Javascript Tips and Tricks (2)

Adrian Kosmaczewski

2007-08-04

Object-Oriented Programming in JavaScript

Functions are also used to represent classes when doing object-oriented JavaScript. There are several possible ways to write object-oriented JavaScript code, but they all turn around the concept of the `Function` class:

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

    // See below for more explanations about "this"   :)
    var self = this;

    var privateMethod = function() {
        alert('Private methods can be called from public methods');
        self.anotherPublicMethod();
    }

    this.publicField = "PUBLIC";

    this.publicMethod = function() {
        privateMethod();
        alert('From the public method;\nthis is a public value: '
                    + this.publicField
                    + '\nand this is a private value: '
                    + privateField);
    };

    this.anotherPublicMethod = function() {
        alert('You need a trick to call this
                        from a private method!');
    };
}

// Creating a new instance of "Thing"
```

```
var thingy = new Thing();
thingy.publicMethod();
// you can also call thingy["publicMethod"]();
```

As you can see, "methods" are just function objects attached as any other property. You can attach any other function to this property, changing the behavior of your class on the fly.

The "self" trick

As you can see in the above code, there is a variable called "self" (it could have any name) that is equal to "this". This is a trick that allows private methods to access public methods, and you will see it in many JavaScript libraries. The problem can be summarized as follows: you cannot write the following

```
var privateMethod = function() {
    alert('Private methods can be called from public methods');
    this.anotherPublicMethod();
}
```

because `this` in that context means the `privateMethod()` function. What we want is the "this" that "points to" the current "Thing()" instance. Yes, it's kinda complex, but it works perfectly well, because the `privateMethod()` function is a closure, and can access the stack variables of the `Thing()` function. Since `self` points to the right object, you can now call the public method that you want.

In summary, "this" points always to the immediately containing "function" object where you are located.

More ways to do the same thing

Another way to write the above code would be like this, but it has the drawback that you cannot access the "private" members, since you are attaching the public members to the "prototype" of the function, and as such, you are outside of the main context of the function Thing():

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

    var privateMethod = function() {
        alert('Private methods can be called from public methods');
    }
}

Thing.prototype.publicField = "PUBLIC";

Thing.prototype.publicMethod = function() {
```

```
    // privateMethod(); cannot be called here!
    // We're outside of the "Thing()" context
    alert('From the public method;\nthis is a public value: '
            + this.publicField
            + '\nbut you cannot access a private value!');
};

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

// You can also call the public method as follows:
thingy["publicMethod"]();
```

The above syntax can also be written as follows:

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

    var privateMethod = function() {
        alert('Whatever');
    }
}

function Thing_publicMethod() {
    alert('Implementation');
};

function Thing_anotherPublicMethod() {
    alert('More implementation');
};

// "Header file" with the interfaces, all together
Thing.prototype.publicMethod = Thing_publicMethod;
Thing.prototype.anotherPublicMethod = Thing_anotherPublicMethod;

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

which makes all the public methods appear together, like in a good old C or C++ interface header file.

The "Ext" way to do OO

Finally, nearly all Ext classes are written this way:

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

    var privateMethod = function() {
        alert('Private methods can be called from public methods');
    }

    return {
        publicField: "PUBLIC",

        publicMethod: function() {
            privateMethod();
            alert('From the public method;
                        \nthis is a public value: '
                        + this.publicField
                        + '\nand this is a private value: '
                        + privateField);
        }
    };
}

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

In this last way of doing things, we're encapsulating the public interface of the class inside the "return" statement of the class, returning a dictionary (or literal) of members (fields and methods). This creates a neat separation of the public and private parts, with the neat advantage of allowing access to the private fields.

To use this last writing style, remember two things:

- Always remember to separate every member in the "return" clause with commas.
- Always remember to put the "return" and the opening curly bracket in the same line! That is, you must write "return {", otherwise, since semicolons are optional, the function will return null!

Inheritance

For inheritance, you can use the "prototype" keyword (basically the JavaScript native inheritance functionality) but you can use the Ext framework inheritance functions, which I recommend.