12. Namespace in JS

# Namespacing in JavaScript

Global variables should be reserved for objects that have system-wide relevance and they should be named to avoid ambiguity and minimize the risk of naming collisions. In practice this means you should avoid creating global objects unless they are absolutely necessary.

But, hey, you already knew all that…..

So what do you do about it? Conventional wisdom tells us that the best global abatement strategy is to create a small number of global objects which will serve as de facto namespaces for underlying modules and subsystems. I’m going to explore several approaches to namespacing, culminating in an elegant, safe and flexible solution that I based on a recent article by [James Edwards](http://www.brothercake.com/).

#### Static Namespacing

I’m using static namespacing as an umbrella term for solutions in which the namespace label is effectively hard coded. It’s true, you could re-assign one namespace to another but the new namespace will reference the same objects as the old one.

**1. By Direct Assignment**

The most basic approach. Its verbose and if you ever wanted to rename the namespace you’ve got a job on your hands. However its safe and unambiguous.

|  |  |
| --- | --- |
|  | var myApp = {}    myApp.id = 0;    myApp.next = function() {      return myApp.id++;  }    myApp.reset = function() {      myApp.id = 0;  }    window.console && console.log(      myApp.next(),      myApp.next(),      myApp.reset(),      myApp.next()  ); //0, 1, undefined, 0 |

You could make future maintenance a little easier by using this to reference sibling properties – but this is a little risky since there is nothing to stop your namespaced functions from being reassigned:

|  |  |
| --- | --- |
|  | var myApp = {}    myApp.id = 0;    myApp.next = function() {      return this.id++;  }    myApp.reset = function() {      this.id = 0;  }    myApp.next(); //0  myApp.next(); //1  var getNextId = myApp.next;  getNextId(); //NaN whoops! |

**2. Using Object Literal Notation**

Now we need only refer to the namespace name once, so switching the name later is a little easier (assuming you haven’t already referenced the namespace too often). There is still a danger that the value of this might throw a surprise – but its a little safer to assume that objects defined within an object literal construct will not be reassigned.

|  |  |
| --- | --- |
|  | var myApp = {        id: 0,        next: function() {          return this.id++;      },        reset: function() {          this.id = 0;      }  }  window.console && console.log(      myApp.next(),      myApp.next(),      myApp.reset(),      myApp.next()  ) //0, 1, undefined, 0 |

**3. The Module Pattern**

I find myself using the [module pattern](http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth) more often these days. The logic is shielded from the global scope by a function wrapper (usually self-invoking) which returns an object representing the module’s public interface. By immediately invoking the function and assigning the result to a namespace variable, we lock up the module’s API in the namespace. Additionally any variables not included in the return value will remain forever private, visible only to the public functions that reference them.

|  |  |
| --- | --- |
|  | var myApp = (function() {        var id= 0;        return {          next: function() {              return id++;          },            reset: function() {              id = 0;          }      };  })();    window.console && console.log(      myApp.next(),      myApp.next(),      myApp.reset(),      myApp.next()  ) //0, 1, undefined, 0 |

Like the object literal example above, the receiving namespace can be easily switched, but there are added advantages: object literal notation is rigid – its all about property assignments, with no room for supporting logic. Moreover all properties must be initialized and property values cannot easily cross reference one another (so, for example, internal closures are not possible). The module pattern suffers none of these constraints and gives us the added benefit of privacy.

#### Dynamic Namespacing

We could also call this section namespace injection. The namespace is represented by a proxy which is directly referenced inside the function wrapper – which means we no longer have to bundle up a return value to assign to the namespace. This makes namespace definition more flexible and makes it very easy to have multiple independent instances of a module existing in separate namespaces (or even in the global context). Dynamic namespacing supports all the features of the module pattern with the added advantage of being intuitive and readable.

**4. Supply a Namespace Argument**

Here we simply pass the namespace as an argument to a self-invoking function. The id variable is private because it does not get assigned to the context.

|  |  |
| --- | --- |
|  | var myApp = {};  (function(context) {      var id = 0;        context.next = function() {          return id++;      };        context.reset = function() {          id = 0;      }  })(myApp);    window.console && console.log(      myApp.next(),      myApp.next(),      myApp.reset(),      myApp.next()  ) //0, 1, undefined, 0 |

We can even set the context to the global object (with a one word change!). This is a big asset for library vendors – who can wrap their features in a self-invoking function and leave it to the user to decide whether they should be global or not (John Resig was an early adopter of this concept when he wrote JQuery)

|  |  |
| --- | --- |
|  | var myApp = {};  (function(context) {      var id = 0;        context.next = function() {          return id++;      };        context.reset = function() {          id = 0;      }  })(this);    window.console && console.log(      next(),      next(),      reset(),      next()  ) //0, 1, undefined, 0 |

**5. Use this as a Namespace Proxy**

A recent posting by [James Edwards](http://www.brothercake.com/) piqued my interest. [My Favorite JavaScript Design Pattern](http://blogs.sitepoint.com/2010/11/30/my-favorite-javascript-design-pattern/)was apparently misunderstood by many commentators, who thought he might as well resort to the module pattern. The article peddles multiple techniques (which probably contributed to readers’ confusion) but at its heart is a little bit of genius which I’ve revamped and presented a namespacing tool.

The beauty of the pattern is that it simply uses the language as designed – nothing more, nothing less, no tricks, no sugar. Moreover because the namespace is injected via the this keyword (which is static within a given execution context) it cannot be accidentally modified.

|  |  |
| --- | --- |
|  | var myApp = {};  (function() {      var id = 0;        this.next = function() {          return id++;      };        this.reset = function() {          id = 0;      }  }).apply(myApp);    window.console && console.log(      myApp.next(),      myApp.next(),      myApp.reset(),      myApp.next()  ); //0, 1, undefined, 0 |

Even better, the apply (and call) APIs provide natural separation of context and arguments – so passing additional arguments to the module creator is very clean. The following example demonstrates this, and also shows how to run the module independently over multiple namespaces:

|  |  |
| --- | --- |
|  | var subsys1 = {}, subsys2 = {};  var nextIdMod = function(startId) {      var id = startId || 0;        this.next = function() {          return id++;      };        this.reset = function() {          id = 0;      }  };    nextIdMod.call(subsys1);  nextIdMod.call(subsys2,1000);    window.console && console.log(      subsys1.next(),      subsys1.next(),      subsys2.next(),      subsys1.reset(),      subsys2.next(),      subsys1.next()  ) //0, 1, 1000, undefined, 1001, 0 |

Of course if we wanted a global id generator, its a breeze…

|  |  |
| --- | --- |
|  | nextIdMod();    window.console && console.log(      next(),      next(),      reset(),      next()  ) //0, 1, undefined, 0 |

The id generator tool we’ve been using as an example does not do justice to the full potential of this pattern. By wrapping an entire library and using the this keyword as a stand in for the namespace we make it easy for the user to run the library in whichever context they choose (including the global context)

|  |  |
| --- | --- |
|  | //library code  var protoQueryMooJo = function() {      //everything  }    //user code  var thirdParty = {};  protoQueryMooJo.apply(thirdParty); |