Writing Modular JavaScript With AMD, CommonJS & ES Harmony

WRITTEN BY: ADDY OSMANI TECHNICAL REVIEW: ANDRÉE HANSSON
WITH SPECIAL THANKS TO JAMES BURKE, JOHN HANN AND THOMAS DAVIS

Modularity The Importance Of Decoupling Your Application

When we say an application is **modular**, we generally mean it's composed of a set of highly decoupled, distinct pieces of functionality stored in modules. As you probably know, loose coupling facilitates easier maintainability of apps by removing *dependencies* where possible. When this is implemented efficiently, its quite easy to see how changes to one part of a system may affect another.

Unlike some more traditional programming languages however, the current iteration of JavaScript (ECMA-262) doesn't provide developers with the means to import such modules of code in a clean, organized manner. It's one of the concerns with specifications that haven't required great thought until more recent years where the need for more organized JavaScript applications became apparent.

Instead, developers at present are left to fall back on variations of the module or object literal patterns. With many of these, module scripts are strung together in the DOM with namespaces being described by a single global object where it's still possible to incur naming collisions in your architecture. There's also no clean way to handle dependency management without some manual effort or third party tools.

Whilst native solutions to these problems will be arriving in ES Harmony, the good news is that writing modular JavaScript has never been easier and you can start doing it today.

In this article, we're going to look at three formats for writing modular JavaScript: **AMD**, **CommonJS** and proposals for the next version of JavaScript, **Harmony**.

Prelude A Note On Script Loaders

It's difficult to discuss AMD and CommonJS modules without talking about the elephant in the room - script loaders. At present, script loading is a means to a goal, that goal being modular JavaScript that can be used in applications today - for this, use of a compatible script loader is unfortunately necessary. In order to get the most out of this article, I recommend gaining a **basic understanding** of how popular script loading tools work so the explanations of module formats make sense in context.

There are a number of great loaders for handling module loading in the AMD and CJS formats, but my personal preferences are RequireJS and curl.js. Complete tutorials on these tools are outside the scope of this article, but I can recommend reading John Hann's post about curl.js and James Burke's RequireJS API documentation for more.

From a production perspective, the use of optimization tools (like the RequireJS optimizer) to concatenate scripts is recommended for deployment when working with such modules. Interestingly, with the Almond AMD shim, RequireJS doesn't need to be rolled in the deployed site and what you might consider a script loader can be easily shifted outside of development.

That said, James Burke would probably say that being able to dynamically load scripts after page load still has its use cases and RequireJS can assist with this too. With these notes in mind, let's get started.

AMD A Format For Writing Modular JavaScript In The Browser

The overall goal for the AMD (Asynchronous Module Definition) format is to provide a solution for modular JavaScript that developers can use today. It was born out of Dojo's real world experience using XHR+eval and proponents of this format wanted to avoid any future solutions suffering from the weaknesses of those in the past.

The AMD module format itself is a proposal for defining modules where both the module and dependencies can be asynchronously loaded. It has a number of distinct advantages including being both asynchronous and highly flexible by nature which removes the tight coupling one might commonly find between code and module identity. Many developers enjoy using it and one could consider it a reliable stepping stone towards the module system proposed for ES Harmony.

AMD began as a draft specification for a module format on the CommonJS list but as it wasn't able to reach full concensus, further development of the format moved to the amdjs group.

Today it's embraced by projects including Dojo (1.7), MooTools (2.0), Firebug (1.8) and even jQuery (1.7). Although the term *CommonJS AMD format* has been seen in the wild on occasion, it's best to refer to it as just AMD or Async Module support as not all participants on the CJS list wished to pursue it.

> **Note:** There was a time when the proposal was referred to as Modules Transport/C, however as the spec wasn't geared for transporting existing CJS modules, but rather, for defining modules it made more sense to opt for the AMD naming convention.

Getting Started With Modules

The two key concepts you need to be aware of here are the idea of a `define` method for facilitating module definition and a `require` method for handling dependency loading. *define* is used to define named or unnamed modules based on the proposal using the following signature:

```
define(
    module_id /*optional*/,
    [dependencies] /*optional*/,
    definition function /*function for instantiating the module or object*/
);
```

As you can tell by the inline comments, the `module_id` is an optional argument which is typically only required when non-AMD concatenation tools are being used (there may be some other edge cases where it's useful too). When this argument is left out, we call the module anonymous.

When working with anonymous modules, the idea of a module's identity is DRY, making it trivial to avoid duplication of filenames and code. Because the code is more portable, it can be easily moved to other locations (or around the file-system) without needing to alter the code itself or change its ID. The `module_id` is equivalent to folder paths in simple packages and when not used in packages. Developers can also run the same code on multiple environments just by using an AMD optimizer that works with a CommonJS environment such as r.js.

Back to the define signature, the dependencies argument represents an array of dependencies which are required by the module you are defining and the third argument ('definition function') is a function that's executed to instantiate your module. A barebone module could be defined as follows:

Understanding AMD: define()

```

// A module_id (myModule) is used here for demonstration purposes only

define('myModule',
    ['foo', 'bar'],
    // module definition function
    // dependencies (foo and bar) are mapped to function parameters
    function ( foo, bar ) {
        // return a value that defines the module export
        // (i.e the functionality we want to expose for consumption)

        // create your module here
        var myModule = {
            doStuff:function(){
                console.log('Yay! Stuff');
```

```
            }
        }

        return myModule;
});

// An alternative example could be..
define('myModule',
    ['math', 'graph'],
    function ( math, graph ) {

        // Note that this is a slightly different pattern
        // With AMD, it's possible to define modules in a few
        // different ways due as it's relatively flexible with
        // certain aspects of the syntax
        return {
            plot: function(x, y){
                return graph.drawPie(math.randomGrid(x,y));
            }
        }
    };
});
```

require on the other hand is typically used to load code in a top-level JavaScript file or within a module should you wish to dynamically fetch dependencies. An example of its usage is:

Understanding AMD: require()

```
// Consider 'foo' and 'bar' are two external modules
// In this example, the 'exports' from the two modules loaded are passed as
// function arguments to the callback (foo and bar)
// so that they can similarly be accessed

require(['foo', 'bar'], function ( foo, bar ) {
        // rest of your code here
        foo.doSomething();
});
```

Dynamically-loaded Dependencies

```

define(function ( require ) {
    var isReady = false, foobar;

    // note the inline require within our module definition
    require(['foo', 'bar'], function (foo, bar) {
        isReady = true;
        foobar = foo() + bar();
    });

    // we can still return a module
```

```
    return {
        isReady: isReady,
        foobar: foobar
    };
});

```

Understanding AMD: plugins

The following is an example of defining an AMD-compatible plugin:

```
// With AMD, it's possible to load in assets of almost any kind
// including text-files and HTML. This enables us to have template
// dependencies which can be used to skin components either on
// page-load or dynamically.

define(['./templates', 'text!./template.md','css!./template.css'],
    function( templates, template ){
        console.log(templates);
        // do some fun template stuff here.
    }
});
```

Note: Although css! is included for loading CSS dependencies in the above example, it's important to remember that this approach has some caveats such as it not being fully possible to establish when the CSS is fully loaded. Depending on how you approach your build, it may also result in CSS being included as a dependency in the optimized file, so use CSS as a loaded dependency in such cases with caution.

Loading AMD Modules Using require.js

```
require(['app/myModule'],
    function( myModule ){
        // start the main module which in-turn
        // loads other modules
        var module = new myModule();
        module.doStuff();
});
```

Loading AMD Modules Using curl.js

```
curl(['app/myModule.js'],
    function( myModule ){
        // start the main module which in-turn
        // loads other modules
        var module = new myModule();
        module.doStuff();
});
```

Modules With Deferred Dependencies

```
// This could be compatible with jQuery's Deferred implementation,
// futures.js (slightly different syntax) or any one of a number
// of other implementations
define(['lib/Deferred'], function( Deferred ){
    var defer = new Deferred();
    require(['lib/templates/?index.html','lib/data/?stats'],
        function( template, data ){
            defer.resolve({ template: template, data:data });
        }
    );
    return defer.promise();
});
```

Why Is AMD A Better Choice For Writing Modular JavaScript?

- Provides a clear proposal for how to approach defining flexible modules.
- Significantly cleaner than the present global namespace and `<script>` tag solutions many of us rely on. There's a clean way to declare stand-alone modules and dependencies they may have.
- Module definitions are encapsulated, helping us to avoid pollution of the global namespace.
- Works better than some alternative solutions (eg. CommonJS, which we'll be looking at shortly). Doesn't have issues with cross-domain, local or debugging and doesn't have a reliance on server-side tools to be used. Most AMD loaders support loading modules in the browser without a build process.
- Provides a 'transport' approach for including multiple modules in a single file. Other approaches like CommonJS have yet to agree on a transport format.
- It's possible to lazy load scripts if this is needed.

Related Reading
The RequireJS Guide To AMD
What's the fastest way to load AMD modules?
AMD vs. CJS, what's the better format?
AMD Is Better For The Web Than CommonJS Modules
The Future Is Modules Not Frameworks
AMD No Longer A CommonJS Specification
On Inventing JavaScript Module Formats And Script Loaders
The AMD Mailing List

AMD Modules With Dojo

Defining AMD-compatible modules using Dojo is fairly straight-forward. As per above, define any module dependencies in an array as the first argument and provide a callback (factory) which will execute the module once the dependencies have been loaded. e.g:

```
define(["dijit/Tooltip"], function( Tooltip ){
    //Our dijit tooltip is now available for local use
    new Tooltip(...);
});
```

Note the anonymous nature of the module which can now be both consumed by a Dojo asynchronous loader, RequireJS or the standard dojo.require() module loader that you may be used to using.

For those wondering about module referencing, there are some interesting gotchas that are useful to know here. Although the AMD-advocated way of referencing modules declares them in the dependency list with a set of matching arguments, this isn't supported by the Dojo 1.6 build system - it really only works for AMD-compliant loaders. e.g:

```
define(["dojo/cookie", "dijit/Tooltip"], function( cookie, Tooltip ){
    var cookieValue = cookie("cookieName");
    new Tree(...);
});
```

This has many advances over nested namespacing as modules no longer need to directly reference complete namespaces every time - all we require is the 'dojo/cookie' path in dependencies, which once aliased to an argument, can be referenced by that variable. This removes the need to repeatedly type out 'dojo.' in your applications.

> **Note:** Although Dojo 1.6 doesn't officially support user-based AMD modules (nor asynchronous loading), it's possible to get this working with Dojo using a number of different script loaders. At present, all Dojo core and Dijit modules have been transformed to the AMD syntax and improved overall AMD support will likely land between 1.7 and 2.0.

The final gotcha to be aware of is that if you wish to continue using the Dojo build system or wish to migrate older modules to this newer AMD-style, the following more verbose version enables easier migration. Notice that dojo and dijit and referenced as dependencies too:

```
define(["dojo", "dijit", "dojo/cookie", "dijit/Tooltip"], function(dojo,
dijit){
    var cookieValue = dojo.cookie("cookieName");
    new dijit.Tooltip(...);
});
```

AMD Module Design Patterns (Dojo)

If you've followed any of my previous posts on the benefits of design patterns, you'll know that they can be highly effective in improving how we approach structuring solutions to common development problems. John Hann recently gave an excellent presentation about AMD module design patterns covering the Singleton, Decorator, Mediator and others. I highly recommend checking out his slides if you get a chance.

Some samples of these patterns can be found below:

Decorator pattern:

```
1.
2. // mylib/UpdatableObservable: a decorator for dojo/store/Observable
3. define(['dojo', 'dojo/store/Observable'], function ( dojo, Observable ) {
4.     return function UpdatableObservable ( store ) {
5.
6.         var observable = dojo.isFunction(store.notify) ? store :
7.                 new Observable(store);
8.
9.         observable.updated = function( object ) {
10.             dojo.when(object, function ( itemOrArray) {
11.                 dojo.forEach( [].concat(itemOrArray), this.notify, this );
12.             };
13.         };
14.
15.         return observable; // makes `new` optional
16.     };
17. });
18.
19.
20. // decorator consumer
21. // a consumer for mylib/UpdatableObservable
22.
23. define(['mylib/UpdatableObservable'], function ( makeUpdatable ) {
24.     var observable, updatable, someItem;
25.     // ... here be code to get or create `observable`
26.
27.     // ... make the observable store updatable
28.     updatable = makeUpdatable(observable); // `new` is optional!
29.
30.     // ... later, when a cometd message arrives with new data item
31.     updatable.updated(updatedItem);
32. });
33.
```

Adapter pattern

```
1.
2. // 'mylib/Array' adapts `each` function to mimic jQuery's:
3. define(['dojo/_base/lang', 'dojo/_base/array'], function (lang, array) {
4.     return lang.delegate(array, {
5.         each: function (arr, lambda) {
```

```
            array.forEach(arr, function (item, i) {
                lambda.call(item, i, item); // like jQuery's each
            })
        }
    });
});

// adapter consumer
// 'myapp/my-module':
define(['mylib/Array'], function ( array ) {
    array.each(['uno', 'dos', 'tres'], function (i, esp) {
        // here, `this` == item
    });
});

```

AMD Modules With jQuery

The Basics

Unlike Dojo, jQuery really only comes with one file, however given the plugin-based nature of the library, we can demonstrate how straight-forward it is to define an AMD module that uses it below.

```
define(['js/jquery.js','js/jquery.color.js','js/underscore.js'],
    function($, colorPlugin, _){
        // Here we've passed in jQuery, the color plugin and Underscore
        // None of these will be accessible in the global scope, but we
        // can easily reference them below.

        // Pseudo-randomize an array of colors, selecting the first
        // item in the shuffled array
        var shuffleColor = _.first(_.shuffle(['#666','#333','#111']));

        // Animate the background-color of any elements with the class
        // 'item' on the page using the shuffled color
        $('.item').animate({'backgroundColor': shuffleColor });

        return {};
        // What we return can be used by other modules
    });
```

There is however something missing from this example and it's the concept of registration.

Registering jQuery As An Async-compatible Module

One of the key features that landed in jQuery 1.7 was support for registering jQuery as an asynchronous module. There are a number of compatible script loaders (including

RequireJS and curl) which are capable of loading modules using an asynchronous module format and this means fewer hacks are required to get things working.

As a result of jQuery's popularity, AMD loaders need to take into account multiple versions of the library being loaded into the same page as you ideally don't want several different versions loading at the same time. Loaders have the option of either specifically taking this issue into account or instructing their users that there are known issues with third party scripts and their libraries.

What the 1.7 addition brings to the table is that it helps avoid issues with other third party code on a page accidentally loading up a version of jQuery on the page that the owner wasn't expecting. You don't want other instances clobbering your own and so this can be of benefit.

The way this works is that the script loader being employed indicates that it supports multiple jQuery versions by specifying that a property, `define.amd.jQuery` is equal to true. For those interested in more specific implementation details, we register jQuery as a named module as there is a risk that it can be concatenated with other files which may use AMD's `define()` method, but not use a proper concatenation script that understands anonymous AMD module definitions.

The named AMD provides a safety blanket of being both robust and safe for most use-cases.

```
// Account for the existence of more than one global
// instances of jQuery in the document, cater for testing
// .noConflict()

var jQuery = this.jQuery || "jQuery",
$ = this.$ || "$",
originaljQuery = jQuery,
original$ = $,
amdDefined;

define(['jquery'] , function ($) {
    $('.items').css('background','green');
    return function () {};
});

// The very easy to implement flag stating support which
// would be used by the AMD loader
define.amd = {
    jQuery: true
};
```

Smarter jQuery Plugins

I've recently discussed some ideas and examples of how jQuery plugins could be written using Universal Module Definition (UMD) patterns here. UMDs define modules that can work on both the client and server, as well as with all popular script loaders available at the moment. Whilst this is still a new area with a lot of concepts still being finalized, feel free to look at the code samples in the section title *AMD && CommonJS* below and let me know if you feel there's anything we could do better.

What Script Loaders & Frameworks Support AMD?

In-browser:

- RequireJS http://requirejs.org
- curl.js http://github.com/unscriptable/curl
- bdLoad http://bdframework.com/bdLoad
- Yabble http://github.com/jbrantly/yabble
- PINF http://github.com/pinf/loader-js
- (and more)

Server-side:

- RequireJS http://requirejs.org
- PINF http://github.com/pinf/loader-js

AMD Conclusions

The above are very trivial examples of just how useful AMD modules can truly be, but they hopefully provide a foundation for understanding how they work.

You may be interested to know that many visible large applications and companies currently use AMD modules as a part of their architecture. These include IBM and the BBC iPlayer, which highlight just how seriously this format is being considered by developers at an enterprise-level.

For more reasons why many developers are opting to use AMD modules in their applications, you may be interested in this post by James Burke.

CommonJS A Module Format Optimized For The Server

CommonJS are a volunteer working group which aim to design, prototype and standardize JavaScript APIs. To date they've attempted to ratify standards for both modules and packages. The CommonJS module proposal specifies a simple API for declaring modules server-side and unlike AMD attempts to cover a broader set of concerns such as io, filesystem, promises and more.

Getting Started

From a structure perspective, a CJS module is a reusable piece of JavaScript which exports specific objects made available to any dependent code - there are typically no function wrappers around such modules (so you won't see `define` used here for example).

At a high-level they basically contain two primary parts: a free variable named `exports` which contains the objects a module wishes to make available to other modules and a `require` function that modules can use to import the exports of other modules.

Understanding CJS: require() and exports

```
// package/lib is a dependency we require
var lib = require('package/lib');

// some behaviour for our module
function foo(){
    lib.log('hello world!');
}

// export (expose) foo to other modules
exports.foo = foo;
```

Basic consumption of exports

```

// define more behaviour we would like to expose
function foobar(){
        this.foo = function(){
                console.log('Hello foo');
        }

        this.bar = function(){
                console.log('Hello bar');
        }
}

// expose foobar to other modules
exports.foobar = foobar;


// an application consuming 'foobar'

// access the module relative to the path
// where both usage and module files exist
// in the same directory

var foobar = require('./foobar').foobar,
    test   = new foobar();

test.bar(); // 'Hello bar'

```

AMD-equivalent Of The First CJS Example

```
define(['package/lib'], function(lib){

    // some behaviour for our module
    function foo(){
        lib.log('hello world!');
    }

```

```
    // export (expose) foo for other modules
    return {
        foobar: foo
    };
});
```

Consuming Multiple Dependencies

app.js

```
var modA = require('./foo');
var modB = require('./bar');

exports.app = function(){
    console.log('Im an application!');
}

exports.foo = function(){
    return modA.helloWorld();
}
```

bar.js

```
exports.name = 'bar';
```

foo.js

```
require('./bar');
exports.helloWorld = function(){
    return 'Hello World!!''
}
```

What Loaders & Frameworks Support CJS?

In-browser:

- curl.js http://github.com/unscriptable/curl
- SproutCore 1.1 http://sproutcore.com
- PINF http://github.com/pinf/loader-js
- (and more)

Server-side:

- Nodehttp://nodejs.org
- Narwhal https://github.com/tlrobinson/narwhal
- Perseverehttp://www.persvr.org/
- Wakandahttp://www.wakandasoft.com/

Is CJS Suitable For The Browser?

There are developers that feel CommonJS is better suited to server-side development which is one reason there's currently a level of **disagreement** over which format should and will be used as the de facto standard in the pre-Harmony age moving forward. Some of the arguments against CJS include a note that many CommonJS APIs address server-oriented features which one would simply not be able to implement at a browser-level in JavaScript - for example, *io*, *system* and *js* could be considered unimplementable by the nature of their functionality.

That said, it's useful to know how to structure CJS modules regardless so that we can better appreciate how they fit in when defining modules which may be used everywhere. Modules which have applications on both the client and server include validation, conversion and templating engines. The way some developers are approaching choosing which format to use is opting for CJS when a module can be used in a server-side environment and using AMD if this is not the case.

As AMD modules are capable of using plugins and can define more granular things like constructors and functions this makes sense. CJS modules are only able to define objects which can be tedious to work with if you're trying to obtain constructors out of them.

Although it's beyond the scope of this article, you may have also noticed that there were different types of 'require' methods mentioned when discussing AMD and CJS.

The concern with a similar naming convention is of course confusion and the community are currently split on the merits of a global require function. John Hann's suggestion here is that rather than calling it 'require', which would probably fail to achieve the goal of informing users about the different between a global and inner require, it may make more sense to rename the global loader method something else (e.g. the name of the library). It's for this reason that a loader like curl.js uses `curl()` as opposed to `require`.

Related Reading
Demystifying CommonJS Modules
JavaScript Growing Up
The RequireJS Notes On CommonJS
Taking Baby Steps With Node.js And CommonJS - Creating Custom Modules
Asynchronous CommonJS Modules for the Browser
The CommonJS Mailing List

AMD && CommonJS Competing, But Equally Valid Standards

Whilst this article has placed more emphasis on using AMD over CJS, the reality is that both formats are valid and have a use.

AMD adopts a browser-first approach to development, opting for asynchronous behaviour and simplified backwards compatability but it doesn't have any concept of File I/O. It

supports objects, functions, constructors, strings, JSON and many other types of modules, running natively in the browser. It's incredibly flexible.

CommonJS on the other hand takes a server-first approach, assuming synchronous behaviour, no global *baggage* as John Hann would refer to it as and it attempts to cater for the future (on the server). What we mean by this is that because CJS supports unwrapped modules, it can feel a little more close to the ES.next/Harmony specifications, freeing you of the `define()` wrapper that AMD enforces. CJS modules however only support objects as modules.

Although the idea of yet another module format may be daunting, you may be interested in some samples of work on hybrid AMD/CJS and Univeral AMD/CJS modules.

Basic AMD Hybrid Format (John Hann)

```
define( function (require, exports, module){

    var shuffler = require('lib/shuffle');

    exports.randomize = function( input ){
        return shuffler.shuffle(input);
    }
});
```

AMD/CommonJS Universal Module Definition (Variation 2, UMDjs)

```
/**
 * exports object based version, if you need to make a
 * circular dependency or need compatibility with
 * commonjs-like environments that are not Node.
 */
(function (define) {
    //The 'id' is optional, but recommended if this is
    //a popular web library that is used mostly in
    //non-AMD/Node environments. However, if want
    //to make an anonymous module, remove the 'id'
    //below, and remove the id use in the define shim.
    define('id', function (require, exports) {
        //If have dependencies, get them here
        var a = require('a');

        //Attach properties to exports.
        exports.name = value;
    });
}(typeof define === 'function' && define.amd ? define : function (id,
factory) {
    if (typeof exports !== 'undefined') {
        //commonjs
        factory(require, exports);
    } else {
        //Create a global function. Only works if
```

```
        //the code does not have dependencies, or
        //dependencies fit the call pattern below.
        factory(function(value) {
            return window[value];
        }, (window[id] = {}));
    }
}));
```

Extensible UMD Plugins With (Variation by myself and Thomas Davis).

core.js

```
// Module/Plugin core
// Note: the wrapper code you see around the module is what enables
// us to support multiple module formats and specifications by
// mapping the arguments defined to what a specific format expects
// to be present. Our actual module functionality is defined lower
// down, where a named module and exports are demonstrated.

;(function ( name, definition ){
  var theModule = definition(),
      // this is considered "safe":
      hasDefine = typeof define === 'function' && define.amd,
      // hasDefine = typeof define === 'function',
      hasExports = typeof module !== 'undefined' && module.exports;

  if ( hasDefine ){ // AMD Module
    define(theModule);
  } else if ( hasExports ) { // Node.js Module
    module.exports = theModule;
  } else { // Assign to common namespaces or simply the global object
(window)
    (this.jQuery || this.ender || this.$ || this)[name] = theModule;
  }
})( 'core', function () {
    var module = this;
    module.plugins = [];
    module.highlightColor = "yellow";
    module.errorColor = "red";

  // define the core module here and return the public API

  // this is the highlight method used by the core highlightAll()
  // method and all of the plugins highlighting elements different
  // colors
  module.highlight = function(el,strColor){
    // this module uses jQuery, however plain old JavaScript
    // or say, Dojo could be just as easily used.
    if(this.jQuery){
      jQuery(el).css('background', strColor);
```

```
    }
  }
  return {
      highlightAll:function(){
        module.highlight('div', module.highlightColor);
      }
  };

});
```

myExtension.js

```
;(function ( name, definition ) {
    var theModule = definition(),
        hasDefine = typeof define === 'function',
        hasExports = typeof module !== 'undefined' && module.exports;

    if ( hasDefine ) { // AMD Module
        define(theModule);
    } else if ( hasExports ) { // Node.js Module
        module.exports = theModule;
    } else { // Assign to common namespaces or simply the global object (window)


        // account for for flat-file/global module extensions
        var obj = null;
        var namespaces = name.split(".");
        var scope = (this.jQuery || this.ender || this.$ || this);
        for (var i = 0; i < namespaces.length; i++) {
            var packageName = namespaces[i];
            if (obj && i == namespaces.length - 1) {
                obj[packageName] = theModule;
            } else if (typeof scope[packageName] === "undefined") {
                scope[packageName] = {};
            }
            obj = scope[packageName];
        }

    }
})('core.plugin', function () {

    // define your module here and return the public API
    // this code could be easily adapted with the core to
    // allow for methods that overwrite/extend core functionality
    // to expand the highlight method to do more if you wished.
    return {
        setGreen: function ( el ) {
            highlight(el, 'green');
        },
        setRed: function ( el ) {
```

```
            highlight(el, errorColor);
        }
    };

});
```

app.js

```
$(function(){

    // the plugin 'core' is exposed under a core namespace in
    // this example which we first cache
    var core = $.core;

    // use then use some of the built-in core functionality to
    // highlight all divs in the page yellow
    core.highlightAll();

    // access the plugins (extensions) loaded into the 'plugin'
    // namespace of our core module:

    // Set the first div in the page to have a green background.
    core.plugin.setGreen("div:first");
    // Here we're making use of the core's 'highlight' method
    // under the hood from a plugin loaded in after it

    // Set the last div to the 'errorColor' property defined in
    // our core module/plugin. If you review the code further down
    // you'll see how easy it is to consume properties and methods
    // between the core and other plugins
    core.plugin.setRed('div:last');
});
```

ES Harmony Modules Of The Future

TC39, the standards body charged with defining the syntax and semantics of ECMAScript and its future iterations is composed of a number of very intelligent developers. Some of these developers (such as Alex Russell) have been keeping a close eye on the evolution of JavaScript usage for large-scale development over the past few years and are acutely aware of the need for better language features for writing more modular JS.

For this reason, there are currently proposals for a number of exciting additions to the language including flexible modules that can work on both the client and server, a module loader and more. In this section, I'll be showing you some code samples of the syntax for modules in ES.next so you can get a taste of what's to come.

Note: Although Harmony is still in the proposal phases, you can already try out (partial) features of ES.next that address native support for writing modular JavaScript thanks to Google's Traceur compiler. To get up and running with Traceur in under a minute, read this getting started guide. There's also a JSConf presentation about it that's worth looking at if you're interested in learning more about the project.

Modules With Imports And Exports

If you've read through the sections on AMD and CJS modules you may be familiar with the concept of module dependencies (imports) and module exports (or, the public API/variables we allow other modules to consume). In ES.next, these concepts have been proposed in a slightly more succinct manner with dependencies being specified using an `import` keyword. `export` isn't greatly different to what we might expect and I think many developers will look at the code below and instantly 'get' it.

- **import** declarations bind a module's exports as local variables and may be renamed to avoid name collisions/conflicts.
- **export** declarations declare that a local-binding of a module is externally visible such that other modules may read the exports but can't modify them. Interestingly, modules may export child modules however can't export modules that have been defined elsewhere. You may also rename exports so their external name differs from their local names.

```
1.
2. module staff{
3.     // specify (public) exports that can be consumed by
4.     // other modules
5.     export var baker = {
6.         bake: function( item ){
7.             console.log('Woo! I just baked ' + item);
8.         }
9.     }
10. }
11.
12. module skills{
13.     export var specialty = "baking";
14.     export var experience = "5 years";
15. }
16.
17. module cakeFactory{
18.
19.     // specify dependencies
20.     import baker from staff;
21.
22.     // import everything with wildcards
23.     import * from skills;
24.
25.     export var oven = {
26.         makeCupcake: function( toppings ){
```

```
            baker.bake('cupcake', toppings);
        },
        makeMuffin: function( mSize ){
            baker.bake('muffin', size);
        }
    }
}
```

Modules Loaded From Remote Sources

The module proposals also cater for modules which are remotely based (e.g. a third-party API wrapper) making it simplistic to load modules in from external locations. Here's an example of us pulling in the module we defined above and utilizing it:

```
module cakeFactory from 'http://addyosmani.com/factory/cakes.js';
cakeFactory.oven.makeCupcake('sprinkles');
cakeFactory.oven.makeMuffin('large');
```

Module Loader API

The module loader proposed describes a dynamic API for loading modules in highly controlled contexts. Signatures supported on the loader include `load( url, moduleInstance, error)` for loading modules, `createModule( object, globalModuleReferences)` and others. Here's another example of us dynamically loading in the module we initially defined. Note that unlike the last example where we pulled in a module from a remote source, the module loader API is better suited to dynamic contexts.

```
Loader.load('http://addyosmani.com/factory/cakes.js',
    function(cakeFactory){
        cakeFactory.oven.makeCupcake('chocolate');
    });
```

CommonJS-like Modules For The Server

For developers who are server-oriented, the module system proposed for ES.next isn't just constrained to looking at modules in the browser. Below for examples, you can see a CJS-like module proposed for use on the server:

```
// io/File.js
export function open(path) { ... };
export function close(hnd) { ... };
```

```
// compiler/LexicalHandler.js
module file from 'io/File';

import { open, close } from file;
export function scan(in) {
    try {
        var h = open(in) ...
```

```
    }
    finally { close(h) }
}
```

```
module lexer from 'compiler/LexicalHandler';
module stdlib from '@std';

//... scan(cmdline[0]) ...
```

Classes With Constructors, Getters & Setters

The notion of a class has always been a contentious issue with purists and we've so far got along with either falling back on JavaScript's prototypal nature or through using frameworks or abstractions that offer the ability to use *class* definitions in a form that desugars to the same prototypal behavior.

In Harmony, classes come as part of the language along with constructors and (finally) some sense of true privacy. In the following examples, I've included some inline comments to help you understand how classes are structured, but you may also notice the lack of the word 'function' in here. This isn't a typo error: TC39 have been making a conscious effort to decrease our abuse of the `function` keyword for everything and the hope is that this will help simplify how we write code.

```
class Cake{

    // We can define the body of a class' constructor
    // function by using the keyword 'constructor' followed
    // by an argument list of public and private declarations.
    constructor( name, toppings, price, cakeSize ){
        public name = name;
        public cakeSize = cakeSize;
        public toppings = toppings;
        private price = price;

    }

    // As a part of ES.next's efforts to decrease the unnecessary
    // use of 'function' for everything, you'll notice that it's
    // dropped for cases such as the following. Here an identifier
    // followed by an argument list and a body defines a new method

    addTopping( topping ){
        public(this).toppings.push(topping);
    }

    // Getters can be defined by declaring get before
    // an identifier/method name and a curly body.
    get allToppings(){
        return public(this).toppings;
    }
```

```
28.
29.     get qualifiesForDiscount(){
30.         return private(this).price > 5;
31.     }
32.
33.     // Similar to getters, setters can be defined by using
34.     // the 'set' keyword before an identifier
35.     set cakeSize( cSize ){
36.         if( cSize < 0 ){
37.             throw new Error('Cake must be a valid size -
38.             either small, medium or large');
39.         }
40.         public(this).cakeSize = cSize;
41.     }
42.
43.
44. }
```

ES Harmony Conclusions

As you can see, ES.next is coming with some exciting new additions. Although Traceur can be used to an extent to try our such features in the present, remember that it may not be the best idea to plan out your system to use Harmony (just yet). There are risks here such as specifications changing and a potential failure at the cross-browser level (IE9 for example will take a while to die) so your best bets until we have both spec finalization and coverage are AMD (for in-browser modules) and CJS (for those on the server).

Related Reading
A First Look At The Upcoming JavaScript Modules
David Herman On JavaScript/ES.Next (Video)
ES Harmony Module Proposals
ES Harmony Module Semantics/Structure Rationale
ES Harmony Class Proposals

Conclusions And Further Reading A Review

In this article we've reviewed several of the options available for writing modular JavaScript using modern module formats. These formats have a number of advantages over using the (classical) module pattern alone including: avoiding a need for developers to create global variables for each module they create, better support for static and dynamic dependency management, improved compatibility with script loaders, better (optional) compatibility for modules on the server and more.

In short, I recommend trying out what's been suggested today as these formats offer a lot of power and flexibility that can help when building applications based on many reusable

blocks of functionality.

And that's it for now. If you have further questions about any of the topics covered today, feel free to hit me up on twitter and I'll do my best to help!