Universidad Autónoma De Nuevo León

Facultad de Ciencias Físico Matemáticas

LSTI

Maestro: Miguel Salazar

Alumno: Kenneth Rigoberto Rodríguez Pinto

Materia: Diseño Orientado a Objetos

Tema: 10 cosas extrañas de JavaScript

26 de enero del 2017 Guadalupe N.L.

10 cosas extrañas en JavaScript

1.Null es un Objeto:

¿Null? ¿Un objeto? "Sin duda, la definición de null es la total ausencia de valor significativo ", dice usted. Estaríamos en lo cierto. Pero esa es la manera que es. Aquí está la prueba:

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

A pesar de esto, null no se considera una instancia de un objeto. (En caso de que no lo sabía, los valores en JavaScript son instancias de objetos de la base. Así, cada número es una instancia del Numberobjeto, cada objeto es una instancia del Object objeto, y así sucesivamente.) Esto nos trae de vuelta a la cordura, porque si null es la ausencia de valor, entonces es obvio que puede no ser una instancia de nada. Por lo tanto, la siguiente se evalúa como false:

```
alert (null instance of Object); //evaluates false
```

2. NaN es un número:

NaN- “no es un número” - ser un número! ¡Por otra parte, NaN no se considera igual a sí mismo!

```
alert(typeof NaN); //alerts 'Number'
```

```
alert(NaN === NaN); //evaluates false
```

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

3. array() ‘==’ False es True:

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

Para entender lo que está pasando aquí, es necesario comprender los conceptos de Truthy y Falsy . Estos son una especie de verdadero / falso-Lite, que a ira tanto si se especializó en la lógica o la filosofía.

He leído muchas explicaciones de lo Truthy y Falsy son, y me siento la más fácil de entender es la siguiente: en JavaScript, todos los valores no booleana tiene incorporado un indicador booleano que se llama

cuando se le pide el valor de comportarse como un valor lógico; como, por ejemplo, cuando se compara con un valor lógico.

Debido a que las manzanas no se pueden comparar con las peras, cuando se le pide JavaScript para comparar los valores de diferentes tipos de datos, que primero " coacciona " ellos en un tipo de datos común. False, zero, null, undefined, Cadenas vacías y NaNtodos terminan convirtiéndose false- no de forma permanente, sólo por la expresión dada. Un ejemplo al rescate:

```
var someVar = 0;

alert(someVar == false); //evaluates true
```

En este caso, estamos tratando de comparar el número 0de la booleano false. Debido a que estos tipos de datos son incompatibles, JavaScript coacciona en secreto nuestra variable en su equivalente Truthy o Falsy , que en el caso de 0(como he dicho anteriormente) es Falsy.

Usted puede haber notado que no he incluido matrices vacías en la lista de los postizos anteriores. Matrices vacías son cosas curiosas: en realidad se evalúan como Truthy pero , cuando se compara con un valor lógico, se comportan como un Falsy. Confundido todavía? Con una buena causa. Otro ejemplo, tal vez?

```
var someVar = []; //empty array

alert(someVar == false); //evaluates true

if (someVar) alert('hello'); //alert runs, so someVar evaluates to true
```

Para evitar la coacción, se puede utilizar el operador de valor y la comparación de tipos , ===, (a diferencia de ==, que se compara solamente por 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
```

4. La función replace() acepta como parámetro funciones callback:

Este es uno de los secretos mejor guardados de JavaScript y llegó a v1.3. La mayoría de los usos de replace()ser algo como esto:

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

Se trata de una simple sustitución: una cadena, un asterisco. Pero lo que si queríamos más control sobre cómo y cuándo los reemplazos se llevan a cabo? ¿Y si quisiéramos sustituir sólo números menores de 30 años? Esto no se puede lograr con las expresiones regulares por sí solos (que está sobre todos cuerdas, después de todo, no las 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;
}));
```

Por cada partido hecho, llamadas JavaScript nuestra función, pasando el partido a nuestro argumento partido. Entonces, volvemos ya sea el asterisco (si el número que coincide es menor de 30 años) o el partido en sí (es decir, ningún partido debería tener lugar).

5. Las expresiones regulares se pueden testear con test() además de con match()

Muchos desarrolladores de JavaScript intermedios sobrevivir sólo en matchy replacecon las expresiones regulares. Pero JavaScript define más métodos que estos dos.

De particular interés es test(), que funciona como matchla excepción de que no vuelva partidos: simplemente confirma si un patrón coincide.En este sentido, es computacionalmente más ligero.

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

Las miradas por encima de un patrón de tres o más caracteres alfanuméricos, y porque la cadena Hello cumple con ese requisito, obtenemos true. No conseguimos el partido real, sólo el resultado.

También hay que resaltar el RegExpobjeto, mediante el cual se pueden crear expresiones regulares dinámicos, en contraposición a los estáticos. La mayoría de las expresiones regulares se declaran utilizando la forma corta (es decir, encerrado en barras inclinadas, como lo hicimos anteriormente). De esa manera, sin embargo, se puede hacer referencia las variables, por lo que hacer patrones dinámicos es imposible. Con RegExp(), sin embargo, se puede.

```
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 del argumento word. La función devuelve el número de veces que wordaparece en cadena como una palabra en su propio derecho (es decir, no como parte de otras palabras). Por lo tanto, nuestro ejemplo se devuelve car una vez, haciendo caso omiso de las carfichas en las palabras Carly card. Obliga esto mediante la comprobación de un límite de palabra (b) a cada lado de la palabra que estamos buscando.

Debido a que RegExpson representados como cadenas, no a través de la sintaxis barra inclinada, podemos utilizar variables en la construcción del modelo. Esto también significa, sin embargo, que hay que hacer doble escapar ningún carácter especial, como lo hicimos con nuestra palabra carácter límite.

6. Puedes falsear el alcance de una variable o función:

El ámbito en el que se ejecuta algo define qué variables son accesibles. Free-standing JavaScript (es decir JavaScript que no se ejecuta dentro de una función) opera dentro del ámbito global del window objeto, a la que todo tiene acceso; mientras que las variables

locales declaradas en funciones sólo son accesibles dentro de esa función, no afuera.

```
var animal = 'dog';

function getAnimal(adjective) { alert(adjective+' '+this.animal); }

getAnimal('lovely'); //alerts 'lovely dog';
```

Aquí, nuestra variable y función se declararon tanto en el ámbito global (es decir, sobre window). Debido a que este siempre apunta al ámbito actual, en este ejemplo que apunta window. Por lo tanto, la función busca window. animal, el que se encuentra. Hasta ahora, todo normal. Pero en realidad podemos estafar nuestra función en el pensamiento de que se está ejecutando en un ámbito diferente, con independencia de su propio ámbito natural. Hacemos esto llamando a su base de call()método, en lugar de la propia función:

```
var animal = 'dog';

function getAnimal(adjective) { alert(adjective+' '+this.animal); };

var myObj = {animal: 'camel'};

getAnimal.call(myObj, 'lovely'); //alerts 'lovely camel'
```

En este caso, nuestra función se ejecuta no en window sino en myObj- especificado como el primer argumento del método llamada. En esencia, call()pretende que nuestra función es un método de myObj(si esto no tiene sentido, es posible que desee leer en el sistema de la herencia de prototipos de JavaScript). Tenga en cuenta también que los argumentos que pasamos a call()después de la primera parte se lo pasamos a nuestra función - por lo tanto estamos pasando en lovely como nuestro adjective argumento.

He oído los desarrolladores de JavaScript decir que han pasado años sin que ni siquiera tenga que usar esto, entre otras cosas porque un buen diseño de código garantiza que usted no necesita este humo y espejos. No obstante, es ciertamente interesante.

Como acotación al margen, apply()hace el mismo trabajo que call(), a excepción de que argumentos de la función se especifican como una

matriz, en lugar de argumentos como individuales. Por lo tanto, utilizando el ejemplo anterior apply()sería el siguiente:

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

7. Las funciones se pueden ejecutar a si mismas:

No se puede negar que:

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

La sintaxis es bastante simple: declaramos una función e inmediatamente llamamos al igual que nosotros llamamos otras funciones, con ()la sintaxis. Usted podría preguntarse por qué íbamos a hacer esto. Parece una contradicción en los términos: una función que normalmente contiene código que queremos ejecutar más adelante, no ahora, de lo contrario no habríamos poner el código en una función.

Un buen uso de las funciones de aplicación inmediata (SEF) es enlazar los valores actuales de las variables para su uso dentro de código de retraso, como las devoluciones de llamada a los acontecimientos, los tiempos de espera y los intervalos. Aquí está el problema:

```
var someVar = 'hello';

setTimeout(function() { alert(someVar); }, 1000);

var someVar = 'goodbye';
```

Novatos en foros invariablemente preguntan por qué el alerten el timeoutdice goodbye, no hello. La respuesta es que la timeoutfunción de devolución de llamada es precisamente que - una devolución de llamada - por lo que no se evalúa el valor de someVarhasta que se agote. Y para entonces, someVardesde hace mucho tiempo ha sido sobrescrito por goodbye.

SEF proporcionar una solución a este problema. En lugar de especificar el tiempo de espera de devolución de llamada de forma implícita como lo hacemos anterior, volverlo a partir de un SEF, en la que se pasa el valor actual de someVarcomo argumentos. Efectivamente, esto significa que pasamos en aislamos y el valor

actual de someVar, protegiéndolo de lo que suceda con la variable real someVara partir de entonces . Esto es como tomar una foto de un coche antes de que Respray; la foto no se actualizará con el color resprayed; se mostrará siempre el color del coche en el momento se tomó la foto.

```
var someVar = 'hello';

setTimeout((function(someVar) {

        return function()  { alert(someVar); }

})(someVar), 1000);

var someVar = 'goodbye';
```

Esta vez, alerta hello, según se desee, ya que está alertando a la versión aislada de someVar(es decir, el argumento de la función, no la variable externa).

8. Firefox no lee y devuelve los colores en hexadecimal sino en RGB:

Nunca he entendido realmente por qué Mozilla hace esto. Sin duda, se da cuenta de que cualquier persona interrogar colores calculadas a través de JavaScript está interesada en formato hexadecimal y no 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 navegadores le avise ff9900, Firefox devuelve rgb(255, 153, 0), el equivalente RGB. Un montón de funciones de JavaScript están ahí fuera para la conversión de RGB a hexagonal.

Tenga en cuenta que cuando digo computé color, me refiero al color actual, independientemente de cómo se aplica al elemento . Compare

esto con estilo, que sólo lee las propiedades de estilo que se han establecido implícitamente en atributo de estilo de un elemento. También, como usted habrá notado en el ejemplo anterior, el IE cuenta con un método diferente de detectar estilos calculadas a partir de otros navegadores.

Como acotación al margen, de jQuery css()método abarca este tipo de detección computarizada, y vuelve estilos sin embargo, fueron aplicadas a un elemento: implícita o por herencia o lo que sea. Por lo tanto, sería relativamente rara vez se necesita el nativo getComputedStyley currentStyle.

9. 0,1 + 0,2! == 0.3

Esta es una rareza no sólo en JavaScript; en realidad es un problema que prevalece en la informática, y que afecta a muchos idiomas. La salida de este es ,30000000000000004.

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

Aquí es donde empieza el problema. 0,1 no es realmente 0,1, sino más bien su equivalente binario, que es una cerca-ish (pero no idéntico) valor. En esencia, tan pronto como se escribe los valores, que están condenados a perder su precisión. Es posible que haya sólo quería dos decimales simples, pero lo que se obtiene, como notas de Chris Pine, es la aritmética binaria de punto flotante. Algo así como querer su texto traducido al ruso, pero conseguir bielorruso. Similar, pero no el mismo.

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

Soluciones a este problema son las favoritas en los foros de informática y desarrolladores. Su elección, a un punto, se reduce a la clase de cálculos que estás haciendo. Los pros y los contras de cada uno están más allá del alcance de este artículo, pero la elección es común entre los siguientes:

La conversión a números enteros y el cálculo de aquellos en su lugar, a continuación, volver a convertir decimales después; o

Ajustar su lógica para permitir un rango 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
```

... Nos volveremos a hacer esto:

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

Traducido, esto dice que debido a que 0,1 + 0,2 aparentemente no es 0.3, cheque en lugar de que es más o menos 0,3 - específicamente, dentro de un rango de 0.001 a cada lado de ella. El inconveniente obvio es que, para los cálculos muy precisos, esto devolverá resultados inexactos.

10. Undefined puede ser definido

OK, vamos a terminar con una tonta uno, más intrascendente. Por extraño que pueda parecer, undefined no es en realidad una palabra reservada en JavaScript, a pesar de que tiene un significado especial y es la única manera de determinar si una variable no está definida. Asi que:

```
var someVar;

alert(someVar == undefined); //evaluates true
```

Hasta ahora, todo normal. Pero:

```
undefined = "I'm not undefined!";

var someVar;

alert(someVar == undefined); //evaluates false!
```