Introduction

This article is an introduction to the weird side of JavaScript, and it definitely has a weird side! Software developers who usually write code in another language will find a lot of intriguing "features" when they begin to write code in the world's most widely used language. Hopefully, even seasoned JavaScript developers will find some new gotchas in this article that they can watch out for. Enjoy!

Contents

Functions and Operators

Double-equals

The == operator performs a comparison with type coercion. This means that you can compare objects of two different types and it will attempt to convert them to the same type before performing a comparison. For example:

Hide Copy Code

```
"1" == 1 //true
```

However, this is very often misleading, and never needed. In the case above, you can convert the `String` to a`Number` and use the type-sensitive triple-equals:

Hide Copy Code

```
Number("1") === 1; //true
```

Or, better still, ensure your operands are of the correct type in the first place.

Due to its type-coercing behaviour, double-equals frequently breaks the transitivity rule, which is a little bit scary:

Hide Copy Code

```
"" == 0 //true - empty string is coerced to Number 0.
0 == "0" //true - Number 0 is coerced to String "0"
"" == "0" //false - operands are both String so no coercion is done.
```

The above comparisons will all yield `false` when triple-equals is used.

parseInt doesn't assume base-10

If you omit the second parameter in a call to `parseInt`, then the base will be determined according to the following rules:

- By default, assume a radix 10.
- If the number begins with **0x** then assume radix 16.
- If the number begins with **0** then assume radix 8.

The common mistake is to allow a user-input or suchlike which begins with a **0**. Radix 8 (octal) is then used and we see this effect:

Hide Copy Code

```
parseInt("8"); //8
parseInt("08"); //0
```

Therefore, always include the second parameter:

Hide Copy Code

```
parseInt("8", 10); //8
parseInt("08", 10); //8
```

ECMAScript5 side note: ES5 no longer includes the radix-8 assumption. Additionally, omission of the second parameter helpfully raises a JSLint warning.

String replace

The string `replace` function only replaces the first match, not all matches as you may expect.

Hide Copy Code

```
"bob".replace("b", "x"); // "xob"
"bob".replace(/b/, "x"); // "xob" (regular expression version)
```

To replace all matches, you must use a Regular Expression, and add the global modifier to it:

Hide Copy Code

```
"bob".replace(/b/g, "x"); // "xox"
"bob".replace(new RegExp("b", "g"), "x"); // "xox" (alternate explicit RegExp)
```

The global modifier ensures that the replacement does not stop after the first match.

The "+" Operator Both Adds and Concatenates

PHP, another loosely typed language, has the '.' operator for `string` concatenation. JavaScript does not - so "`a + b`" always results in concatenation when either of the operands is a `string`. This might catch you out if you're trying to add a number to, say, the contents of an input element (which will be a `string`), so you need to first cast to `Number`:

Hide Copy Code

```
1 + document.getElementById("inputElem").value; // Concatenates
1 + Number(document.getElementById("inputElem").value); // Adds
```

Note that the subtract operator attempts to convert the operands to `Number`:

Hide Copy Code

```
"3" - "1"; // 2
```

Although if you're trying to subtract one `string` from another, then there's probably something wrong with your logic.

Sometimes people try to convert a `Number` to a `string` by concatenating with the empty `string`:

Hide Copy Code

```
3 + ""; // "3"
```

But that's not very nice, so use `String(3)` instead.

typeof

Returns the type of an instance of a fundamental type. `Array` is actually not a fundamental type, so the `typeof`an `Array` object is `Object`:

Hide Copy Code

```
typeof {} === "object" //true
typeof "" === "string" //true
typeof [] === "array"; //false
```

You will get the same result (`typeof = "object"`) when you use this operator against instances of your own objects.

As a side note, "`typeof null`" yields "`object`", which is a bit weird.

instanceof

Returns whether the object (or an object in its prototype chain) was constructed using the specified constructor, which is useful when attempting to check the type of one of your own defined object types. However, it is pretty misleading if you create an instance of one of the built-in types using literal syntax:

Hide Copy Code

```
"hello" instanceof String; //false
new String("hello") instanceof String; //true
```

Since `Array` isn't really one of the built-in types (it just pretends to be - hence `typeof` doesn't work as expected), `instanceof` does work as expected:

Hide Copy Code

```
["item1", "item2"] instanceof Array;  //true
new Array("item1", "item2") instanceof Array;  //true
```

Phew! So in summary, if you want to test the type of a `Boolean`, `String`, `Number`, or `Function`, you can use`typeof`. For anything else, you can use `instanceof`.

Oh, one more thing. Within a function, there is a predefined variable, "`arguments`", which gives an array of arguments passed to the function. However, it's not really an array, it's an array-like object with a `length`property, and properties from `0 - length`. Pretty strange...but you can convert it to a real array using this common trick:

Hide Copy Code

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

The same goes for `NodeList` objects returned by DOM calls such as `getElementsByTagName` - they can also be converted to proper arrays using the above code.

eval

`eval` interprets a `string` as code but its use is generally frowned upon. It's slow - when JavaScript is loaded into the browser, it gets compiled into native code; however, every time an `eval` statement is reached during execution, the compilation engine has to be started up all over again, and that is quite expensive. It looks quite ugly, too, because in most cases it is misused. Additionally, the code being `eval`'d is executed in the current scope, so it can modify local variables and add stuff to your scope which may be unintended.

Parsing JSON is a common usage; normally people will use "`var obj = eval(jsonText);`", however almost all browsers now support the native JSON object which you can use instead:

"`var obj = JSON.parse(jsonText);`". There is also the `JSON.stringify` function for the opposite use. Even better, you can use the `jQuery.parseJSON` function which will always work.

The `setTimeout` and `setInterval` functions can take a `string` as its first parameter which will be interpreted, so that shouldn't be done either. Use an actual function as that parameter.

Finally, the `Function` constructor is much the same as `eval` - the only difference being that it will be executed in the global context.

with

The `with` statement gives you a shorthand for accessing the properties of an object, and there are conflicting views on whether it should be used. Douglas Crockford doesn't like it. John Resig finds a number of clever uses for it in his book, but also concedes that it has a performance hit and can be a little confusing. Looking at a `with`block in isolation, it isn't possible to tell exactly what is happening. For example:

Hide Copy Code

```
with (obj) {
    bob = "mmm";
    eric = 123;
}
```

Did I just modify a local variable called "`bob`", or did I set `obj.bob`? Well, if `obj.bob` is already defined, then it is reset to "mmm". Otherwise, if there is another `bob` variable in scope, it is changed. Otherwise the global variable`bob` is set. In the end, it is just clearer to write exactly what you mean to do.

Hide Copy Code

```
obj.bob = "mmm";
obj.eric = 123;
```

ECMAScript5 side note: ES5 strict mode will not support the `with` statement.

Math.min and Math.max

These are two utility functions that give you the max or min of the arguments. However, if you use them without any arguments, you get:

Hide Copy Code

```
Math.max(); // -Infinity
Math.min(); // Infinity
```

It does make sense really, since `Math.max()` is returning the largest of an empty list of numbers. But if you're doing that, you're probably after `Number.MAX_VALUE` or `Number.MIN_VALUE`.

Types and Constructors

Constructing Built-in Types with the 'new' Keyword

JavaScript has the types `Object`, `Array`, `Boolean`, `Number`, `String`, and `Function`. Each has its own literal syntax and so the explicit constructor is never required.

Explicit (bad)	Literal (good)
`var a = new Object();` `a.greet = "hello";`	`var a = { greet: "hello" };`
`var b = new Boolean(true);`	`var b = true;`
`var c = new Array("one", "two");`	`var c = ["one", "two"];`
`var d = new String("hello");`	`var d = "hello"`
`var e = new Function("greeting",` `"alert(greeting);");`	`var e = function(greeting) {` `alert(greeting); };`

However, if you use the `new` keyword to construct one of these types, what you actually get is an object of type`Object` that inherits the prototype of the type you want to construct (the exception is `Function`). So although you can construct a `Number` using the `new` keyword, it'll be of type `Object`;

Hide Copy Code

```
typeof new Number(123); // "object"
typeof Number(123); // "number"
typeof 123; // "number"
```

The third option, the literal syntax, should always be used when constructing one of these types to avoid any confusion.

Constructing Anything Else Without the 'new' Keyword

If you write your own constructor function and forget to include the `new` keyword, then bad things happen:

Hide Copy Code

```
var Car = function(colour) {
    this.colour = colour;
};

var aCar = new Car("blue");
console.log(aCar.colour); // "blue"

var bCar = Car("blue");
console.log(bCar.colour); // error
console.log(window.colour); //"blue"
```

Calling a function with the `new` keyword creates a new object and then calls the function with that new object as its context. The object is then returned. Conversely, invoking a function without '`new`' will result in the context being the global object if that function is not invoked on an object (which it won't be anyway if it's used as a constructor!)

The danger in accidentally forgetting to include '`new`' means that a number of alternative object-constructing patterns have emerged that completely remove the requirement for this keyword, although that's beyond the scope of this article, so I suggest some further reading!

There is no Integer

Numerical calculations are comparatively slow because there is no Integer type, only `Number` - and `Number` is an IEEE floating point double-precision (64 bit) type. This means that `Number` exhibits the floating point rounding error:

Hide Copy Code

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

Since there's no distinction between integers and floats, unlike C# or Java, this is true:

Hide Copy Code

```
0.0 === 0; //true
```

Finally a `Number` puzzle. How can you achieve the following?

Hide Copy Code

```
a === b; //true
1/a === 1/b; //false
```

The answer is that the specification of `Number` allows for both positive and negative zero values. Positive zero equals negative zero, but positive infinity does not equal negative infinity:

Hide Copy Code

```
var a = 0 * 1; // This simple sum gives 0
var b = 0 * -1; // This simple sum gives -0 (you could also just
                // do "b = -0" but why would you do that?)
a === b; //true: 0 equals -0
1/a === 1/b; //false: Infinity does not equal -Infinity
```

Scope

No Block Scope

As you may have noticed in the previous point, there is no concept of block scoping, only functional scoping. Try the following piece of code:

Hide Copy Code

```
for(var i=0; i<10; i++) {
    console.log(i);
}
var i;
console.log(i); // 10
```

When `i` is declared in the `for` loop, it remains in scope after the loop exits. So the final call to `console.log` will output `10`. There is a JSLint warning which aims to avoid this confusion: forcing all variables to be declared at the start of a function so that it is obvious what is happening.

It is possible to create a scope by writing an immediately-executing function:

Hide Copy Code

```
(function (){
    for(var i=0; i<10; i++) {
        console.log(i);
    }
}());
var i;
console.log(i); // undefined
```

There is another twist which becomes apparent when you declare a `var` *before* the inner function, and re-declare it *within* that function. Look at this example:

Hide Copy Code

```
var x = 3;
(function (){
    console.log(x + 2); // 5
    x = 0; //No var declaration
}());
```

However, if you re-declare `x` as a `var` in the inner function, there is strange behaviour:

Hide Copy Code

```
var x = 3;
(function (){
    console.log(x + 2); //NaN - x is not defined
    var x = 0; //var declaration
}());
```

This is because "`x`" is redefined inside the function. This means that the interpreter moves the `var` statements to the top of the function and we end up executing this:

Hide Copy Code

```
var x = 3;
(function (){
    var x;
    console.log(x + 2); //NaN - x is not defined
    x = 0;
}());
```

Which makes perfect sense!

Global Variables

JavaScript has a global scope which you should use sparingly by namespacing your code. Global variables add a performance hit to your application, since when you access them, the runtime has

to step up through each scope until it finds them. They can be accessed and modified either intentionally or by accident by your own code and other libraries. This leads to another, more serious issue - cross-site scripting. If a bad person figures out how to execute some code on your page, then they can also easily interfere with your application itself by modifying global variables. Inexperienced developers constantly add variables to the global scope by accident, and there are a few examples of how that happens throughout this article.

I have seen the following code which attempts to declare two local variables with the same value:

Hide Copy Code

```
var a = b = 3;
```

This correctly results in `a = 3` and `b = 3`, however `a` is in local scope and `b` is in global scope. "`b = 3`" is executed first, and the result of that global operation, 3, is then assigned to the local var `a`.

This code has the desired effect of declaring two local variables with value `3`:

Hide Copy Code

```
var a = 3,
b = a;
```

'this' and Inner Functions

The '`this`' keyword always refers to the object on which the current function was called. However, if the function is not invoked on an object, such as is the case with inner functions, then '`this`' is set to the global object (window).

Hide Copy Code

```
var obj = {
    doSomething: function () {
        var a = "bob";
        console.log(this); // obj
        (function () {
            console.log(this); // window - "this" is reset
            console.log(a); // "bob" - still in scope
        }());
    }
};
obj.doSomething();
```

However, we have closures in JavaScript so we can just take a reference to '`this`' if we want to refer to it in the inner function.

ECMAScript5 side note: ES5 strict mode introduces a change to this functionality whereby '`this`' will be undefined instead of being set to the global object.

Miscellaneous

The Absence of data: 'null' and 'undefined'

There are two object states that represent the lack of a value, `null` and `undefined`. This is pretty confusing for a programmer coming from another language like C#. You might expect the following to be `true`:

Hide Copy Code

```
var a;
a === null; //false
a === undefined; //true
```

'a' is in fact undefined (although if you do a double-equals comparison with `null` then it will be `true`, but that is just another mistake that results in the code "seeming" to work).

If you want to check that a variable actually has a value, and also follow the rule of never using double-equals, you need to do the following:

Hide Copy Code

```
if(a !== null && a !== undefined) {
    ...
}
```

Or, you could do this, which is exactly the same thing (although JSLint will unfortunately complain at you):

Hide Copy Code

```
if (a != null) {
    ...
}
```

"Aha!", you might say. `null` and `undefined` are both falsy (i.e., coerced to `false`) so you can do this:

Hide Copy Code

```
if(a) {
    ...
}
```

But, of course, zero is also falsy, and so is the empty `string`. If one of those is a valid value for 'a', then you will have to use the former, more verbose option. The shorter option can always be used for objects, arrays, and booleans.

Redefining Undefined

That's right, you can redefine `undefined`, as it isn't a reserved word:

Hide Copy Code

```
undefined = "surprise!";
```

However, you can get it back by assigning an undefined variable or using the "`void`" operator (which is otherwise pretty useless):

Hide Copy Code

```
undefined = void 0;
```

...and that's why the first line of the jQuery library is:

Hide Copy Code

```
(function ( window, undefined ) {
    ... // jQuery library!
}(window));
```

That function is called with a single parameter ensuring that the second parameter, `undefined`, is in fact undefined.

By the way, you can't redefine `null` - but you can redefine `NaN`, `Infinity`, and the constructor functions for the built-in types. Try this:

Hide Copy Code

```
Array = function (){ alert("hello!"); }
var a = new Array();
```

Of course, you should be using the literal syntax to declare an `Array` anyway.

Optional Semicolons

Semicolons are optional so that it's easier for beginners to write, but it's pretty yucky to leave them out and doesn't really make anything easier for anyone. The result is that when the interpreter gets an error, it backtracks and tries to guess where the semicolon is supposed to go.

Here is a classic example, courtesy of Douglas Crockford:

Hide Copy Code

```
return
{
    a: "hello"
};
```

The above code does not return an object, it returns `undefined` - and no error is thrown. A semicolon is automatically inserted immediately after the `return` statement. The remainder of the code is perfectly valid, even if it does not do anything, and this should be evidence enough that in JavaScript, an opening curly brace should go at the end of the current line instead of a new line. It isn't just a matter of style! This code returns an object with a single property "`a`":

Hide Copy Code

```
return {
    a: "hello"
};
```

NaN

The type of `NaN` (Not a Number) is... `Number`.

Hide Copy Code

```
typeof NaN === "number" //true
```

Additionally, `NaN` compared to anything is false:

Hide Copy Code

```
NaN === NaN; // false
```

Since you can't do a `NaN` comparison, the only way to test whether a number is equal to `NaN` is with the helper function `isNaN`.

As a side note, we also have the helper function `isFinite`, which returns `false` when the argument is `NaN` or`Infinity`.

The 'arguments' Object

Within a function, you can reference the `arguments` object to retrieve the list of arguments. The first gotcha is that this object is not an `Array` but an array-like object (which is an object with a `length` property, and a value at `[length-1]`). To convert to a proper array, you can use the array `splice` function to create an array from the properties in `arguments`:

Hide Copy Code

```
(function(){
console.log(arguments instanceof Array); // false
var argsArray = Array.prototype.slice.call(arguments);
console.log(argsArray instanceof Array); // true
}());
```

The second gotcha is that when a function has explicit arguments in its signature, they can be reassigned and the`arguments` object will also be changed. This indicates that the `arguments` object points to the variables themselves as opposed to their values; you can't rely on `arguments` to give you their original values.

Hide Copy Code

```
(function(a){
    alert(arguments[0]); //1
    a = 2;
    alert(arguments[0]); //2
}(1));
```

In ES5 strict mode, `arguments` will definitely point to the original input argument values, as opposed to their current values.

The End!

So that concludes my list of gotchas. I am sure there are more, and I look forward to seeing lots of insightful comments!

P.S. - I do actually *like* JavaScript.

References

- http://www.scottlogic.co.uk/blog/luke-page/
- http://www.felixcrux.com/posts/douglas-crockford-talk-waterloo/
- http://dev.opera.com/articles/view/efficient-javascript/?page=2
- http://www.amazon.co.uk/JavaScript-Good-Parts-Douglas-Crockford/dp/0596517742
- https://developer.mozilla.org/en/JavaScript/Reference/Operators/Special/typeof
- http://ejohn.org/blog/ecmascript-5-strict-mode-json-and-more/