Essential JavaScript Namespacing Patterns

SEPTEMBER 23, 2011

In this post, I'll be discussing both intermediate and advanced patterns and approaches for namespacing in JavaScript. We're going to begin with the latter as I believe many of my readers have some prior experience in this area. If however you're new to namespacing with the language and would like to learn more about some of the fundamentals, please feel free to skip tothe section titled 'namespacing fundamentals' to continue reading.

What is namespacing?

In many programming languages, namespacing is a technique employed to avoid collisions with other objects or variables in the global namespace. They're also extremely useful for helping organize blocks of functionality in your application into easily manageable groups that can be uniquely identified.

In JavaScript, namespacing at an enterprise level is critical as it's important to safeguard your code from breaking in the event of another script on the page using the same variable or method names as you are. With the number of third-party tags regularly injected into pages these days, this can be a common problem we all need to tackle at some point in our careers. As a well-behaved 'citizen' of the global namespace, it's also imperative that you do your best to similarly not prevent other developer's scripts executing due to the same issues.

Whilst JavaScript doesn't really have built-in support for namespaces like other languages, it does have objects and closures which can be used to achieve a similar effect.

Advanced namespacing patterns

In this section, I'll be exploring some advanced patterns and utility techniques that have helped me when working on larger projects requiring a re-think of how application namespacing is approached. I should state that I'm not advocating any of these as *the* way to do things, but rather just ways that I've found work in practice.

Automating nested namespacing

As you're probably aware, a nested namespace provides an organized hierarchy of structures in an application and an example of such a namespace could be the following: *application.utilities.drawing.canvas.2d*. In JavaScript the equivalent of this definition using the object literal pattern would be:

```
var application = {
            utilities:{
                    drawing:{
                            canvas:{
                                    2d:{
                                            /*...*/
                                    }
                            }
                    }
            }
};
```

Wow, that's ugly.

One of the obvious challenges with this pattern is that each additional depth you wish to create requires yet another object to be defined as a child of some parent in your top-level namespace. This can become particularly laborious when multiple depths are required as your application increases in complexity.

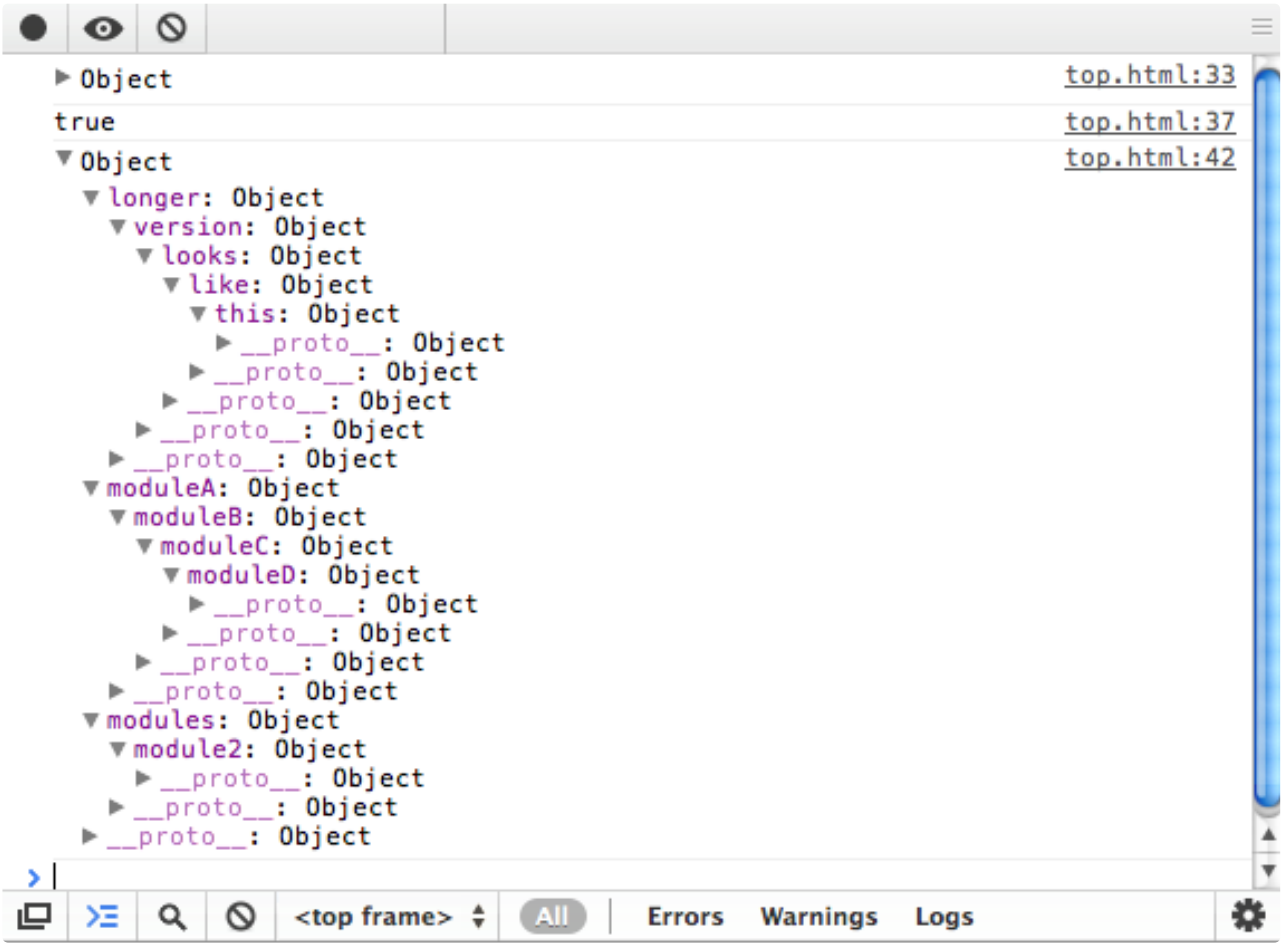
How can this problem be better solved? In JavaScript Patterns, Stoyan Stefanov presents a very-clever approach for automatically defining nested namespaces under an existing global variable using a convenience method that takes a single string argument for a nest, parses this and automatically populates your base namespace with the objects required.

The method he suggests using is the following, which I've updated it to be a generic function for easier re-use with multiple namespaces:

```
// top-level namespace being assigned an object literal
var myApp = myApp || {};

// a convenience function for parsing string namespaces and
// automatically generating nested namespaces
function extend( ns, ns_string ) {
    var parts = ns_string.split('.'),
        parent = ns,
        pl, i;

    if (parts[0] == "myApp") {
        parts = parts.slice(1);
    }

    pl = parts.length;
    for (i = 0; i < pl; i++) {
        //create a property if it doesnt exist
        if (typeof parent[parts[i]] == 'undefined') {
            parent[parts[i]] = {};
        }

        parent = parent[parts[i]];
    }

    return parent;
}

// sample usage:
// extend myApp with a deeply nested namespace
var mod = extend(myApp, 'myApp.modules.module2');
// the correct object with nested depths is output
console.log(mod);
// minor test to check the instance of mod can also
// be used outside of the myApp namesapce as a clone
// that includes the extensions
console.log(mod == myApp.modules.module2); //true
```

```
// further demonstration of easier nested namespace
// assignment using extend
extend(myApp, 'moduleA.moduleB.moduleC.moduleD');
extend(myApp, 'longer.version.looks.like.this');
console.log(myApp);
```

Web inspector output:



Note how where one would previously have had to explicitly declare the various nests for their namespace as objects, this can now be easily achieved using a single, cleaner line of code. This works exceedingly well when defining purely namespaces alone, but can seem a little less flexible when you want to define both functions and properties at the same time as declaring your namespaces. Regardless, it is still incredibly powerful and I regularly use a similar approach in some of my projects.

Dependency declaration pattern

In this section we're going to take a look at a minor augmentation to the nested namespacing pattern you may be used to seeing in some applications. We all know that local references to objects can decrease overall lookup times, but let's apply this to namespacing to see how it might look in practice:

```
// common approach to accessing nested namespaces
myApp.utilities.math.fibonacci(25);
myApp.utilities.math.sin(56);
myApp.utilities.drawing.plot(98,50,60);


// with local/cached references
Var utils = myApp.utilities,
maths = utils.math,
drawing = utils.drawing;

// easier to access the namespace
maths.fibonacci(25);
maths.sin(56);
drawing.plot(98, 50,60);

// note that the above is particularly performant when
// compared to hundreds or thousands of calls to nested
// namespaces vs. a local reference to the namespace
```

Working with a local variable here is almost always faster than working with a top-level global (eg.myApp). It's also both more convenient and more performant than accessing nested properties/sub-namespaces on every subsequent line and can improve readability in more complex applications.

Stoyan recommends declaring localized namespaces required by a function or module at the top of your function scope (using the single-variable pattern) and calls this a dependancy declaration pattern. One if the benefits this offers is a decrease in locating dependencies and resolving them, should you have an extendable architecture that dynamically loads modules into your namespace when required.

In my opinion this pattern works best when working at a modular level, localizing a namespace to be used by a group of methods. Localizing namespaces on a per-function level, especially where there is significant overlap between namespace dependencies would be something I would recommend avoiding where possible. Instead, define it further up and just have them all access the same reference.

Deep object extension

An alternative approach to automatic namespacing is deep object extension. Namespaces defined using object literal notation may be easily extended (or merged) with other objects (or namespaces) such that the properties and functions of both namespaces can be accessible under the same namespace post-merge.

This is something that's been made fairly easy to accomplish with modern JavaScript frameworks (eg. see jQuery's $.extend), however, if you're looking to extend object (namespaces) using vanilla JS, the following routine may be of assistance.

```
// extend.js
// written by andrew dupont, optimized by addy osmani
function extend(destination, source) {
    var toString = Object.prototype.toString,
        objTest = toString.call({});
    for (var property in source) {
        if (source[property] && objTest == toString.call(source[property])) {
            destination[property] = destination[property] || {};
            extend(destination[property], source[property]);
        } else {
            destination[property] = source[property];
        }
    }
    return destination;
};


console.group("objExtend namespacing tests");

// define a top-level namespace for usage
var myNS = myNS || {};

// 1. extend namespace with a 'utils' object
extend(myNS, {
        utils:{
        }
});

console.log('test 1', myNS);
//myNS.utils now exists

// 2. extend with multiple depths (namespace.hello.world.wave)
extend(myNS, {
                hello:{
                        world:{
                                wave:{
                                    test: function(){
                                        /*...*/
                                    }
                                }
                        }
                }
});

// test direct assignment works as expected
myNS.hello.test1 = 'this is a test';
myNS.hello.world.test2 = 'this is another test';
console.log('test 2', myNS);
```

```
// 3. what if myNS already contains the namespace being added
// (eg. 'library')? we want to ensure no namespaces are being
// overwritten during extension

myNS.library = {
        foo:function(){}
};

extend(myNS, {
        library:{
                bar:function(){
                    /*...*/
                }
        }
});

// confirmed that extend is operating safely (as expected)
// myNS now also contains library.foo, library.bar
console.log('test 3', myNS);


// 4. what if we wanted easier access to a specific namespace without having
// to type the whole namespace out each time?.

var shorterAccess1 = myNS.hello.world;
shorterAccess1.test3 = "hello again";
console.log('test 4', myNS);
//success, myApp.hello.world.test3 is now 'hello again'

console.groupEnd();
```

If you do happen to be using jQuery in your application, you can achieve the exact same object namespact extensibility using $.extend as seen below:

```
// top-level namespace
var myApp = myApp || {};

// directly assign a nested namespace
myApp.library = {
    foo:function(){ /*..*/}
};

// deep extend/merge this namespace with another
// to make things interesting, let's say it's a namespace
// with the same name but with a different function
// signature: $.extend(deep, target, object1, object2)
$.extend(true, myApp, {
    library:{
        bar:function(){
```

```
            /*..*/
        }
    }
});

console.log('test', myApp);
// myApp now contains both library.foo() and library.bar() methods
// nothing has been overwritten which is what we're hoping for.
```

For the sake of thoroughness, please see here for jQuery $.extend equivalents to the rest of the namespacing experiments found in this section.

Namespacing Fundamentals

Namespaces can be found in almost any serious JavaScript application. Unless you're working with a code-snippet, it's imperative that you do your best to ensure that you're implementing namespacing correctly as it's not just simple to pick-up, it'll also avoid third party code clobbering your own. The patterns we'll be examining in this section are:

- Single global variables
- Object literal notation
- Nested namespacing
- Immediately-invoked Function Expressions
- Namespace injection

1.Single global variables

One popular pattern for namespacing in JavaScript is opting for a single global variable as your primary object of reference. A skeleton implementation of this where we return an object with functions and properties can be found below:

```
var myApplication =  (function(){
        function(){
            /*...*/
        },
        return{
            /*...*/
        }
})();
```

Although this works for certain situations, the biggest challenge with the single global variable pattern is ensuring that no one else has used the same global variable name as

you have in the page.

One solution to this problem, as mentioned by Peter Michaux, is to use prefix namespacing. It's a simple concept at heart, but the idea is you select a unique prefix namespace you wish to use (in this example, "myApplication_") and then define any methods, variables or other objects after the prefix as follows:

```
var myApplication_propertyA = {};
var myApplication_propertyB = {};
funcion myApplication_myMethod(){ /*..*/ }
```

This is effective from the perspective of trying to lower the chances of a particular variable existing in the global scope, but remember that a uniquely named object can have the same effect. This aside, the biggest issue with the pattern is that it can result in a large number of global objects once your application starts to grow. There is also quite a heavy reliance on your prefix not being used by any other developers in the global namespace, so be careful if opting to use this.

For more on Peter's views about the single global variable pattern, read his excellent post on them here.

2. Object literal notation

Object literal notation can be thought of as an object containing a collection of key:value pairs with a colon separating each pair of keys and values. It's syntax requires a comma to be used after each key:value pair with the exception of the last item in your object, similar to a normal array.

```
var myApplication = {
    getInfo:function(){ /**/ },

    // we can also populate our object literal to support
    // further object literal namespaces containing anything
    // really:
    models : {},
    views : {
        pages : {}
    },
    collections : {}
};
```

One can also opt for adding properties directly to the namespace:

```
myApplication.foo = function(){
    return "bar";
}
myApplication.utils = {
    toString:function(){
        /*..*/
    },
    export: function(){
        /*..*/
    }
}
```

Object literals have the advantage of not polluting the global namespace but assist in organizing code and parameters logically. They're beneficial if you wish to create easily-readable structures that can be expanded to support deep nesting. Unlike simple global variables, object literals often also take into account tests for the existence of a variable by the same name so the chances of collision occurring are significantly reduced.

The code at the very top of the next sample demonstrates the different ways in which you can check to see if a variable (object namespace) already exists before defining it. You'll commonly see developers using Option 1, however Options 3 and 5 may be considered more thorough and Option 4 is considered a good best-practice.

```
// This doesn't check for existence of 'myApplication' in
// the global namespace. Bad practice as you can easily
// clobber an existing variable/namespace with the same name
var myApplication = {};

/*
The following options *do* check for variable/namespace existence.
If already defined, we use that instance, otherwise we assign a new
object literal to myApplication.

Option 1: var myApplication = myApplication || {};
Option 2  if(!MyApplication) MyApplication = {};
Option 3: var myApplication = myApplication = myApplication || {}
Option 4: myApplication || (myApplication = {});
Option 5: var myApplication = myApplication === undefined ? {} : myApplication;

*/
```

There is of course a huge amount of variance in how and where object literals are used for organizing and structuring code. For smaller applications wishing to expose a nested API for a particular self-enclosed module, you may just find yourself using this next

pattern when returning an interface for other developers to use. It's a variation on the module pattern where the core structure of the pattern is an IIFE, however the returned interface is an object literal:

```
var namespace = (function () {

    // defined within the local scope
    var privateMethod1 = function () { /* ... */ }
    var privateMethod2 = function () { /* ... */ }
    var privateProperty1 = 'foobar';

    return {
        // the object literal returned here can have as many
        // nested depths as you wish, however as mentioned,
        // this way of doing things works best for smaller,
        // limited-scope applications in my personal opinion
        publicMethod1: privateMethod1,

        //nested namespace with public properties
        properties:{
            publicProperty1: privateProperty1
        },

        //another tested namespace
        utils:{
            publicMethod2: privateMethod2
        }
        ...
    }
})();
```

The benefit of object literals is that they offer us a very elegant key/value syntax to work with; one where we're able to easily encapsulate any distinct logic or functionality for our application in a way that clearly separates it from others and provides a solid foundation for extending your code.

A possible downside however is that object literals have the potential to grow into long syntactic constructs. Opting to take advantage of the nested namespace pattern (which also uses the same pattern as it's base)

This pattern has a number of other useful applications too. In addition to namespacing, it's often of benefit to decouple the default configuration for your application into a single area that can be easily modified without the need to search through your entire codebase just to alter them - object literals work great for this purpose. Here's an example of a hypothetical object literal for configuration:

```
var myConfig = {
    language: 'english',
    defaults: {
        enableGeolocation: true,
        enableSharing: false,
        maxPhotos: 20
    },
    theme: {
        skin: 'a',
        toolbars: {
            index: 'ui-navigation-toolbar',
            pages: 'ui-custom-toolbar'
        }
    }
}
```

Note that there are really only minor syntactical differences between the object literal pattern and a standard JSON data set. If for any reason you wish to use JSON for storing your configurations instead (e.g. for simpler storage when sending to the back-end), feel free to. For more on the object literal pattern, I recommend reading Rebecca Murphey's excellent article on the topic.

3. Nested namespacing

An extension of the object literal pattern is nested namespacing. It's another common pattern used that offers a lower risk of collision due to the fact that even if a namespace already exists, it's unlikely the same nested children do.

Does this look familiar?

```
YAHOO.util.Dom.getElementsByClassName('test');
```

Yahoo's YUI framework uses the nested object namespacing pattern regularly and at AOL we also use this pattern in many of our main applications. A sample implementation of nested namespacing may look like this:

```
var myApp =  myApp || {};
```

```
// perform a similar existence check when defining nested
// children
myApp.routers = myApp.routers || {};
myApp.model = myApp.model || {};
myApp.model.special = myApp.model.special || {};

// nested namespaces can be as complex as required:
// myApp.utilities.charting.html5.plotGraph(/*..*/);
// myApp.modules.financePlanner.getSummary();
// myApp.services.social.facebook.realtimeStream.getLatest();
```

You can also opt to declare new nested namespaces/properties as indexed properties as follows:

```
myApp["routers"] = myApp["routers"] || {};
myApp["models"] = myApp["models"] || {};
myApp["controllers"] = myApp["controllers"] || {};
```

Both options are readable, organized and offer a relatively safe way of namespacing your application in a similar fashion to what you may be used to in other languages. The only real caveat however is that it requires your browser's JavaScript engine first locating the myApp object and then digging down until it gets to the function you actually wish to use.

This can mean an increased amount of work to perform lookups, however developers such as Juriy Zaytsev have previously tested and found the performance differences between single object namespacing vs the 'nested' approach to be quite negligible.

4. Immediately-invoked Function Expressions (IIFE)s

An IIFE is effectively an unnamed function which is immediately invoked after it's been defined. In JavaScript, because both variables and functions explicitly defined within such a context may only be accessed inside of it, function invocation provides an easy means to achieving privacy.

This is one of the many reasons why IIFEs are a popular approach to encapsulating application logic to protect it from the global namespace. You've probably come across this pattern before under the name of a self-executing (or self-invoked) anonymous function, however I personally prefer Ben Alman's naming convection for this particular pattern as I believe it to be both more descriptive and more accurate.

The simplest version of an IIFE could be the following:

```
// an (anonymous) immediately-invoked function expression
(function(){ /*...*/})();
// a named immediately-invoked function expression
(function foobar(){ /*..*/}());
// this is technically a self-executing function which is quite different
function foobar(){ foobar(); }
```

whilst a slightly more expanded version of the first example might look like:

```
var namespace = namespace || {};

// here a namespace object is passed as a function
// parameter, where we assign public methods and
// properties to it
(function( o ){
    o.foo = "foo";
    o.bar = function(){
        return "bar";
    };
})(namespace);

console.log(namespace);
```

Whilst readable, this example could be significantly expanded on to address common development concerns such as defined levels of privacy (public/private functions and variables) as well as convenient namespace extension. Let's go through some more code:

```
// namespace (our namespace name) and undefined are passed here
// to ensure 1. namespace can be modified locally and isn't
// overwritten outside of our function context
// 2. the value of undefined is guaranteed as being truly
// undefined. This is to avoid issues with undefined being
// mutable pre-ES5.

;(function ( namespace, undefined ) {
    // private properties
    var foo = "foo",
        bar = "bar";

    // public methods and properties
    namespace.foobar = "foobar";
    namespace.sayHello = function () {
        speak("hello world");
    };
```

```
        // private method
        function speak(msg) {
            console.log("You said: " + msg);
        };

        // check to evaluate whether 'namespace' exists in the
        // global namespace - if not, assign window.namespace an
        // object literal
    })(window.namespace = window.namespace || {});


    // we can then test our properties and methods as follows

    // public
    console.log(namespace.foobar); // foobar
    namescpace.sayHello(); // hello world

    // assigning new properties
    namespace.foobar2 = "foobar";
    console.log(namespace.foobar2);
```

Extensibility is of course key to any scalable namespacing pattern and IIFEs can be used to achieve this quite easily. In the below example, our 'namespace' is once again passed as an argument to our anonymous function and is then extended (or decorated) with further functionality:

```
    // let's extend the namespace with new functionality
    (function( namespace, undefined ){
        // public method
        namespace.sayGoodbye = function(){
            console.log(namespace.foo);
            console.log(namespace.bar);
            speak('goodbye');
        }
    })( window.namespace = window.namespace || {});
```

namespace.sayGoodbye(); //goodbye

That's it for IIFEs for the time-being. If you would like to find out more about this pattern, I recommend reading both Ben's IIFE post and Elijah Manor's post on namespace patterns from C#.

5. Namespace injection

Namespace injection is another variation on the IIFE where we 'inject' the methods and properties for a specific namespace from within a function wrapper using *this* as a

namespace proxy. The benefit this pattern offers is easy application of functional behaviour to multiple objects or namespaces and can come in useful when applying a set of base methods to be built on later (eg. getters and setters).

The disadvantages of this pattern are that there may be easier or more optimal approaches to achieving this goal (eg. deep object extension / merging) which I cover earlier in the article..

Below we can see an example of this pattern in action, where we use it to populate the behaviour for two namespaces: one initially defined (utils) and another which we dynamically create as a part of the functionality assignment for utils (a new namespace called *tools*).

```
var myApp = myApp || {};
myApp.utils =  {};


(function() {
    var val = 5;

    this.getValue = function() {
        return val;
    };

    this.setValue = function(newVal) {
        val = newVal;
    }

    // also introduce a new sub-namespace
    this.tools = {};

}).apply(myApp.utils);

// inject new behaviour into the tools namespace
// which we defined via the utilities module

(function(){
    this.diagnose = function(){
        return 'diagnosis';
    }
}).apply(myApp.utils.tools);

// note, this same approach to extension could be applied
// to a regular IIFE, by just passing in the context as
// an argument and modifying the context rather than just
// 'this'

// testing
console.log(myApp); //the now populated namespace
```

```
console.log(myApp.utils.getValue()); // test get
myApp.utils.setValue(25); // test set
console.log(myApp.utils.getValue());
console.log(myApp.utils.tools.diagnose());
```

Angus Croll has also previously suggested the idea of using the call API to provide a natural separation between contexts and arguments. This pattern can feel a lot more like a module creator, but as modules still offer an encapsulation solution, I'll briefly cover it for the sake of thoroughness:

```
// define a namespace we can use later
var ns = ns || {}, ns2 = ns2 || {};

// the module/namespace creator
var creator = function(val){
    var val = val || 0;

    this.next = function(){
        return val++
    };

    this.reset = function(){
        val = 0;
    }
}

creator.call(ns);
// ns.next, ns.reset now exist
creator.call(ns2, 5000);
// ns2 contains the same methods
// but has an overridden value for val
// of 5000
```

As mentioned, this type of pattern is useful for assigning a similar base set of functionality to multiple modules or namespaces, but I'd really only suggest using it where explicitly declaring your functionality within an object/closure for direct access doesn't make sense.

Conclusions

Reviewing the namespace patterns above, the option that I would personally use for most larger applications is nested object namespacing with the object literal pattern.

IIFEs and single global variables may work fine for applications in the small to medium

range, however, larger codebases requiring both namespaces and deep sub-namespaces require a succinct solution that promotes readability and scales. I feel this pattern achieves all of these objectives well.

I would also recommend trying out some of the suggested advanced utility methods for namespace extension as they really can save you time in the long-run.

Addy Osmani
Engineer at Google working on open web tooling
