INTRODUCCION A ERRORES HALDING EN JAVASCRIPT

2º DAW - Desarrollo Web en Entorno Cliente

Manejo de errores, "try...catch"

- No importa lo buenos que seamos en la programación, a veces nuestros scripts tienen errores. Pueden ocurrir debido a nuestros descuidos, una entrada inesperada del usuario, una respuesta errónea del servidor y por otras razones más.
- Por lo general, un script “muere” (se detiene inmediatamente) en caso de error, imprimiéndolo en la consola. 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.

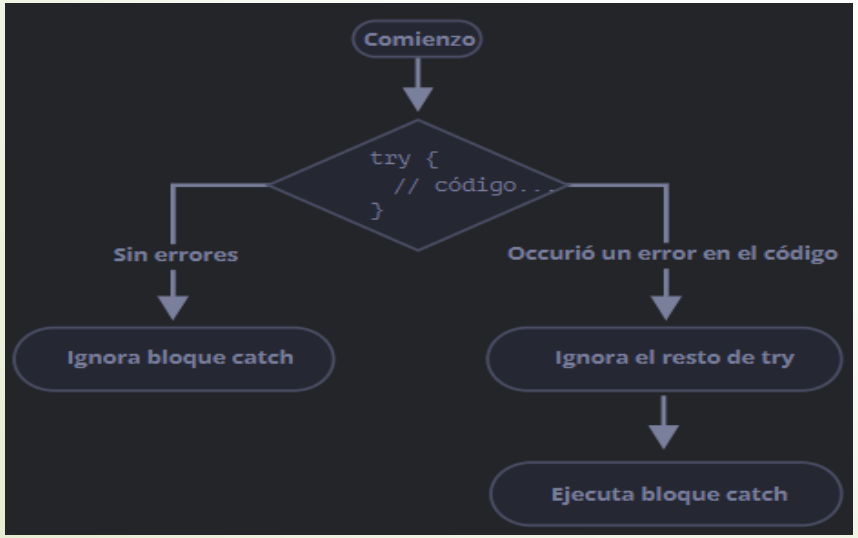
La sintaxis "try...catch"

La construcción try...catch tiene dos bloques principales: try, y luego catch:

```
try {

  // código...

} catch (err) {

  // manipulación de error

}
```

Funciona así:

1. Primero, se ejecuta el código en try {...}.
2. Si no hubo errores, se ignora catch (err): la ejecución llega al final de try y continúa, omitiendo catch.
3. 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.



Entonces, un error dentro del bloque try {...} no mata el script; tenemos la oportunidad de manejarlo en catch. Veamos algunos ejemplos.

Un ejemplo sin errores: muestra alert (1) y (2):

```
try {

  alert('Inicio de intentos de prueba');  // (1) <--

  // ...no hay errores aquí

  alert('Fin de las ejecuciones de try');   // (2) <--

} catch (err) {

  alert('Se ignora catch porque no hay errores'); // (3)

}
```

Un ejemplo con un error: muestra (1) y (3):

```
try {

  alert('Inicio de ejecuciones try');  // (1) <--

  lalala; // error, variable no está definida!

  alert('Fin de try (nunca alcanzado)');  // (2)

} catch (err) {

  alert(`¡Un error ha ocurrido!`); // (3) <--

}
```

Nota: 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

Objeto Error

Cuando se produce un error, JavaScript genera un objeto que contiene los detalles al respecto. El objeto se pasa como argumento para catch:

```
try {
  // ...
} catch(err) { // <-- el "objeto error", podría usar otra palabra en lugar de err
  // ...
}
```

Para todos los errores integrados, el objeto error tiene dos propiedades principales:

- Name: Nombre de error. Por ejemplo, para una variable indefinida que es "ReferenceError".
- Message: Mensaje de texto sobre detalles del error.

Hay otras propiedades no estándar disponibles en la mayoría de los entornos. Uno de los más utilizados y compatibles es:

- Stack: Pila de llamadas actual: una cadena con información sobre la secuencia de llamadas anidadas que condujeron al error. Utilizado para fines de depuración.

Por ejemplo:

```
try {
  lalala; // error, la variable no está definida!
} catch (err) {
  alert(err.name); // ReferenceError
  alert(err.message); // lalala no está definida!
  alert(err.stack); // ReferenceError: lalala no está definida en (...call stack)

  // También puede mostrar un error como un todo
  // El error se convierte en cadena como "nombre: mensaje"
alert(err); // ReferenceError: lalala no está definido
}
```

Si no necesitamos detalles del error, catch puede omitirlo:

```
try {
  // ...
} catch { // <-- sin (err)
  // ...
}
```

Usando "try...catch"

Exploremos un caso de uso de la vida real de try...catch. Como ya sabemos, JavaScript admite el método JSON.parse(str) para leer valores codificados con JSON.

Por lo general, se utiliza para decodificar datos recibidos a través de la red, desde el servidor u otra fuente. Lo recibimos y llamamos a JSON.parse así:

```
let json = '{"name":"John", "age": 30}'; // datos del servidor

let user = JSON.parse(json); // convierte la representación de texto a objeto JS

// ahora user es un objeto con propiedades de la cadena
alert( user.name ); // John
alert( user.age );  // 30
```

Puede encontrar información más detallada sobre JSON en el capítulo Métodos JSON, toJSON.

Si json está mal formado, JSON.parse genera un error, por lo que el script "muere".

¿Deberíamos estar satisfechos con eso? ¡Por supuesto no!

De esta manera, si algo anda mal con los datos, el visitante nunca lo sabrá (a menos que abra la consola del desarrollador). Y a la gente realmente no le gusta cuando algo "simplemente muere" sin ningún mensaje de error. Usemos try...catch para manejar el error:

```
let json = "{ json malo }";

try {

  let user = JSON.parse(json); // <-- cuando ocurre un error ...
  alert( user.name ); // no funciona

} catch (err) {
  // ...la ejecución salta aquí
  alert( "Nuestras disculpas, los datos tienen errores, intentaremos solicitarlos una
  alert( err.name );
  alert( err.message );
}
```

Aquí usamos el bloque catch solo para mostrar el mensaje, pero podemos hacer mucho más: enviar una nueva solicitud de red, sugerir una alternativa al visitante, enviar información sobre el error a una instalación de registro, ... Todo mucho mejor que solo morir.

Lanzando nuestros propios errores

¿Qué sucede si json es sintácticamente correcto, pero no tiene una propiedad requerida de name? Como este:

```
let json = '{ "age": 30 }'; // dato incompleto

try {

  let user = JSON.parse(json); // <-- sin errores
  alert( user.name ); // sin nombre!

} catch (err) {
  alert( "no se ejecuta" );
}
```

Aquí JSON.parse se ejecuta normalmente, pero la ausencia de name es en realidad un error nuestro. Para unificar el manejo de errores, usaremos el operador throw.

El operador "throw"

El operador throw genera un error. La sintaxis es:

```
throw <error object>
```

Técnicamente, podemos usar cualquier cosa como un objeto error. Eso puede ser incluso un primitivo, como un número o una cadena, pero es mejor usar objetos, preferiblemente con propiedades name y message (para mantenerse algo compatible con los errores incorporados).

JavaScript tiene muchos constructores integrados para manejar errores estándar: Error, SyntaxError, ReferenceError, TypeError y otros. Podemos usarlos para crear objetos de error también. Su sintaxis es:

```
let error = new Error(message);
// or
let error = new SyntaxError(message);
let error = new ReferenceError(message);
// ...
```

Para errores incorporados (no para cualquier objeto, solo para errores), la propiedad name es exactamente el nombre del constructor. Y mensaje se toma del argumento. Por ejemplo:

```
let error = new Error("Estas cosas pasan... o_O");

alert(error.name); // Error
alert(error.message); // Estas cosas pasan... o_O
```

Veamos qué tipo de error genera JSON.parse:

```
try {
  JSON.parse("{ json malo o_0 }");
} catch (err) {
  alert(err.name); // SyntaxError
  alert(err.message); // Token b inesperado en JSON en la posición 2
}
```

Como podemos ver, ese es un SyntaxError. Y en nuestro caso, la ausencia de name es un error, ya que los usuarios deben tener un name. Así que vamos a lanzarlo:

```
let json = '{ "age": 30 }'; // dato incompleto

try {

  let user = JSON.parse(json); // <-- sin errores

  if (!user.name) {
    throw new SyntaxError("dato incompleto: sin nombre"); // (*)
  }

  alert( user.name );

} catch (err) {
  alert( "Error en JSON: " + err.message ); // Error en JSON: dato incompleto: sin nom
}
```

En la línea (*), el operador throw genera un SyntaxError con el message dado, de la misma manera que JavaScript lo generaría él mismo. La ejecución de try se detiene inmediatamente y el flujo de control salta a catch.

Ahora catch se convirtió en un lugar único para todo el manejo de errores: tanto para JSON.parse como para otros casos.

Relanzando (rethrowing)

En el ejemplo anterior usamos try...catch para manejar datos incorrectos. Pero, ¿es posible que ocurra otro error inesperado dentro del bloque try{...}? Como un error de programación (la variable no está definida) o algo más, no solo “datos incorrectos”. Por ejemplo:

```
let json = '{ "age": 30 }'; // dato incompleto

try {
  user = JSON.parse(json); // <-- olvidé poner "let" antes del usuario

  // ...
} catch (err) {
  alert("Error en JSON: " + err); // Error en JSON: ReferenceError: user no está defi
  // (no es error JSON)
}
```

¡Por supuesto, todo es posible! Los programadores cometen errores. Incluso en las utilidades de código abierto utilizadas por millones durante décadas, de repente se puede descubrir un error que conduce a hacks terribles.

En nuestro caso, try...catch está destinado a detectar errores de “datos incorrectos”. Pero por su naturaleza, catch obtiene todos los errores de try. Aquí recibe un error inesperado, pero aún muestra el mismo mensaje de “Error en JSON”. Eso está mal y también hace que el código sea más difícil de depurar.

Para evitar tales problemas, podemos emplear la técnica de “rethrowing”. La regla es simple:

Catch solo debe procesar los errores que conoce y “volver a lanzar” (rethrow) a todos los demás.

La técnica de “rethrowing” puede explicarse con más detalle:

- Catch captura todos los errores.
- En el bloque catch (err) {...} analizamos el objeto error err.
- Si no sabemos cómo manejarlo, hacemos 'throw err`.

Por lo general, podemos verificar el tipo de error usando el operador instanceof:

```
try {
  user = { /*...*/ };
} catch (err) {
  if (err instanceof ReferenceError) {
    alert('ReferenceError'); // "ReferenceError" para acceder a una variable indefinid
  }
}
```

También podemos obtener el nombre de la clase error con la propiedad err.name. Todos los errores nativos lo tienen. Otra opción es leer err.constructor.name. En el siguiente código, usamos el rethrowing para que catch solo maneje SyntaxError:

```
let json = '{ "age": 30 }'; // dato incompleto
try {

  let user = JSON.parse(json);

  if (!user.name) {
    throw new SyntaxError("dato incompleto: sin nombre");
  }

  blabla(); // error inesperado

  alert( user.name );

} catch (err) {

  if (err instanceof SyntaxError) {
    alert( "Error en JSON: " + err.message );
  } else {
    throw err; // rethrow (*)
  }

}
```

El error lanzado en la línea (*) desde el interior del bloque catch cae desde try...catch y puede ser atrapado por una construcción externa try...catch (si existe), o mata al script.

Por lo tanto, el bloque catch en realidad maneja solo los errores con los que sabe cómo lidiar y “omite” todos los demás. El siguiente ejemplo demuestra cómo dichos errores pueden ser detectados por un nivel más de try...catch:

try…catch…finally

Espera, eso no es todo. La construcción try...catch puede tener una cláusula de código más: finally.

Si existe, se ejecuta en todos los casos:

- después de try, si no hubo errores,
- después de catch, si hubo errores.

La sintaxis extendida se ve así:

```
try {
   ... intenta ejecutar el código ...
} catch (err) {
   ... manejar errores ...
} finally {
   ... ejecutar siempre ...
}
```

Intenta ejecutar este código:

```
try {
  alert( 'intenta (try)' );
  if (confirm('¿Cometer un error?')) BAD_CODE();
} catch (err) {
  alert( 'atrapa (catch)' );
} finally {
  alert( 'finalmente (finally)' );
}
```

El código tiene dos formas de ejecución:

- Si responde “Sí” a “¿Cometer un error?”, Entonces try -> catch -> finally.
- Si dice “No”, entonces try -> finally.

La cláusula finally a menudo se usa cuando comenzamos a hacer algo y queremos finalizarlo en cualquier resultado. Por ejemplo, queremos medir el tiempo que tarda una función de números de Fibonacci fib(n). Naturalmente, podemos comenzar a medir antes de que se ejecute y terminar después. ¿Pero qué pasa si hay un error durante la llamada a la función? En particular, la implementación de fib(n) en el código siguiente devuelve un error para números negativos o no enteros.

La cláusula finally es un excelente lugar para terminar las mediciones, pase lo que pase. Aquí finally garantiza que el tiempo se medirá correctamente en ambas situaciones, en caso de una ejecución exitosa de fib y en caso de error:

```
let num = +prompt("Ingrese un número entero positivo?", 35)

let diff, result;

function fib(n) {
  if (n < 0 || Math.trunc(n) != n) {
    throw new Error("Debe ser un número positivo y entero.");
  }
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

let start = Date.now();

try {
  result = fib(num);
} catch (err) {
  result = 0;
} finally {
  diff = Date.now() - start;
}

alert(result || "error ocurrido");

alert( `la ejecución tomó ${diff}ms` );
```

Puede verificar ejecutando el código e ingresando 35 en prompt; se ejecuta normalmente, finally después de try. Y luego ingrese -1 – habrá un error inmediato, y la ejecución tomará 0ms. Ambas mediciones se realizan correctamente.

Captura global

Imaginemos que tenemos un error fatal fuera de try...catch, y el script murió. Como un error de programación o alguna otra cosa terrible.

¿Hay alguna manera de reaccionar ante tales ocurrencias? Es posible que queramos registrar el error, mostrarle algo al usuario (normalmente no ve mensajes de error), etc.

No hay ninguna en la especificación, pero los entornos generalmente lo proporcionan, porque es realmente útil. Por ejemplo, Node.js tiene process.on("uncaughtException") para eso. Y en el navegador podemos asignar una función a la propiedad especial window.onerror, que se ejecutará en caso de un error no detectado. La sintaxis:

```
window.onerror = function(message, url, line, col, error) {
  // ...
};
```

- Message: Mensaje de error.
- Url: URL del script donde ocurrió el error.
- Line, col: Números de línea y columna donde ocurrió el error.
- Error: El objeto error.

Por ejemplo:

```
<script>
  window.onerror = function(message, url, line, col, error) {
    alert(`${message}\n At ${line}:${col} of ${url}`);
  };

  function readData() {
    badFunc(); // ¡Vaya, algo salió mal!
  }

  readData();
</script>
```

El rol del controlador global window.onerror generalmente no es recuperar la ejecución del script, probablemente sea imposible en caso de errores de programación, pero sí enviar el mensaje de error a los desarrolladores.

También hay servicios web que proporcionan registro de errores para tales casos, como https://errorception.com o https://www.muscula.com. Estos servicios funcionan así:

1. Nos registramos en el servicio y obtenemos un fragmento de JS (o la URL de un script) para insertar en las páginas.
2. Ese script JS establece una función personalizada window.onerror.
3. Cuando se produce un error, se envía una solicitud de red al servicio.
4. Podemos iniciar sesión en la interfaz web del servicio y ver los errores registrados.

RESUMEN

- La construcción try...catch permite manejar errores de tiempo de ejecución. Literalmente permite "intentar (try)" ejecutar el código y "atrapar (catch)" errores que pueden ocurrir en él. La sintaxis es:

```
try {
  // ejecuta este código
} catch (err) {
  // si ocurrió un error, entonces salta aquí
  // err es el objeto error
} finally {
  // hacer en cualquier caso después de try/catch
}
```

- Puede que no haya una sección catch o finally, por lo que las construcciones más cortas try...catch y try...finally también son válidas.
- Los objetos Error tienen las siguientes propiedades:
  - message – el mensaje de error legible por humanos.
  - name – la cadena con el nombre del error (nombre del constructor de error).
  - stack (No estándar, pero bien soportado) – la pila en el momento de la creación del error.
- Si no se necesita un objeto error, podemos omitirlo usando catch { en lugar de catch (err) {.
- También podemos generar nuestros propios errores utilizando el operador throw. Técnicamente, el argumento de throw puede ser cualquier cosa, pero generalmente es un objeto error heredado de la clase incorporada Error. Más sobre la extensión de errores en el próximo capítulo.
- *Relanzado (rethrowing)* es un patrón muy importante de manejo de errores: un bloque catch generalmente espera y sabe cómo manejar el tipo de error en particular, por lo que debería relanzar errores que no conoce.
- Incluso si no tenemos try...catch, la mayoría de los entornos nos permiten configurar un controlador de errores "global" para detectar los errores que caigan. En el navegador, eso es window.onerror.

TAREAS

Finally o solo el código?

Compara los dos fragmentos de código.

El primero usa `finally` para ejecutar el código después de `try..catch`:

```
try {
  trabajo trabajo
} catch (err) {
  maneja errores
} finally {
  limpiar el espacio de trabajo
}
```

El segundo fragmento coloca la limpieza justo después de `try..catch`:

```
try {
  trabajo trabajo
} catch (err) {
  manejo de errores
}

limpiar el espacio de trabajo
```

Definitivamente necesitamos la limpieza después del trabajo, no importa si hubo un error o no.

¿Hay alguna ventaja aquí en usar `finally` o ambos fragmentos de código son iguales? Si existe tal ventaja, entonces da un ejemplo cuando sea importante.