  
Universidad Autónoma de Nuevo León   
Facultad de Ciencias Físico Matemáticos

Licenciado en Seguridad en Tecnologías de Información

Diseño Orientado a Objetos

The weird things in JavaScript

**Tarea**

Lic. Miguel Ángel Salazar Santillán

Grupo: 007 Matrícula: 1732645

Diana Elizabeth Díaz Rodríguez

27/ 01/ 2017

The weird things of JavaScript

Comparison craziness

Let’s start by making x an array with 0 in it.

js> x=[0] [0]

It should equal itself, which is good.

js> x==x true

But it also equals not itself, which is not so good.

js> x==!x true

What about comparing a 3 element array and a string?

js> Array(3)==",," true

## Type twattery

The type of not a number is, well, a number. Sensible.

js> typeof NaN "number"

Well at least that means it ought to be equal to itself, right?

js> NaN==NaN false

So much for NaN. I wonder what the type of null is …

js> typeof null "object"

… but of course it is not actually an instanceof Object

js>null instanceof Object false

So, what is a string then?

js> "string" instanceof String false

Uh-huh.

## Bounds buffoonery

How about a really big number first of all. Wait. What?

js> 9999999999999999 10000000000000000

Hopefully small numbers make more sense.

js> 0.1+0.2==0.3 false

I guess not. I bet that max is still bigger than min though.

js> Math.max()>Math.min() false js> Math.max()<Math.min() true

Maybe not.

## Array assinity

Calling + on an empty array and an empty array gives the empty string. Who knew?

js> []+[] ""

How about an empty array and an empty object?

js> []+{} "[object Object]"

An empty object and an empty array gives, ummm, 0.

js> {}+[] 0

Whereas an empty object and an empty object is not a number. Which is true, I suppose.

js> {}+{} NaN

## Boolean balderdash

I’ve heard of that boolean arithmetic. Let’s give it a try.

js> true+true===2 true js> true-true===0 true

Ah. It looks like true is equal to one. I’ll just check.

js>true===1 false

## And one for luck

See if you can work out what this will evaluate to

js> (!+[]+[]+![]).length

### Calentar

Para entrar en el estado de ánimo que hablamos acerca de los objetos y que las propiedades del objeto o bien se puede acceder mediante el. (punto) o **[]** operador (entre corchetes) en el que el operador punto sólo acepta los nombres de variables de JavaScript válidos y los corchetes pueden tomar cualquier cadena:

**var** persona = {   
nombre: 'David',   
'& propiedad raro': 'YYCJS'   
}   
  
**var** prop = 'nombre';   
  
person.name // -> David  
 persona [ 'nombre'] // -> David  
 persona [ "nombre"] // -> David  
 persona [prop] // -> David  
  
 persona [ "propiedad extraños & '] // -> YYCJS   
  
// ERROR  
 persona. y rara propiedad

*No hay ninguna diferencia real entre el uso de comillas simples o dobles para cuerdas, excepto que el personaje que tendrá que escapar. Como siempre tiene sentido ser constante y no parece haber una preferencia general por comillas simples (menos teclas presionar en la mayoría de los teclados).*

### Parámetros de las funciones

En una función de argumentos es una variable especial tipo array que contiene todos los parámetros que recibe:

**la función** suma () {   
 **var** suma = 0;   
 **para** (var i = 0; i <arguments.length; i ++) {   
 sum + = argumentos [i];   
 }   
 **Devolver** suma;   
}

sum (1, 2, 3, 4); // -> 10

El uso de argumentos a menudo se utiliza para emular la sobrecarga de funciones cuando una función es llamada con un número diferente de parámetros.

argumentos*no es un verdadero arsenal, es sólo un objeto de matriz-como con una*longitud de*propiedad y los índices. Si desea utilizar métodos de arreglos como el*empuje*o*emergente*, se puede convertir utilizando [jQuery.toArray ()](http://api.jquery.com/toarray/" \t "_blank)o la siguiente manera:*

args var = Array.prototype.slice.call (argumentos, 0);

### Truthy- y falsyness

En JavaScript, valores que no sean reales booleanos pueden evaluar como verdadera (Truthy) o falsa (Falsy) en una expresión lógica. Los siguientes valores serán tratados como Falsy :

**falsa**   
**nula**   
**indefinido**  
 0   
'' // cadena vacía  
 person.undefinedProperty

Pueden, por ejemplo, ser evaluados en un caso de declaración:

**var** persona = {nombre: 'David'}

**si** (person.name) {   
 // Hacer cosas si existe la propiedad y no es Falsy  
 }

*Nota: Esto no funciona para los nombres de las variables no declaradas. Si desea ver si existe una variable (y no es indefinido) utiliza el operador typeof:*

**si** (typeof mivar! == 'no definido') {}

El resultado de una expresión lógica no siempre es un verdadero booleano. usando || (O) devolverá el primer valor Truthy o el último en la expresión. Esto a menudo se utiliza para asignar valores por defecto a un objeto o para los parámetros predeterminados de una función:

person.meetup = person.meetup || defaults.meetup || 'YYCJS';

**la función** suma (primero, segundo) {   
primera = primera || 0;   
segundo = segundo || 0;   
   
 **volver** segunda primera +;   
}

sum () // -> 0 en lugar de NaN  
 suma (1) // -> 1 en lugar de NaN

*Nota: Yo recomiendo única reasignación de variables de argumentos de la función de la configuración de ajustes y el uso de una variable local si va a cambiar de ninguna otra manera.*

### Igualdad

Sólo en los talones de Truthy y falsyness es comparación de igualdad. El == básica (igual) y = (no igual) los operadores sólo comparan el valor (sin el tipo), por ejemplo:

1 == 1 // -> true  
 1 == '1' // -> true  
 1 == 2 // -> false

// Ahora las cosas se ponen raras  
 '' == **false** // -> true  
 [] == **false** // -> true, [] no Falsy aunque   
**nulos** == **indefinido** // -> true

El === (triple igual) y == (igual no triple?) Los operadores comparan el valor! Y tipo:

1 === 1 // -> true  
 1 === 1 '' // -> false  
 1 === parseInt ( '1') // -> true  
 [] === **false** // -> false   
**nula** = == **indefinido** // -> false

Los objetos siempre se comparan por su referencia por lo que la expresión sólo será verdadera si en realidad es el mismo objeto:

var persona = {nombre: 'David'};   
var = otherperson persona;

persona === {nombre: 'David'} // -> false, creado nuevo objeto   
persona === otherperson // -> true, misma referencia

*Por razones de coherencia y para evitar la muy difícil de encontrar errores utilizan siempre === y! ==. A veces esto puede significar más código para escribir (como la conversión de cadenas si se espera un número) pero va a hacer las cosas mucho más fácil y más predecible.*

### Alcance

JavaScript sólo sabe ámbito de la función.Esto significa que bloquea o bucles for y while no introducen un nuevo ámbito de las variables con eficacia haciendo lo mismo que declarar todas las variables al comienzo de la función.

Funciones dentro de otras funciones, pueden acceder a las variables de su ámbito primario (cierre), pero no a la inversa.

**var** x = "exterior";   
**la función** count () {   
 **a** (var i = 0; i <10; i ++) {   
 **var** x = 'prueba';   
 **var** interno = function () {   
 **var** i = 42;   
 // -> X === 'prueba'   
 // -> i === 42   
 }   
 interior ();   
 // -> I === recuento actual   
 }   
}   
// -> x === 'exterior'

*A veces es posible ver todas las variables que se declaran en el comienzo de la función. Esto es principalmente una cuestión de estilo y no es necesario si los nombres de variables se mantienen descriptiva y las funciones se mantienen pequeñas.*

### Los cierres

Veamos el siguiente fragmento de código jQuery que crea diez botones, agrega un controlador de clic y los añade al cuerpo.

**para** ( **var** i = 0; i <10; i ++) {   
 **var** botón = $ ( '<botón> botón #' + i 'botón </>' +);   
 button.click (function () {   
 alerta (i);   
 });   
 . $ ( "cuerpo") (botón) adjuntar;   
}

Al hacer clic en cualquiera de los botones sin embargo no alertar a este número de botones, como cabe esperar, pero el número 10 para cada botón en su lugar.

La razón es nuevo ámbito de las variables. La función de controlador click se accede a la misma variable de su padre y ya que siempre es ejecutado después el bucle se hace, tendrá el mismo valor en todas partes.

El truco aquí es introducir un nuevo alcance a través de una función de contenedor y lo llaman con la cuenta actual. A continuación, se copia en una nueva variable de contador :

**para** ( **var** i = 0; i <10; i ++) {   
 **var** botón = $ ( '<botón> botón #' + i 'botón </>' +);   
 **var** envoltorio = function (contador) {   
 button.click (function () {   
 alerta (contador);   
 });   
 }   
 Envoltura (i);   
 . $ ( "cuerpo") (botón) adjuntar;   
}

*Sólo se copiarán los valores primitivos (String, Boolean, número). Si se pasa un objeto obtendrá la misma referencia de objeto que es importante saber si tiene la intención de modificar ese objeto.*

### Variables globales

En el entorno del navegador, las variables declaradas sin var se convierten automáticamente en global, en el nodo que será global para el módulo:

**la función** test () {   
 **var** local = 42;   
mundial = "global";   
}   
  
Test ();

Esta es probablemente siempre involuntaria. Una buena manera de evitar fugas accidentales de variable global en el nodo y los nuevos navegadores es permitir [ECMAScript 5 modo estricto](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/Strict_mode" \t "_blank) con

"Use strict ';

al comienzo de una función o un archivo JavaScript. Una fuga variable global luego tirar un ReferenceError.

Si necesita variable global, agregar y acceder a ellos a través del objeto global ( ventana en el navegador, mundial en el Nodo) de forma explícita:

**la función** test () {   
 **var** local = 42;   
 window.global = "global";   
}   
  
Test ();