**Depuración**

**Usa la consola de JavaScript para comprobar el valor de una variable**

Tanto Chrome como Firefox tienen excelentes consolas de JavaScript, también conocidas como DevTools, para depurar tu JavaScript.

Puedes encontrar las herramientas para desarrolladores en el menú de Chrome o la consola web en el menú de Firefox. Si utilizas otro navegador, o un teléfono móvil, te recomendamos encarecidamente que cambies a Firefox o Chrome de escritorio.

El método console.log(), que "imprime" la salida de lo que está dentro de sus paréntesis a la consola, será probablemente la herramienta de depuración más útil. Colocarlo en puntos estratégicos de tu código puede mostrarte los valores intermedios de las variables. Es una buena práctica tener una idea de lo que debería ser la salida antes de ver lo que es. Tener puntos de control para ver el estado de tus cálculos a lo largo de tu código ayudará a acotar dónde está el problema.

Aquí hay un ejemplo para imprimir la cadena Hello world! en la consola:

**console.log('Hello world!');**

**let a = 5;**

**let b = 1;**

**a++;**

**// Cambia solo el código debajo de esta línea**

**console.log(a)**

**let sumAB = a + b;**

**console.log(sumAB);**

**Entendiendo las diferencias entre la consola de freeCodeCamp y la del navegador**

Habrás notado que algunos desafíos de freeCodeCamp tienen su propia consola. Esta consola se comporta un poco diferente a la consola del navegador.

Hay muchos métodos para usar con console para mostrar mensajes. log, warn y clear para nombrar algunos. La consola de freeCodeCamp solamente mostrará mensajes de tipo log, mientras que la consola del navegador mostrará todos los mensajes. Cuando hagas cambios a tu código, se ejecutara automáticamente y mostrará los registros. La consola de freeCodeCamp se borra cada vez que ejecutas tu código.

**let output = "Get this to show once in the freeCodeCamp console and not at all in the browser console";**

**console.log(output)**

**console.clear()**

**tiliza typeof para comprobar el tipo de una variable**

Puedes utilizar typeof para comprobar la estructura de datos, o tipo, de una variable. Esto es útil en la depuración cuando se trabaja con múltiples tipos de datos. Si crees que estás sumando dos números, pero uno es en realidad una cadena, los resultados pueden ser inesperados. Los errores de tipo pueden estar al acecho en los cálculos o en las llamadas a funciones. Ten cuidado especialmente cuando accedas y trabajes con datos externos en forma de objeto de JavaScript Object Notation (JSON).

**Aquí hay algunos ejemplos que utilizan typeof:**

**console.log(typeof "");**

**console.log(typeof 0);**

**console.log(typeof []);**

**console.log(typeof {});**

En orden, la consola mostrará las cadenas string, number, object, y object.

JavaScript reconoce siete tipos de datos primitivos (inmutables): Boolean, Null, Undefined, Number, String, Symbol (nuevo con ES6), y BigInt (nuevo con ES2020), y un tipo para elementos mutables: Object. Ten en cuenta que en JavaScript, los arreglos son técnicamente un tipo de objeto.

**let seven = 7;**

**let three = "3";**

**console.log(seven + three);**

**// Cambia solo el código debajo de esta línea**

**console.log(typeof(seven));**

**console.log(typeof(three));**

**Captura nombres de variables y funciones mal escritos**

Los métodos console.log() y typeof son las dos formas principales de comprobar los valores intermedios y los tipos de salida de un programa. Ahora es el momento de entrar en las formas comúnes que adoptan los errores (bugs). Un problema a nivel de sintaxis con el que las personas que escriben rápido pueden simpatizar es el humilde error ortográfico.

Los caracteres transpuestos, omitidos o mal escritos en el nombre de una variable o función harán que el navegador busque un objeto que no existe, y se queje en forma de error de referencia. Los nombres de variables y funciones de JavaScript distinguen entre mayúsculas y minúsculas.

**let receivables = 10;**

**let payables = 8;**

**let netWorkingCapital = receivables - payables;**

**console.log(`Net working capital is: ${netWorkingCapital}`);**

**Captura paréntesis, corchetes, llaves y comillas sin cerrar**

Otro error de sintaxis a tener en cuenta es que todos los paréntesis de apertura, corchetes, llaves y comillas tienen un par de cierre. Olvidar una pieza suele suceder cuando se edita el código existente y se insertan elementos con uno de los tipos de pares. También hay que tener cuidado al anidar bloques de código dentro de otros, como agregar una función de callback como argumento de un método.

Una forma de evitar este error es, tan pronto como se escriba el caracter de apertura, incluir inmediatamente su caracter de cierre, luego mover el cursor hacia atrás entre ellos y continuar escribiendo. Afortunadamente, la mayoría de los editores de código modernos generan la segunda mitad del par automáticamente.

**let myArray = [1, 2, 3];**

**let arraySum = myArray.reduce((previous, current) => previous + current);**

**console.log(`Sum of array values is: ${arraySum}`);**

**Captura el uso mixto de comillas simples y dobles**

JavaScript permite el uso de comillas simples (') y dobles (") para declarar una cadena. Decidir cuál usar se reduce generalmente a la preferencia personal, con algunas excepciones.

Tener dos opciones es genial cuando una cadena tiene contracciones u otro fragmento de texto que está entre comillas. Sólo hay que tener cuidado de no cerrar la cadena demasiado pronto, lo que provoca un error de sintaxis.

Aquí hay algunos ejemplos de comillas mezcladas:

**const grouchoContraction = "I've had a perfectly wonderful evening, but this wasn't it.";**

**const quoteInString = "Groucho Marx once said 'Quote me as saying I was mis-quoted.'";**

**const uhOhGroucho = 'I've had a perfectly wonderful evening, but this wasn't it.';**

Los dos primeros son correctos, pero el tercero es incorrecto.

Por supuesto, está bien utilizar sólo un estilo de comillas. Puedes escapar las comillas dentro de una cadena usando el carácter de barra diagonal invertida (\):

**const allSameQuotes = 'I\'ve had a perfectly wonderful evening, but this wasn\'t it.';**

**let innerHtml = "<p>Click here to <a href='#Home'>return home</a></p>";**

**console.log(innerHtml);**

**Captura el uso del operador de asignación en lugar del**

**operador de igualdad**

Los programas de bifurcación (branching), es decir, los que hacen cosas diferentes si se cumplen ciertas condiciones, se basan en las sentencias if, else if y else de JavaScript. La condición a veces toma la forma de probar si un resultado es igual a un valor.

Esta lógica se habla (en español, al menos) como "si x es igual a y, entonces...", lo que puede traducirse literalmente en código utilizando el =, u operador de asignación. Esto lleva a un flujo de control inesperado en tu programa.

Como hemos visto en desafíos anteriores, el operador de asignación (=) en JavaScript asigna un valor a una variable. Y los operadores == y === comprueban la igualdad (el triple === comprueba la igualdad estricta, lo que significa que tanto el valor como el tipo son iguales).

El código siguiente asigna a x el valor de 2, que se evalúa como true. Casi todos los valores por sí solos en JavaScript se evalúan como true, excepto lo que se conoce como valores "falsos" (falsy values): false, 0, "" (una cadena vacía), NaN, undefined, y null.

**let x = 1;**

**let y = 2;**

**if (x = y) {**

**} else {**

**}**

En este ejemplo, el bloque de código dentro de la sentencia if se ejecutará para cualquier valor de y, a menos que y sea algún valor falsy. El bloque else, que esperamos que se ejecute aquí, no se ejecutará realmente.

**let x = 7;**

**let y = 9;**

**let result = "to come";**

**if(x == y) {**

**result = "Equal!";**

**} else {**

**result = "Not equal!";**

**}**

**console.log(result);**

**Captura los paréntesis de apertura y cierre que faltan**

**después de una llamada a una función**

Cuando una función o método no recibe argumentos, puedes olvidarte de incluir los paréntesis de apertura y cierre (vacíos) al llamarla. A menudo, el resultado de una llamada a una función se guarda en una variable para su uso en el código. Este error puede detectarse registrando los valores de las variables (o sus tipos) en la consola y viendo que uno de ellos se establece como una referencia a la función, en lugar del valor esperado que la función devuelve.

Las variables del siguiente ejemplo son diferentes:

**function myFunction() {**

**return "You rock!";**

**}**

**let varOne = myFunction;**

**let varTwo = myFunction();**

**Aquí varOne es la función myFunction, y varTwo es la cadena You rock!.**

**function getNine() {**

**let x = 6;**

**let y = 3;**

**return x + y;**

**}**

**let result = getNine();**

**console.log(result);**

**Captura argumentos pasados en el orden**

**incorrecto al llamar a una función**

Siguiendo con la discusión sobre la llamada a funciones, el siguiente error a tener en cuenta es cuando los argumentos de una función se suministran en el orden incorrecto. Si los argumentos son de tipos diferentes, como una función que espera un arreglo y un entero, es probable que se produzca un error de ejecución. Si los argumentos son del mismo tipo (todos enteros, por ejemplo), la lógica del código no tendrá sentido. Asegúrate de proporcionar todos los argumentos requeridos, en el orden correcto para evitar estos problemas.

**function raiseToPower(b, e) {**

**return Math.pow(b, e);**

**}**

**let base = 2;**

**let exp = 3;**

**let power = raiseToPower(base, exp);**

**console.log(power);**

**Captura los errores por uno al utilizar indexación**

Los errores por uno o por un paso (en inglés: Off-by-one error -OBOE) aparecen cuando se intenta apuntar a un índice específico de una cadena o arreglo (para cortar o acceder a un segmento), o cuando se hace un bucle sobre los índices de los mismos. La indexación en JavaScript comienza en cero, no en uno, lo que significa que el último índice es siempre uno menos que la longitud del elemento. Si intentas acceder a un índice igual a la longitud, el programa puede lanzar un error de referencia "index out of range" (índice fuera de rango) o imprimir undefined.

Cuando se utilizan métodos de cadenas o arreglos que toman rangos de índices como argumentos, es útil leer la documentación y entender si son inclusivos (el elemento en el índice dado es parte de lo que se devuelve) o no. Estos son algunos ejemplos de errores por un paso:

**let alphabet = "abcdefghijklmnopqrstuvwxyz";**

**let len = alphabet.length;**

**for (let i = 0; i <= len; i++) {**

**console.log(alphabet[i]);**

**}**

**for (let j = 1; j < len; j++) {**

**console.log(alphabet[j]);**

**}**

**for (let k = 0; k < len; k++) {**

**console.log(alphabet[k]);**

**}**

El primer ejemplo aquí hace un bucle de más, y el segundo hace un bucle de menos (falta el primer índice, 0). El tercer ejemplo es correcto.

**function countToFive() {**

**let firstFive = "12345";**

**let len = firstFive.length;**

**// Cambia solo el código debajo de esta línea**

**for (let i = 0; i < len; i++) {**

**// Cambia solo el código encima de esta línea**

**console.log(firstFive[i]);**

**}**

**}**

**countToFive();**

**Ten cuidado al reinicializar variables dentro de un bucle**

A veces es necesario guardar información, incrementar contadores o reajustar variables dentro de un bucle. Un problema potencial es cuando las variables deberían ser reiniciadas y no lo son, o viceversa. Esto es particularmente peligroso si accidentalmente se restablece la variable que se utiliza para la condición terminal, causando un bucle infinito.

La impresión de los valores de las variables con cada ciclo de su bucle mediante el uso de console.log() puede descubrir un comportamiento erróneo relacionado con el restablecimiento, o la falta de restablecimiento de una variable.

**function zeroArray(m, n) {**

**// Crea un arreglo de 2 dimensiones con m filas y n columnas de ceros**

**let newArray = [];**

**for (let i = 0; i < m; i++) {**

**// Agrega la fila número m a newArray**

**let row = [];**

**for (let j = 0; j < n; j++) {**

**// Inserta n ceros a la fila actual para crear las columnas**

**row.push(0);**

**}**

**// Inserta la fila actual, que ahora contiene n ceros, al arreglo**

**newArray.push(row);**

**}**

**return newArray;**

**}**

**let matrix = zeroArray(3, 2);**

**console.log(matrix);**

**Prevenir bucles infinitos con una condición terminal válida**

El último tema es el temido bucle infinito. Los bucles son una gran herramienta cuando necesitas que tu programa ejecute un bloque de código un determinado número de veces o hasta que se cumpla una condición, pero necesitan una condición terminal para que finalice el bucle. Los bucles infinitos pueden congelar o bloquear el navegador, y causar un caos en la ejecución del programa en general, lo que nadie quiere.

Había un ejemplo de un bucle infinito en la introducción de esta sección - no tenía ninguna condición terminal para salir del bucle while dentro de loopy(). ¡NO llames esta función!

**function loopy() {**

**while(true) {**

**console.log("Hello, world!");**

**}**

**}**

El trabajo del programador es asegurarse de que la condición terminal, que indica al programa cuándo debe salir del código del bucle, se alcance finalmente. Un error es incrementar o decrementar una variable de contador en la dirección incorrecta desde la condición terminal. Otra es reiniciar accidentalmente un contador o una variable de índice dentro del código del bucle, en lugar de incrementarlo o decrementarlo.

**function myFunc() {**

**for (let i = 1; i <= 4; i += 2) {**

**console.log("Still going!");**

**}**

**}**