Basic JavaScript Part 10: The Module Pattern

February 15th, 2011

Here are the links to the previous installments:

1. Functions
2. Objects
3. Prototypes
4. Enforcing New on Constructor Functions
5. Hoisting
6. Automatic Semicolon Insertion
7. Static Properties and Methods
8. Namespaces
9. Reusing Methods of Other Objects

The module pattern is quite popular in the JavaScript community as is heavily applied by many JavaScript developers. There's also the CommonJS initiative, which defines a specification for a common set of JavaScript API's that are organized using self-contained modules. These specifications are supported by a growing community as they provide the foundation for themodules that are built into Node.js and numerous other open-source JavaScript libraries. This pattern has become so widespread because it's an excellent way to package and organize an independent, self-containing piece of JavaScript code. The module pattern is composed by using self-executing functions combined with namespaces. Let's show a simple example.

```
namespace('media');

media.podcast = (function(name) {
    var fileExtension = 'mp3';

    function determineFileExtension() {
        console.log('File extension is of type ' + fileExtension);
    }

    return {
        download: function(episode) {
            console.log('Downloading ' + episode + ' of ' + name);
            determineFileExtension();
        }
    }
}('Astronomy podcast'));
```

First we define a namespace called *media*. Then we use a self-executing function that returns an anonymous object with a method named *download* that can be invoked by external code. Inside the self-executing function we have a variable *fileExtension* and a function*determineFileExtension* that are private and can only be used inside the *module*. Notice that we provide a fixed parameter value for the self-executing function. This technique is usually applied to pass in some kind of global object. jQuery uses this same approach to inject a reference to the global window object into the scope of its module.

We can use the *download* method of our module like so …

```
media.podcast.download('the first episode');
```

… which outputs what we expect:

> *Downloading the first episode of Astronomy podcast*
>
> *File extension is of type mp3*

The way we implemented the module pattern here has at least one major downside. We're able to completely replace the implementation of the *download* method that is exported by the anonymous object returned from the self-executing function. This can become quite troublesome if we have other functions inside our module that also make use of the *download*method and thereby rely on its functionality. The way to fix this issue is to make all functions private and export them using the anonymous object:

```
namespace('media');

media.podcast = (function(name) {
    var fileExtension = 'mp3';

    function determineFileExtension() {
        console.log('File extension is of type ' +fileExtension);
    }

    function download(episode) {
        console.log('Downloading ' + episode + ' of ' + name);
        determineFileExtension();
    }

    return {
        download: download
    }
}('Astronomy podcast'));
```

The *download* method exposed by the anonymous object can still be replaced, but at least the correct implementation is preserved by the private *download* function for other functions that rely on its behavior. This approach is commonly called the *"revealing module pattern"*.

Another neat approach is to export a constructor function instead of an anonymous object.

```
namespace('media');

media.Podcast = (function() {
    var fileExtension = 'mp3';

    function determineFileExtension() {
        console.log('File extension is of type ' +fileExtension);
    }

    var podcastConstructor = function Podcast(name) {
        if(false === (this instanceof Podcast)) {
            return new Podcast();
        }

        this.getName = function() {
            return name;
        }
    }

    podcastConstructor.prototype.download = function (episode) {
        console.log('Downloading ' + episode + ' of ' + this.getName());
        determineFileExtension();
    }

    return podcastConstructor;
}());
```

Instead of returning an anonymous object from our self-executing function, we create another function and add the *download* method to the prototype of this constructor function. Notice that we also moved the *name* parameter to the constructor function instead of passing it into the self-executing function. At the end of the self-executing function we just return this constructor function like we did with the anonymous object.

We can now use this module like so …

```
var astronomyCast = new media.Podcast('Astronomy podcast');
astronomyCast.download('the first episode');
```

… which yields the same output as before.

The module pattern is a very powerful concept in JavaScript. Being able to expose and use JavaScript code, treating it as a black box, is a very common technique that is used in lots of JavaScript libraries and frameworks.

Happy coding!