Javascript Tips and Tricks (1)

Adrian Kosmaczewski

2007-08-03

As I said yesterday, JavaScript is the world's most misunderstood language, which means that you must unlearn what you have learned. However complicated it might seem at first, it is quite easy to write and understand the most complex of JavaScript codes with just some examples.

Just a few observations before starting:

- Semicolons (`;`) are not mandatory, but **strongly recommended!**
- You can create strings using either 'apostrophes' or "quotes". "You can also 'mix them' as you want", but 'always keeping the "order" when using them'.
- Always use the "var" keyword when defining variables. Otherwise, the variables will be created on the "Global Object" of JavaScript, and this is a bad thing(TM): variables created in the "Global Object" **do not get garbage collected!**

This page provides an excellent complement of information to know JavaScript better, as well as the Wikipedia page.

Some Tools

To work properly while programming in JavaScript, it is strongly recommended to use the following tools:

- Firefox
- Firebug
- Web Developer Extension
- Internet Explorer Developer Toolbar

The last one is just if you **must** Internet Exploder, which is a pain in the *** anyway.

Object "Literals"

First of all, every object in JavaScript is a map, and you can access properties and methods using either the **dot.syntax** or the [**"array"**] syntax:

```
var obj = {
    age: 42,
    "first and last name": "John Smith",     // yes, you can do that
    address: {
        street: "32 Kingston St.",
        city: "Springfield",
        zip: 12345
    },
    greet: function() {
        alert('hello! my name is ' + this["first and last name"] +
                ' and my age is ' + this.age.toString());
    }
};

// shows "John Smith"
alert(obj["first and last name"]);

// shows "true" ('===' is the identity operator)
alert(obj.age === obj["age"]);

// shows "object"
alert(typeof obj);
obj.greet();
```

Since the **dot.syntax** and the [**"array"**] syntaxes are equivalent, you must by now imagine that every "dot" in your code means a search into a dictionary (or literal object). **So, the less "dots" you use when calling an object, function or expression, the faster it is!**

To achieve this, use shortcuts:

```
var a = {
    very: {
        interesting: {
            JavaScript: {
                object: {
                    reference: "just an example!"
                }
            }
        }
    }
};

var shortcut = a.very.interesting.JavaScript.object.reference;
alert(shortcut);
```

JavaScript Base Classes

The following are the 7 core JavaScript classes that are part of the ECMA standard:

- `Object` (root class, like in Java)
- `String`
- `Number`
- `Array`
- `Date`
- `RegExp`
- `Function`

For a handy reference of the core JavaScript API, download and print this "cheat sheet" (local copy here: JavaScript cheat sheet).

It is important to know that, since JavaScript is a dynamic language, and that Functions are first-class objects, you can **add methods to a class on the fly**:

```
String.prototype.doSomething = function() {
    alert(this);
}

"hello!".doSomething();
```

The classes enumerated above are always present, in all JavaScript implementations (Adobe Flash ActionScript, Microsoft JScript, ECMAScript, etc).

When JavaScript runs in a web browser, other classes and objects are added, like Window, Document and others. These classes are part of the **DOM** (Document Object Model) and are browser specific (and based on W3C standards).

Functions

Functions are the basic block in JavaScript. You use them everywhere, you can pass them as parameters, attach them as event handlers, override them, delete them, etc. They can be anonymous or not:

```
function nonAnonymous() {
    alert('non anonymous!');
}

var nonAnonymousToo = function() {
    alert('non anonymous too!');
}

// This is the syntax for adding event handlers in Ext
field.on("click", function() {
    alert('now this is an anonymous function!');
});
```

You can also nest functions into functions, which is what they call **closures** in Lisp and other functional languages. Internal functions can access the variables

created in the stack of their "parent" function:

```
function example() {
    var privateVar = 'a private var';

    function execute() {
        function go() {
            alert(privateVar);
        }
        go();
    }
    execute();
}

example();
```