T U E S D A Y , J U N E 2 1 , 2 0

Some Amazing Fact About JavaScript

At once weird and yet beautiful, it is surely the programming language that Pablo Picasso would have invented. This post about some secrets and amazing fact of **JavaScript**.This facts and secretes helps to developers who are curious about more advanced **JavaScript**. It is a collection of JavaScript's oddities and well-kept secrets. I am going to prove you that every each of them can be explained.

Data Types And Definitions

Null is an Object

Null is apparently an object, which, as far as contradictions go, is right up there with the best of them. Null? An object? "Surely, the definition of null is the **total absence of meaningful value**," you say. You'd be right. But that's the way it is. Here's the proof:

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

Despite this, null is not considered an instance *of* an object. (In case you didn't know, values in JavaScript are instances of base objects. So, every number is an instance of the *Number* object, every object is an instance of the *Object* object, and so on.) This brings us back to sanity, because if null is the absence of value, then it obviously can't be an instance of anything. Hence, the following evaluates to *false*:

```
alert(null instanceof Object);  //evaluates false
```

NaN is a Number.

You thought null being an object was ridiculous? Try dealing with the idea of *NaN* — "not a number" — being a number! Moreover, *NaN* is not considered equal to itself! Does your head hurt yet?

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

In fact *NaN* is **not equal to anything**. The only way to confirm that something is *NaN* is via the function *isNaN()*.

An Array With No Keys == False (About Truthy and Falsy)

Here's another much-loved JavaScript oddity:

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

To understand what's happening here, you need to understand the concepts of **truthy and falsy**. These are sort of true/false-lite, which will anger you somewhat if you majored in logic or philosophy.
I've read many explanations of what truthy and falsy are, and I feel the easiest one to understand is this: in JavaScript, **every non-boolean value has a built-in boolean flag** that is called on when the value is asked to behave like a boolean; like, for example, when you compare it to a boolean.
Because apples cannot be compared to pears, when JavaScript is asked to compare values of differing data types, it first "**coerces**" them into a common data type. *False*, *zero*, *null*, *undefined*, empty strings and *NaN* all end up becoming *false* — not permanently, just for the given expression. An example to the rescue:

```
var someVar = 0;
alert(someVar == false; //evaluates true
```

Here, we're attempting to compare the number *0* to the boolean *false*. Because these data types are incompatible, **JavaScript secretly coerces our variable into its truthy or falsy equivalent**, which in the case of *0* (as I said above) is falsy.
You may have noticed that I didn't include empty arrays in the list of falsies above. Empty arrays are curious things: they actually evaluate to truthy *but*, when compared against a boolean, behave like a falsy. Confused yet? With good cause. Another example perhaps?

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

To avoid coercion, you can use the **value and type comparison operator**, ===, (as opposed to ==, which compares only by value). So:

```
var someVar = 0;
```


S E A R C H
Search

C A T E G O
Apple
Database
DotNet
Google
Latest Tech NEWS
Mobiles

B L O G A
2013 (1)
2012 (5)
2011 (46)
November (8)
July (5)
June (33)
Forget '.com', are you ready for '.google', or '.b...
Extension method for IComparable
GUIDs (Globally Unique Identifiers) in .NET
Microsoft Free Books For SharePoint 2010
Windows Phone 7 Development
Apple Vs. Microsoft The History Of Computing (Info...
Windows 8 - Amazing OS
Nokia Window Phone Mango-Neowin
Techdays 2011 : Expression Blend for Silverlight D...
Tips and Tricks : Visual Studio 2010
NEWID() - Generate Randomly Sort Records - TSQL
SQL Server CE 3.5 Overview
Windows Phone 7.5 Mango technical preview
Hi-Tech ITO: Nokia N9 - Design, Social and Interac...
Apple : iOS 5 Features

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

Phew. As you've probably gathered, this is a broad topic, and I recommend reading up more on it—particularly on data coercion, which, while not uniquely a JavaScript concept, is nonetheless prominent in JavaScript.
I discuss the concept of truthy and falsy and data coercion more over here. And if you really want to sink your teeth into what happens internally when JavaScript is asked to compare two values, then check out section 11.9.3 of the ECMA-262 document specification.

Regular Expressions

Replace() Can Accept a Callback Function

This is one of JavaScript's best-kept secrets and arrived in v1.3. Most usages of *replace()* look something like this:

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

This is a simple replacement: a string, an asterisk. But what if we wanted more control over how and when our replacements take place? What if we wanted to replace only numbers under 30? This can't be achieved with regular expressions alone (they're all about strings, after all, not maths). We need to **jump into a callback function to evaluate each match**.

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

For every match made, JavaScript calls our function, passing the match into our match argument. Then, we return either the asterisk (if the number matched is under 30) or the match itself (i.e. no match should take place).

Regular Expressions: More Than Just Match and Replace

Many intermediate JavaScript developers get by just on *match* and *replace* with regular expressions. But JavaScript defines more methods than these two.
Of particular interest is *test()*, which works like *match* except that it doesn't return matches: **it simply confirms whether a pattern matches**. In this sense, it is computationally lighter.

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

The above looks for a pattern of three or more alphanumeric characters, and because the string *Hello* meets that requirement, we get *true*. We don't get the actual match, just the result.
Also of note is the *RegExp* object, by which you can create dynamic regular expressions, as opposed to static ones. The majority of regular expressions are declared using short form (i.e. enclosed in forward slashes, as we did above). That way, though, **you can't reference variables, so making dynamic patterns is impossible**. With *RegExp()*, though, you can.

```
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.');
```

Here, we're making a dynamic pattern based on the value of the argument *word*. The function returns the number of times that *word* appears in string as a word in its own right (i.e. not as a part of other words). So, our example returns *car* once, ignoring the *car* tokens in the words *Carl* and *card*. It forces this by checking for a word boundary (*b*) on either side of the word that we're looking for.
Because *RegExp* are specified as strings, not via forward-slash syntax, we can use variables in building the pattern. This also means, however, that we must double-escape any special characters, as we did with our word boundary character.

Functions And Scope

You Can Fake Scope

The scope in which something executes defines what variables are accessible. Free-standing JavaScript (i.e. JavaScript that does not run inside a function) operates within the global scope of the *window* object, to which everything has access; whereas local variables declared inside functions are accessible only within that function, not outside.

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

Here, our variable and function are both declared in the global scope (i.e. on *window*). Because this always points to the current scope, in this example it points to *window*. Therefore, the function looks for *window.animal*, which it finds. So far, so normal. But we can actually **con our function into thinking that it's running in a different scope**, regardless of its own natural scope. We do this by calling its built-in *call()* method, rather than the function itself:

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


Some Amazing Fact About JavaScript
OOPs Concept
Monthwise report With Pivot Query
Server Side Pagination Query
How to Add Google Plus one (+1) Button to Website/...
TSQL - Decimal to time
Divyang Panchasara: move ViewState to bottom of P...
IE9 Developer Tools: Network Tab
NULLIF (Transact-SQL)
As Operator in C#
Visual studio update on facebook
Web Standards Update for Visual Studio 2010 SP1
New in iOS 5: Split Keyboard
Important Query For DBA
Windows Phone Developer Tools 7.1 Beta
History Of MS SQL Server
Delete VS Truncate
Google Search by Image

K E Y W O R
DotNet C# Database TSQL MS Sql Server Apple Denali Google Mango Sql Server 2008 R2 Developer Tools IE 9 Microsft Phone 7.1 iOS 5 CSV Delete VS. Truncate JavaScript NULLIF Network

There was an error in this gadget

R E F E R E
DEBUG MODE.....
The call is ambiguous between the following methods in C#
12 hours ago

Divyang Panchasara
Reproduce Issues while debugging
1 year ago

Hi-Tech
Polynomial equations in excel
2 years ago

Hottest Technology Articles - Beyondrelational.com
Script to create a Foreign key in SQL Server - Paresh Prajapati
2 years ago

Journey to SQLAuthority
SQL Authority News – Secret Tool Box of Successful Bloggers: 52 Tips to Build a High Traffic Top Ranking Blog
23 hours ago

Web Design Trendz & Tutorial for Beginners
Free Raster to Vector
3 years ago

S U B S C R
Posts

Here, our function runs not on *window* but on *myObj* —specified as the first argument of the call method. Essentially, *call()* pretends that our function is a method of *myObj* (if this doesn't make sense, you might want to read up on JavaScript's system of prototypal inheritance). Note also that any arguments we pass to *call()* after the first will be passed on to our function—hence we're passing in *lovely* as our *adjective* argument.
I've heard JavaScript developers say that they've gone years without ever needing to use this, not least because good code design ensures that you don't need this smoke and mirrors. Nonetheless, it's certainly an interesting one.
As an aside, *apply()* does the same job as *call()*, except that arguments to the function are specified as an array, rather than as individual arguments. So, the above example using *apply()* would look like this:

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

- **Functions Can Execute Themselves**

  There's no denying it:

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

The syntax is simple enough: we declare a function and immediately call it just as we call other functions, with *()* syntax. You might wonder why we would do this. It seems like a contradiction in terms: a function normally contains code that we want to execute later, not now, otherwise we wouldn't have put the code in a function.
One good use of self-executing functions (SEFs) is to **bind the current values of variables** for use inside delayed code, such as callbacks to events, timeouts and intervals. Here is the problem:

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

Newbies in forums invariably ask why the *alert* in the *timeout* says *goodbye* and not *hello*. The answer is that the *timeout* callback function is precisely that—a callback—so it doesn't evaluate the value of *someVar* until it runs. And by then, *someVar* has long since been overwritten by *goodbye*.
SEFs provide a solution to this problem. Instead of specifying the timeout callback implicitly as we do above, we return it from an SEF, into which we pass the current value of *someVar* as arguments. Effectively, this means **we pass in and isolate the current value of *someVar*, protecting it from whatever happens to the actual variable *someVar* thereafter**. This is like taking a photo of a car before you respray it; the photo will not update with the resprayed color; it will forever show the color of the car at the time the photo was taken.

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

This time, it alerts *hello*, as desired, because it is alerting the isolated version of *someVar* (i.e. the function argument, *not* the outer variable).

The Browser

- **Firefox Reads and Returns Colors in RGB, Not Hex**

  I've never really understood why Mozilla does this. Surely it realizes that anyone interrogating computed colors via JavaScript is interested in hex format and not RGB. To clarify, here's an example:

```
<!--
#somePara { color: #f90; }
-->

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>
```

While most browsers will alert *ff9900*, Firefox returns *rgb(255, 153, 0)*, the RGB equivalent. Plenty of JavaScript functions are out there for converting RGB to hex.
Note that when I say computed color, I'm referring to the current color, **regardless of how it is applied to the element**. Compare this to style, which reads only style properties that were implicitly set in an element's style attribute. Also, as you'll have noticed in the example above, IE has a different method of detecting computed styles from other browsers.
As an aside, jQuery's *css()* method encompasses this sort of computed detection, and it returns styles however they were applied to an element: implicitly or through inheritance or whatever. Therefore, you would relatively rarely need the native *getComputedStyle* and *currentStyle*.

Miscellaneous

- **Precision**

JavaScript is converting all numbers to binary format (even floats) and displaying them in proper form. 0.1 in binary is 0.0 0011 0011 0011 0011 0011 … and so on. Because of that we cannot write 0.1 without loosing precision. And that's why in JavaScript 0.1 + 0.2 = 0.30000000000000004.

```
0.1 + 0.2 == 0.3 // false
```

The same thing happens with

```
11111111111111113 == 11111111111111111 // true
```

Except that in that case highest bits are truncated and both *11111111111111111* and *11111111111111113* become *11111111111111112*

- **Strings and Numbers**

  That will be long story.
  We can see many times ToNumber function in above algorithm. There's also long paragraph in ECMA-262 called ToNumber Applied to the String Type. Let me cite part of it:

  > The conversion of a string to a number value is similar overall to the determination of the number value for a numeric literal (see 7.8.3), but some of the details are different, so the process for converting a string numeric literal to a value of Number type is given here in full.

  And actually, result of ToNumber("01234") was equal to literal 01234.
  Soon after the problem was noticed: Number Literal could not be octal number. So we might write 0xFF and it was equal to 255, but we could not write 01234 (which is 668 in decimal).
  They decided to write Annex, so since then *01234 == 668* and *0001234 == 668*. However, new notification says that if there is 8 or 9 in Number Literal, leading zeros are stripped and number is treated as decimal.

```
08 == 010     // true!
01234 == 0668 // true
```

  Moreover they **accidentally** forgot about ToNumber algorithm, so ToNumber("01234") still is 1234. And finnaly that's why:

```
0x123 == "0x123" // true
0123 == "0123"   // false!
01234 == "0668"  // true!
```

- **Undefined Can Be Defined**

  OK, let's end with a silly, rather inconsequential one. Strange as it might sound, *undefined* is not actually a reserved word in JavaScript, even though it has a special meaning and is the only way to determine whether a variable is undefined. So:

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

  So far, so normal. But:

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

  You can also check Mozilla's list of all reserved words in JavaScript for future reference.

Reference

- *Ten Oddities And Secrets About JavaScript*
- *JavaScript oddities explained. Comparing*

N O C O M M E N T S :

P O S T A C O M M E N T

Enter your comment...

Comment as: Select profile...

Publish Preview

L I N K S T O T H I S P O S T

Create a Link

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

R E L A T E D P O S T S

- **Some Amazing Fact About JavaScript**