**Universidad Autónoma de Nuevo León.**

**Facultad de Ciencias Físico Matemática.**

[**Licenciatura en Seguridad en Tecnologías de Información**](http://www.uanl.mx/oferta/licenciatura-en-seguridad-en-tecnologias-de-informacion.html) **Profesor: Lic. Miguel Angel Salazar Santillán**

**Alumna: Sofía Esmeralda Pecina Rojas.**

**Matricula: 1667093  
  
Unidad de aprendizaje: Diseño orientado a objetos  
  
tema: 10 cosas extrañas de javascript**

**San Nicolás de la garza 27 de enero del 2017**

Contenido

[1.- NULL ES UN OBJETO 3](#_Toc473288402)

[2.- NAN ES UN NUMERO 3](#_Toc473288403)

[3.- UNA MATRIZ SIN TECLAS == FALSO 4](#_Toc473288404)

[4. REPLACE () PUEDE ACEPTAR UNA FUNCIÓN DE DEVOLUCIÓN DE LLAMADA 5](#_Toc473288405)

[5. EXPRESIONES REGULARES: MAS QUE SOLO MATCH Y REPLACE 5](#_Toc473288406)

[6. TU PUEDES FAKE SCOPE 6](#_Toc473288407)

[7.- FUNCIONES QUE PUEDEN EJECUTARSE ELLOS MISMOS 7](#_Toc473288408)

[8. FIREFOX LEE Y DEVUELVE COLORES EN RGB, NO HEXADECIMAL 8](#_Toc473288409)

[9.- 0.1 + 0.2! == 0.3 8](#_Toc473288410)

[10. UNDEFINED PUEDE SER DEFINIDO 9](#_Toc473288411)

# **1.- NULL ES UN OBJETO**

**alert(typeof null); //alerts 'object'**

El valor null es un literal de Javascript que representa un valor nulo o "vacío". Null no se considera una instancia de un objeto. (En caso de que no lo supieras, los valores en JavaScript son instancias de objetos base, por lo que cada número es una instancia del objeto Number , cada objeto es una instancia del objeto Object y así sucesivamente). Sanidad, porque si nula es la ausencia de valor, entonces obviamente no puede ser un ejemplo de nada. Por lo tanto, lo siguiente se evalúa como false :

alert(null instanceof Object); //evaluates false

# **2.- NAN ES UN NUMERO**

Un valor representando un Not-A-Number (No es Un Número).

|  |  |
| --- | --- |
| Atributos de la propiedad NaN | |
| Sobrescribir | No |
| Numerable | No |
| Configurable | No |

NaN es una propiedad del global object (objeto global), por ejemplo, es una variable de alcance global.

El valor inicial de NaN es Not-A-Number (No es Un Número) - lo mismo que el valor de [Number.NaN](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Number/NaN" \o "The documentation about this has not yet been written; please consider contributing!). NaN nunca es equivalente con cualquier otro número, incluido el mismo NaN; no puedes chequear el valor de un not-a-number comparándolo con Number.NaN. Usar la función [isNaN()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/isNaN()" \o "The documentation about this has not yet been written; please consider contributing!)para aquello.

Muchos métodos de JavaScript (como son el [Number](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Number" \o "Permite trabajar con valores numéricos. El objeto Number es un objeto envolvente para valores primitivos numéricos.) constructor, [parseFloat](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/parseFloat" \o "Convierte (parsea) un argumento de tipo cadena y devuelve un número de punto flotante.) y [parseInt](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/parseInt" \o "Convierte (parsea) un argumento de tipo cadena y devuelve un entero de la base especificada.)) retornan NaN si el valor especificado en el parámetro no puede ser parseado como un número. Puedes utilizar la propiedad NaN para indicar una condición de error para tu función que devuelva un número en caso de éxito. JavaScript imprime el valor Number.NaN como NaN.

**alert(typeof NaN); //alerts 'Number' alert(NaN === NaN); //evaluates false**

De hecho NaN no es igual a nada . La única manera de confirmar que algo es NaN es a través de la función isNaN() .

# **3.- UNA MATRIZ SIN TECLAS == FALSO**

**alert(new Array() == false); //evaluates true**

Para entender lo que está sucediendo aquí, usted necesita entender los conceptos de la verdad y falso . Estos son una especie de verdad / falso-lite, que le enojará un poco si se especializó en la lógica o la filosofía.

He leído muchas explicaciones de lo que son verdaderos y falsos, y creo que el más fácil de entender es esto: en JavaScript, cada valor no booleano tiene un indicador booleano incorporado que se llama cuando se pide al valor que se comporte como un booleano; Como, por ejemplo, cuando se compara con un booleano.

Debido a que las manzanas no pueden compararse con las peras, cuando se le pide a JavaScript que compare valores de diferentes tipos de datos, primero los " coacciona " a un tipo de datos común. False ,  zero , null , undefined , cadenas vacías y NaN todos terminan convirtiéndose en false - no permanentemente, sólo para la expresión dada. Un ejemplo para el rescate:

**var someVar = 0; alert(someVar == false); //evaluates true**

Aquí, estamos tratando de comparar el número 0 con el booleano false . Debido a que estos tipos de datos son incompatibles, JavaScript secretamente coactiva nuestra variable en su equivalente truthy o falsy , que en el caso de 0 (como he dicho arriba) es falsy.

Usted puede haber notado que no incluí arrays vacíos en la lista de falsies arriba. Las matrices vacías son cosas curiosas: en realidad se evalúan a la verdad pero , cuando se comparan con un booleano, se comportan como un falso. ¿Todavía confundido? Con buena causa. Otro ejemplo quizás?

**var someVar = []; //empty array alert(someVar == false); //evaluates true if (someVar) alert('hello'); //alert runs, so someVar evaluates to true**

Para evitar la coerción, puede utilizar el operador de comparación de valores y tipos , === , (en contraposición a == , que sólo se compara con el valor). Asi que:

**var someVar = 0; alert(someVar == false); //evaluates true – zero is a falsy alert(someVar === false); //evaluates false – zero is a number, not a boolean**

Uf. Como probablemente se ha reunido, este es un tema amplio, y recomiendo leer más sobre él - en particular sobre la coerción de datos, que, aunque no es un concepto exclusivo de JavaScript, es no obstante importante en JavaScript.

# **4. REPLACE () PUEDE ACEPTAR UNA FUNCIÓN DE DEVOLUCIÓN DE LLAMADA**

El método replace() halla un emparejamiento entre una expresión regular y una cadena, y reemplaza la subcadena emparejada con una nueva subcadena. Este método no cambia el objeto String sobre el que se realiza la llamada. Simplemente devuelve una nueva cadena. Este es uno de los secretos mejor guardados de JavaScript y llegó en v1.3. La mayoría de los usos de replace() parecen a esto:

**alert('10 13 21 48 52'.replace(/d+/g, '\*')); //replace all numbers with**

Se trata de un simple reemplazo: una cadena, un asterisco. Pero ¿y si queríamos más control sobre cómo y cuándo suceden nuestros reemplazos? ¿Qué pasa si queremos reemplazar sólo los números menores de 30? Esto no puede lograrse con expresiones regulares solas (son todas sobre cuerdas, después de todo, no matemáticas). Tenemos que saltar a una función de devolución de llamada para evaluar cada partido .

**alert('10 13 21 48 52'.replace(/d+/g, function(match) { return parseInt(match) < 30 ? '\*' : match; }));**

Para cada partido realizado, JavaScript llama a nuestra función, pasando el partido en nuestro argumento de partido. Luego, devolvemos el asterisco (si el número coincide con menos de 30) o el partido en sí (es decir, no debe tener lugar ningún partido).

# **5. EXPRESIONES REGULARES: MAS QUE SOLO MATCH Y REPLACE**

Muchos desarrolladores de JavaScript intermedios se limitan en match y replace con expresiones regulares. Pero JavaScript define más métodos que estos dos.

De particular interés es test() , que funciona como match excepto que no devuelve coincidencias: simplemente confirma si un patrón coincide . En este sentido, es computacionalmente más ligero.

**alert(/w{3,}/.test('Hello')); //alerts 'true'**

Lo anterior busca un patrón de tres o más caracteres alfanuméricos, y debido a que la cadena Hello cumple con ese requisito, obtenemos la true . No conseguimos el partido real, solo el resultado.

También cabe destacar el objeto RegExp , mediante el cual se pueden crear expresiones regulares dinámicas, en oposición a las static. La mayoría de las expresiones regulares se declaran utilizando forma corta (es decir, encerrado en barras inclinadas, como lo hicimos anteriormente). De esa manera, sin embargo, no se puede hacer referencia a variables, por lo que la creación de patrones dinámicos es imposible . Con RegExp() , sin embargo, puede hacerlo.

**function findWord(word, string) { var instancesOfWord = string.match(new RegExp('b'+word+'b', 'ig')); alert(instancesOfWord); } findWord('car', 'Carl went to buy a car but had forgotten his credit card.');**

Aquí, estamos haciendo un patrón dinámico basado en el valor de la word . La función devuelve el número de veces que la word aparece en cadena como una palabra por derecho propio (es decir, no como parte de otras palabras). Por lo tanto, nuestro ejemplo devuelve car una vez, ignorando las fichas de car en las palabras Carl y card . Obliga esto a buscar un límite de palabras ( b ) a cada lado de la palabra que estamos buscando.

Debido a que RegExp se especifican como cadenas, no a través de la sintaxis de barra diagonal, podemos utilizar variables en la construcción del patrón. Esto también significa, sin embargo, que debemos escapar dos caracteres especiales, como lo hicimos con nuestro carácter de frontera de la palabra.

# **6. TU PUEDES FAKE SCOPE**

El ámbito en el que se ejecuta algo define qué variables son accesibles. JavaScript autónomo (es decir, JavaScript que no se ejecuta dentro de una función) opera dentro del ámbito global del objeto de window , al que todo tiene acceso; Mientras que las variables locales declaradas dentro de las funciones son accesibles sólo dentro de esa función, no fuera.

**var animal = 'dog'; function getAnimal(adjective) { alert(adjective+' '+this.animal); } getAnimal('lovely'); //alerts 'lovely dog';**

Aquí, nuestra variable y función se declaran en el ámbito global (es decir, en la window ). Debido a que siempre apunta al ámbito actual, en este ejemplo apunta a window . Por lo tanto, la función busca window.animal , que window.animal . Hasta ahora, tan normal. Pero en realidad podemos con nuestra función en el pensamiento de que está funcionando en un ámbito diferente, independientemente de su propio ámbito natural. Lo hacemos llamando a su método built-in call() , en lugar de la función en sí:

**var animal = 'dog'; function getAnimal(adjective) { alert(adjective+' '+this.animal); }; var myObj = {animal: 'camel'}; getAnimal.call(myObj, 'lovely'); //alerts 'lovely camel'**

Aquí, nuestra función no se ejecuta en la window pero en myObj - se especifica como el primer argumento del método de llamada. Esencialmente, call()pretende que nuestra función es un método de myObj (si esto no tiene sentido, es posible que desee leer en el sistema JavaScript de la herencia prototípica).Tenga en cuenta también que cualquier argumento que pasemos a call()después de la primera se pasará a nuestra función - por lo tanto estamos pasando en adjective como nuestro adjective .

Como un aparte, apply() hace el mismo trabajo que call() , excepto que los argumentos a la función se especifican como una matriz, en lugar de como argumentos individuales. Por lo tanto, el ejemplo anterior usando apply() se vería así:

**getAnimal.apply(myObj, ['lovely']); //func args sent as array**

# **7.- FUNCIONES QUE PUEDEN EJECUTARSE ELLOS MISMOS**

No se puede negar:

**(function() { alert('hello'); })(); //alerts 'hello'**

La sintaxis es bastante simple: declaramos una función y la llamamos inmediatamente igual que nosotros llamamos a otras funciones, con ()sintaxis. Usted puede preguntarse por qué haríamos esto. Parece una contradicción en términos: una función normalmente contiene código que queremos ejecutar más tarde, no ahora, de lo contrario no habría puesto el código en una función.

Un buen uso de las funciones de ejecución automática (SEF) es vincular los valores actuales de las variables para su uso dentro del código retrasado, como devoluciones de llamada a eventos, tiempos de espera e intervalos. Aquí está el problema:

**var someVar = 'hello'; setTimeout(function() { alert(someVar); }, 1000); var someVar = 'goodbye';**

Los novatos en los foros invariablemente preguntan por qué la alert en el timeout dice goodbye y no hello . La respuesta es que la función de devolución de llamada de timeout es precisamente eso - una devolución de llamada - por lo que no evalúa el valor de someVar hasta que se ejecuta. Y para entonces, someVar ha sido desde hace mucho tiempo sobrescrito por goodbye .

Los SEFs proporcionan una solución a este problema. En lugar de especificar la devolución de llamada de tiempo de espera implícitamente como lo hacemos anteriormente, la devolvemos desde un SEF, en el que pasamos el valor actual de someVar como argumentos. Efectivamente, esto significa que pasamos y aislamos el valor actual de someVar , protegiéndolo de lo que suceda a la variable real someVar partir de entonces . Esto es como tomar una foto de un coche antes de resprayarlo; La foto no se actualizará con el color resprayado; Siempre mostrará el color del coche en el momento de la foto fue tomada.

**var someVar = 'hello'; setTimeout((function(someVar) { return function() { alert(someVar); } })(someVar), 1000); var someVar = 'goodbye';**

Esta vez, alertas hello , como se desea, porque está someVar versión aislada de someVar (es decir, el someVar la función, no la variable externa).

# **8. FIREFOX LEE Y DEVUELVE COLORES EN RGB, NO HEXADECIMAL**

Se observa de que cualquiera que interroga los colores calculados a través de JavaScript está interesado en formato hexadecimal y no en RGB. Para aclarar, he aquí un ejemplo:

**Hello, world! <script> var ie = navigator.appVersion.indexOf('MSIE') != -1; var p = document.getElementById('somePara'); alert(ie ? p.currentStyle.color : getComputedStyle(p, null).color); </script>**

Mientras que la mayoría de los ff9900 , Firefox devuelve rgb(255, 153, 0) , el equivalente RGB. Un montón de funciones de JavaScript están ahí para convertir RGB a hexadecimal.

Tenga en cuenta que cuando digo que el color calculado, me refiero al color actual, independientemente de cómo se aplica al elemento . Compare esto con el estilo, que sólo lee las propiedades de estilo que se establecieron implícitamente en el atributo de estilo de un elemento. Además, como habrás notado en el ejemplo anterior, IE tiene un método diferente de detectar estilos computados de otros navegadores.

Como un aparte, el método css() jQuery abarca este tipo de detección computarizada, y devuelve estilos sin embargo se aplicaron a un elemento: implícitamente oa través de la herencia o lo que sea. Por lo tanto, relativamente rara vez se necesita el nativo getComputedStyle y currentStyle

# **9.- 0.1 + 0.2! == 0.3**

Esta es una rareza no sólo en JavaScript; En realidad es un problema predominante en la informática, y afecta a muchos idiomas. La salida de esto es 0.30000000000000004.

Esto tiene que ver con un problema llamado precisión de la máquina . Cuando JavaScript intenta ejecutar la línea anterior, convierte los valores a sus equivalentes binarios.

Aquí es donde comienza el problema. 0,1 no es realmente 0,1, sino más bien su equivalente binario, que es un valor cercano a near- ish (pero no idéntico) . En esencia, tan pronto como escribas los valores, están condenados a perder su precisión. Podrías haber querido dos decimales sencillos, pero lo que obtienes, como observa Chris Pine, es la aritmética binaria en coma flotante. Algo así como querer que su texto sea traducido al ruso pero obteniendo bielorruso. Similar, pero no es lo mismo.

Más está pasando aquí, pero está más allá del alcance de este artículo (por no hablar de las capacidades matemáticas de este autor).

Soluciones alternativas para este problema son un favorito en ciencias de la computación y foros de desarrolladores. Su elección, hasta cierto punto, se reduce a la clase de cálculos que está haciendo. Los pros y contras de cada uno están fuera del alcance de este artículo, pero la opción común es entre lo siguiente:

1. La conversión a enteros y el cálculo de los en su lugar, a continuación, convertir de nuevo a decimales después; o
2. Ajustar su lógica para permitir una gama en lugar de un resultado específico.

Así, por ejemplo, en lugar de ...

**var num1 = 0.1, num2 = 0.2, shouldEqual = 0.3; alert(num1 + num2 == shouldEqual); //false**

**... haríamos esto:**

**alert(num1 + num2 > shouldEqual - 0.001 && num1 + num2 < shouldEqual + 0.001); //true**

Traducido, esto dice que como 0,1 + 0,2 no es aparentemente 0,3, compruebe en su lugar que es más o menos 0,3 - específicamente, dentro de un rango de 0,001 a cada lado de la misma. El inconveniente obvio es que, para cálculos muy precisos, esto devolverá resultados inexactos.

# **10. UNDEFINED PUEDE SER DEFINIDO**

Por extraño que parezca, undefined no es una palabra reservada en JavaScript, aunque tiene un significado especial y es la única manera de determinar si una variable es indefinida. Asi que:

**var someVar; alert(someVar == undefined); //evaluates true**

tan normal. Pero:

**undefined = "I'm not undefined!"; var someVar; alert(someVar == undefined); //evaluates false!**