Javascript Tips and Tricks (3)

Adrian Kosmaczewski

2007-08-05

How to organize code in "namespaces"

When you use lots of libraries in your code, you can easily pick up a function name that corresponds to a pre-existing name in some library that you might have included. To avoid that, you should create namespaces that encapsulate the code of your application:

```
var net = {
    kosmaczewski: {
        adrian: {
            blog: {
                articles: {},
                images: {},
                snippents: {},
                tutorials: {},
                rants: {}
            }
        }
    }
};

// Shortcut (for performance purposes)
var blog = net.kosmaczewski.adrian.blog;
```

Then, you can start adding members (functions, classes, variables, etc) to that namespace:

```
blog.rants.NewRant = function() {
    this.whatever = 'value';
    // code here...
}

blog.images.takePhoto = function() {
    return 'a nice picture';
}
```

```
blog.articles.numberOfPosts = 700;
```

The following image shows how Firebug displays the objects added in their own namespaces, and also how the Ext and the Yahoo! code appear in their own namespaces:

Create Objects and Arrays the Easy Way

To create objects and arrays in JavaScript, you can of course use the constructor+methods syntax:

```
var obj = new Object();
var arr = new Array();

obj.prop = "value";
obj.method = function() {
    alert('method');
};

arr.push(23);
arr.push("yeah");
arr.push(obj);

arr[2].method();
alert(arr);
```

While the above syntax is OK, many JavaScript interpreters can handle the following, completely 100% equivalent version, much faster and more "JavaScript-like" (by this I mean that you are more likely to find this in JavaScript libraries):

```
var obj = {
    prop: "value",
    method: function() {
        alert('method');
    }
};
var arr = [23, "yeah", obj];

arr[2].method();
alert(arr);
```

Create a Singleton Object

If you need to create a "singleton", yet complex object, do not fall in the "classical" way of doing things: do not create a class and then instantiate just one instance!. Since JavaScript objects can be created on the fly, you can use object literals for that.

However, if you need to create just one object, with a complex structure, you can use the following trick:

```
var Singleton = function() {
    var privateValue = "private value";

    return {
        prop: "value",

        method: function() {
            alert(privateValue);
        }
    };
}();

Singleton.method();
```

The trick is in the line 11 of the example above:

```
}();
```

Do you see the parentheses after the closing bracket and before the semicolon? Well, this triggers the execution of the function, which "returns" a literal object with methods and properties, and which can reference private members (since they are closures).

This pattern is very common in the Ext Framework!

Scheduling Function Execution

Adding a method to the "Function" class, you can schedule its execution a couple of milliseconds in the future, encapsulating the `Window.setTimeout()` method:

```
Function.prototype.schedule = function(msec) {
    this.timeout = setTimeout(this, msec);
}

Function.prototype.cancelSchedule = function() {
    clearTimeout(this.timeout);
}

function doSomething() {
    alert('doSomething');
}

doSomething.schedule(5000);
```

The Ext framework has a method that does exactly this, called `defer()`. Very handy!

Concatenating Strings

If you have to concatenate strings, avoid using the `+` operator whenever possible; the Array class can be used as a Java `StringBuffer` or a .NET `StringBuilder`, as follows:

```
var s = ["a", "long", "array", "of", "strings"];
s.push("is");
s.push("here");
document.write(s.join("<br>"));
```

The above code will display the following output on the web page:

```
a
long
array
of
strings
is
here
```

The use of the "join()" method is considered as slightly faster than using the "+" operator, which implies much more object copies in memory, which in turn triggers the garbage collector much more often.

Iterating over Arrays

When you operate on array objects, you usually end up writing code like this:

```
function operate(obj) {
    alert(obj);
}

var arr = [54, 25, 68];
for (var i = 0; i < arr.length; ++i) {
    operate(arr[i]);
}
```

But things do not have to be that awful all the time:

```
Array.prototype.each = function(func) {
    for (var i = 0; i < this.length; ++i) {
        func(this[i]);
    }
}
```

```
function operate(obj) {
    alert(obj);
}

var arr = [54, 25, 68];
arr.each(operate);
```

Now you have code that's much easier to read!

Using `toString()` for Reflection

Let's suppose that you have the code written above

```
function Thing() {
    var privateField = "PRIVATE";

    var privateMethod = function() {
        alert('Private method');
    }

    return {
        publicField: "PUBLIC",

        publicMethod: function() {
            alert('Public method');
        }
    };
}

// Creating a new instance of "Thing"
var thingy = new Thing();

// You want to see what's inside, right?
alert(thingy);
```

The last instruction above shows a laconic `[object Object]` which does not tell much about what's inside your object… Actually, the `alert()` function, when applied to any JavaScript object, will call the "toString()" method to its parameter: so try adding a `toString()` to your objects instead:

```
function Thing() {
    var privateField = "PRIVATE";
    var self = this;

    var privateMethod = function() {
        alert('Private method');
    }
```

```
    return {
        publicField: "PUBLIC",

        publicMethod: function() {
            alert('Public method');
        },

        toString: function() {
            var re = /function (.*)\(\) {/g;
            var a = self.constructor.toString().split("\n")[0];
            var cls = "Class " + re.exec(a)[1];
            var s = [cls, "", "Public API:"];
            for (var item in this) {
                s.push(item);
            }
            return s.join("\n");
        }
    };
}

// Creating a new instance of "Thing"
var thingy = new Thing();

// You'll get a very rudimentary reflection output
alert(thingy);
```

With this code, you'll get this output in a dialog box:

```
Class Thing

Public API:
publicField
publicMethod
```

Easy Code Injection

This code allows you to inject arbitrary code around any function, like if you were doing some cheapo AOP:

```
Function.prototype.wrap = function(before, after) {
    before();
    this();
    after();
};

function doBefore() {
    alert('do before');
```

```
}

function doAfter() {
    alert('do after');
}

function test() {
    alert('inside test');
}

// test();
test.wrap(doBefore, doAfter);
```

Do not hesitate to submit your own snippets or tricks! I hope this is useful to you.