no nos dejan por lo que debemos ser cuidadosos para no cometer errores de los que no se nos va a avisar.

'use strict'

Si ponemos siempre esta sentencia al principio de nuestro código el intérprete nos avisará si usamos una variable sin declarar (muchas veces por equivocarnos al escribir su nombre). En concreto fuerza al navegador a no permitir:

- Usar una variable sin declarar
- Definir más de 1 vez una propiedad de un objeto
- Duplicar un parámetro en una función
- Usar números en octal
- Modificar una propiedad de sólo lectura

Variables

Algunas de las prácticas que deberíamos seguir respecto a las variables son:

- Elegir un buen nombre es fundamental. Evitar abreviaturas o nombres sin significado (a, b, c, …)
- Evitar en lo posible variables globales
- Usar *let* para declararlas
- Usar *const* siempre que una variable no deba cambiar su valor
- Declarar todas las variables al principio
- Inicializar las variables al declararlas
- Evitar conversiones de tipo automáticas
- Usar, para nombrarlas, la notación *camelCase*

También es conveniente, por motivos de eficiencia no usar objetos Number, String o Boolean, sino los tipos primitivos (no usar `let numero = new Number(5)`, sino `let numero = 5`) y lo mismo al crear arrays, objetos o expresiones regulares (no usar `let miArray = new Array()`, sino `let miArray = []`).

Otras

Algunas reglas más que deberíamos seguir son:

- Debemos ser coherentes a la hora de escribir código: por ejemplo, podemos poner (recomendado) o no espacios antes y después del = en una asignación, pero debemos hacerlo siempre igual. Existen muchas guías de estilo y muy buenas: Airbnb, Google, Idiomatic, etc. Para obligarnos a seguir las reglas podemos usar alguna herramienta *linter*.
- También es conveniente para mejorar la legibilidad de nuestro código separar las líneas de más de 80 caracteres.
- Usar === en las comparaciones
- Si un parámetro puede faltar al llamar a una función, darle un valor por defecto
- Y para acabar: **comentar el código** cuando sea necesario, pero mejor que el código sea lo suficientemente claro como para no necesitar comentarios

Clean Code

Estas y otras muchas recomendaciones se recogen en el libro Clean Code de *Robert C. Martin* y en muchos otros libros y articulos. Aquí se puede ver un pequeño resumen traducido al castellano:
https://github.com/devictoribero/clean-code-javascript