Modulo 6. Errores, Excepciones, Depuración y Solución de Problemas

Errores y excepciones en javascript

Cuando JavaScript detecta errores sintácticos o semánticos, genera y ***arroja*** objetos específicos que contienen información sobre el error encontrado. Por lo general, en tal situación, decimos que se ha producido un error. En el caso de errores de sintaxis, el motor de JavaScript no nos permite ejecutar el programa, y en la consola recibimos información sobre lo que es incorrecto. Los errores que no sean de sintaxis (por ejemplo, errores semánticos) generalmente se denominan ***errores de tiempo de ejecución*** en JavaScript. Aparecen mientras se ejecuta el programa. También podemos llamarlas ***excepciones***. De forma predeterminada, las *excepciones* lanzadas interrumpen la ejecución del programa y hacen que aparezca la información adecuada en la consola.

Es posible evitar que el programa se detenga en tal situación. Esto se llama manejo de excepciones (o más generalmente, manejo de errores). Para manejar excepciones generadas en JavaScript (como en muchos otros lenguajes) usamos la instrucción `try ... catch`.

```
try {
    console.log('abc'); // -> abc
    conole.log('abc');
} catch (error) {
    console.log(error.message); // -> conole is not defined
}
```

Si se lanza una excepción en el bloque de código después de la palabra clave try, el programa no se interrumpe por completo, sino que salta a la parte del código después de la palabra clave catch y continúa desde allí.

Los programas no se ejecutan en el vacío. Por lo general, durante su ejecución, hay interacciones con los usuarios (p. ej., ingresar datos necesarios para calcular ciertos valores) u otros programas o sistemas (p. ej., descargar datos del servidor).

Lo correcto es adelantarse a futuros errores. Pensar de que forma podria fallar el programa y sobre ello, cerrar esas brechas de futuros fallos.

Syntax error

Un `SyntaxError` aparece cuando un código está mal formado, cuando hay errores tipográficos en las palabras clave, paréntesis o corchetes que no coinciden, etc. El código ni siquiera se puede ejecutar, ya que JavaScript no es capaz de entenderlo. Por lo tanto, se lanza el error correspondiente antes de que se inicie el programa.

```
"use strict";
iff (true) { //-> Uncaught SyntaxError: Unexpected token '{'
    console.log("true");
}
```

En el ejemplo anterior, cometimos un error tipográfico en la palabra clave `if` y agregamos una letra f adicional. El motor de JavaScript trata el nombre desconocido como una llamada de función (por los paréntesis después del iff) y se sorprende por la presencia de una llave.

Reference error

Ocurre cuando intentamos acceder a una función o variable que no existe. El motor de JavaScript no conoce el significado del nombre dado, por lo que es un error que calificaremos como un error semántico. La excepción correspondiente se lanza en el momento de la ejecución del programa, cuando se alcanza la instrucción incorrecta (es decir, en JavaScript, los errores semánticos son errores de tiempo de ejecución).

```
let a = b; // -> Uncaught ReferenceError: b is not defined
```

Type error

Este tipo de error se produce cuando un determinado valor no es del tipo esperado (es decir, intenta realizar una operación que no es aceptable). Los ejemplos típicos son cambiar el valor constante o verificar la longitud de una variable que no es una cadena. Este error es particularmente importante cuando se trabaja con objetos.

Este es un `run-time error` típico, por lo que se lanzará la excepción apropiada mientras se ejecuta el programa, después de llegar a la instrucción problemática.

```
const someConstValue = 5;
someConstValue = 7; // -> Uncaught TypeError: Assignment to constant variable.
```

Range error

Este tipo de error se genera cuando pasas un valor a una función que está fuera de su rango aceptable.

Nuevamente, es un ***run-time error***, y la excepción se lanza mientras el programa se está ejecutando, después de llegar a la instrucción incorrecta. De hecho, esta excepción es más útil al escribir tus propias funciones y manejar errores. Luego puedes lanzar una excepción en ciertas situaciones.

```
let testArray1 = Array(10);
console.log(testArray1.length); // -> 10
let testArray2 = Array(-1); // -> Uncaught RangeError: Invalid array length
console.log(testArray2.length);
```

En el ejemplo, hemos intentado crear dos arreglos usando el constructor (es decir, la función predeterminada) `Array`. Si pasamos un argumento a esta función, se tratará como el tamaño del arreglo recién creado. El primer arreglo (`testArray1`) se crea sin ningún problema. Como puedes adivinar, falla la creación del arreglo `testArray2` con una longitud negativa.

La sentencia try catch

Las excepciones interrumpen la ejecución del programa. La instrucción `try ... catch`, te permite cambiar esta acción predeterminada. El programa interrumpirá lo que está haciendo actualmente, pero no terminará automáticamente. La sintaxis para `try...catch` se ve así:

```
try {
    //codigo a probar
} catch (error) {
```

```
    // código a ejecutar en caso de un error, que lanza una excepción
}
```

La premisa básica es simple: si tenemos un fragmento de código que posiblemente pueda salir mal, podemos incluirlo en la cláusula `try`. JavaScript intentará ejecutar este código, y si ocurre algún error y se lanza una excepción, se ejecutará el código dentro del bloque `catch`; si el código `try` se ejecuta sin errores, entonces se ignora el bloque `catch`. Después de ejecutar los comandos del bloque `catch`, el programa continuará ejecutándose desde la primera instrucción fuera de la instrucción `try ... catch`.

Toma en cuenta que la palabra clave `catch` va seguida de paréntesis que contienen el error de parámetro. Este es un nombre de variable que contendrá información sobre el error que se detectó, y puede tener cualquier nombre válido, excepto los nombres `error`, `err` o simplemente `e`, son de uso común. En el caso de una excepción lanzada por JavaScript, el objeto de error contendrá información sobre el tipo de error y se convierte en una cadena para ser registrada o procesada de cualquier forma que necesite el desarrollador.

Toma en cuenta que `try...catch` no funcionará en un `SyntaxError`. Esto no debería ser una sorpresa para ti. Como hemos dicho varias veces antes, si el motor JavaScript detecta un *error de sintaxis*, no te permitirá ejecutar el programa. Si el programa no se ha ejecutado, es bastante difícil imaginar que pueda reaccionar de alguna manera a lo que ha sucedido.

Manejo de excepciones condicionales

A veces queremos poder reaccionar de manera diferente a tipos específicos de errores dentro del bloque `catch`. Podemos hacer esto usando el operador `instanceof`. La sintaxis del operador `instanceof` tiene este aspecto:

```
variable instanceof type
```

Si, por ejemplo, recibimos un error en el bloque `catch` (pasado como argumento de error), podemos comprobar si es del tipo `ReferenceError` de la siguiente manera :

```
let result = error instanceof ReferenceError;
```

El operador `instanceof` devuelve un valor booleano, por lo que esta expresión devolverá `true` si la variable de error contiene un tipo `ReferenceError` y false si no es así. Podemos usar `if...else` o sentencias switch para luego ejecutar un código diferente en el caso de diferentes errores si es necesario.

Es importante saber que cualquier variable que se declare usando la palabra clave `let` dentro de un bloque `try` no es accesible en el bloque `catch` (ni en el bloque `finally`).

La sentencia finally

El último bloque opcional de la sentencia `try` es el bloque `finally`. Se puede usar con o sin el bloque `catch`, y siempre se ejecuta después de los bloques `try` y `catch`, independientemente de que se produzca algún error. La sintaxis para `try ... finally` se ve así:

```
try {
    // código a probar
} finally {
    // esto siempre se ejecutará
}
```

El bloque `finally` también se puede usar junto con el bloque `catch`, ya que ambos son opcionales, pero al menos uno de ellos es requerido por `try`, y si ninguno de ellos está presente, se lanza un `SyntaxError`.

Los bloques `try...catch...finally` se pueden anidar, por lo que podemos usar un bloque completo `try...catch` dentro de otro bloque `try...catch`. Esto es útil cuando esperamos que ocurran múltiples errores y necesitamos manejarlos todos.

```
let a = 10;
try {
    a = b; // Primer ReferenceError
} catch (error) {
    try {
        console.log(b); // Segundo ReferenceError
    } catch {
        console.log("¡Segundo catch!"); // -> ¡Segundo catch!
    }
} finally {
    console.log("¡Finalmente!"); // -> ¡Finalmente!
}
```

La instruccion throw y errores personalizados

Para lanzar una excepción, usamos la instrucción throw. Le sigue cualquier valor que será tratado como una excepción. Puede ser, por ejemplo, un número o uno de los objetos de error preparados (por ejemplo, `RangeError`).

Una excepción que lancemos hará que el programa reaccione de la misma manera que las excepciones de JavaScript originales (es decir, detendrá su ejecución). Es decir, a menos que lo arrojemos dentro del bloque try para manejarlo. Este es un ejemplo sencillo de mandar una excepción

```
console.log("inicio"); // -> inicio
throw 100; // -> Uncaught 100
console.log("fin");
```

Con este ejemplo, quedara más claro. Esta función calcula el factorial de un número de forma iterativa. Digamos que estamos un poco asustados por los grandes números que devuelve nuestra función, especialmente el valor `Infinity`, por lo que decidimos limitar el rango máximo de valores admitidos. No aceptaremos argumentos mayores de 20.

```
function factorial(n) {
    if (n > 20) {
        throw new RangeError("Valor máximo 20");
    }
    let result = 1;
    for (; n > 1; n--) {
        result = result * n;
    }
    return result;
}

console.log(factorial(20)); // -> 2432902008176640000
console.log(factorial(1000)); // -> Uncaught RangeError: Valor máximo 20
```

Debbuger

Debbugear nuestro código es muy importante, ya que nos permite encontrar errores en nuestro código. Para llevarse a cabo de manera eficiente, la depuración requiere herramientas, y si nuestro código se ejecuta en el navegador, es casi seguro que ya

tenemos todas las herramientas necesarias disponibles. Para comprobar si nuestro navegador admite esta funcionalidad, simplemente podemos intentar ejecutar este código con la consola del desarrollador.

```
console.log("Antes del depurador");
debugger;
console.log("Despues del depurador");
```

Si el depurador está presente, la consola mostrará solo el mensaje "Antes del depurador" y, según el navegador instalado, deberíamos ver información sobre la ejecución del código detenida, pausada o en modo de depuración. El segundo mensaje no se muestra porque la sentencia debugger funciona como un punto de interrupción en la ejecución del código. Entonces, cada vez que JavaScript encuentra la sentencia debugger, verifica si el depurador está presente y, de ser así, la ejecución del código se detiene en ese punto exacto. Por supuesto, esto no es útil en sí mismo, pero es solo el comienzo de las características del depurador.

Una de las características principales del depurador es su capacidad para ejecutar código **paso a paso**. Esto significa que podemos detener la ejecución del programa en cualquier lugar usando una instrucción `debugger` y luego continuar la ejecución una instrucción a la vez. Algunos controles nos permiten:

- **Reanudar/Continuar**. Esto reanudará la ejecución del script de forma normal y se usa cuando hemos verificado lo que queríamos verificar y ahora queremos continuar con la ejecución del script.
- **Step Into**. Esto ejecuta la siguiente instrucción en el código solamente, y lo pausa de nuevo, y lo usamos cuando queremos analizar el código en detalle, o verificar qué ruta exacta toma la ejecución cuando ocurre una bifurcación compleja debido a las sentencias `if. ..else` u otra lógica complicada. Si la siguiente instrucción es una llamada de función, usar Step Into saltará dentro del código de esta función.
- **Step Over**. Esto funciona como Step Into, excepto que si se usa cuando la siguiente instrucción es una llamada de función, el código no saltará al código de función, pero se ejecutará toda la función, y el programa se pausará nuevamente después de salir de esta función. Esto se usa a menudo cuando la siguiente instrucción es una llamada a una función en la que no sabemos si tendrá algún impacto, o simplemente no estamos interesados en buscar.

- **Step Out**. Esto nos permite salir inmediatamente de una función en la que el código está en pausa.