You Don't Know JS Yet: Scope & Closures - 2nd Edition

Chapter 6: Module Pattern

NOTE: |
:— |
Work in progress |

.

.

.

.

.

.

.

NOTE: |
:— |
Everything below here is previous text from 1st edition, and is only here for reference while 2nd edition work is underway. Please ignore. |

Modules

There are other code patterns which leverage the power of closure but which do not on the surface appear to be about callbacks. Let's examine the most powerful of them: *the module*.

```
function foo() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }
}
```

As this code stands right now, there's no observable closure going on. We simply have some private data variables `something` and `another`, and a couple of inner functions `doSomething()` and `doAnother()`, which both have lexical scope (and thus closure!) over the inner scope of `foo()`.

But now consider:

```
function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
}

var foo = CoolModule();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

This is the pattern in JavaScript we call *module*. The most common way of implementing the module pattern is often called "Revealing Module", and it's the variation we present here.

Let's examine some things about this code.

Firstly, `CoolModule()` is just a function, but it *has to be invoked* for there to be a module instance created. Without the execution of the outer function, the creation of the inner scope and the closures would not occur.

Secondly, the `CoolModule()` function returns an object, denoted by the object-literal syntax `{ key: value, ... }`. The object we return has references on it to our inner functions, but *not* to our inner data variables. We keep those hidden and private. It's appropriate to think of this object return value as essentially a **public API for our module**.

This object return value is ultimately assigned to the outer variable `foo`, and then we can access those property methods on the API, like `foo.doSomething()`.

Note: It is not required that we return an actual object (literal) from our module. We could just return back an inner function directly. jQuery is actually a good example of this. The `jQuery` and `$` identifiers are the public API for the jQuery "module", but they are, themselves, just a function (which can itself have properties, since all functions are objects).

The `doSomething()` and `doAnother()` functions have closure over the inner scope of the module "instance" (arrived at by actually invoking `CoolModule()`). When we transport those functions outside of the lexical scope, by way of property references on the object we return, we have now set up a condition by which closure can be observed and exercised.

To state it more simply, there are two "requirements" for the module pattern to be exercised:

1. There must be an outer enclosing function, and it must be invoked at least once (each time creates a new module instance).
2. The enclosing function must return back at least one inner function, so that this inner function has closure over the private scope, and can access and/or modify that private state.

An object with a function property on it alone is not *really* a module. An object which is returned from a function invocation which only has data properties on it and no closured functions is not *really* a module, in the observable sense.

The code snippet above shows a standalone module creator called `CoolModule()` which can be invoked any number of times, each time creating a new module instance. A slight variation on this pattern is when you only care to have one instance, a "singleton" of sorts:

```
var foo = (function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }
    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
})();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

Here, we turned our module function into an IIFE (see Chapter 3), and we *immediately* invoked it and assigned its return value directly to our single module instance identifier `foo`.

Modules are just functions, so they can receive parameters:

```
function CoolModule(id) {
    function identify() {
        console.log( id );
    }

    return {
        identify: identify
    };
}

var foo1 = CoolModule( "foo 1" );
var foo2 = CoolModule( "foo 2" );

foo1.identify(); // "foo 1"
foo2.identify(); // "foo 2"
```

Another slight but powerful variation on the module pattern is to name the object you are returning as your public API:

```
var foo = (function CoolModule(id) {
    function change() {
        // modifying the public API
        publicAPI.identify = identify2;
    }

    function identify1() {
        console.log( id );
    }
    function identify2() {
        console.log( id.toUpperCase() );
    }

    var publicAPI = {
        change: change,
        identify: identify1
    };

    return publicAPI;
})( "foo module" );

foo.identify(); // foo module
foo.change();
foo.identify(); // FOO MODULE
```

By retaining an inner reference to the public API object inside your module instance, you can modify that module instance **from the inside**, including adding and removing methods, properties, *and* changing their values.

Modern Modules

Various module dependency loaders/managers essentially wrap up this pattern of module definition into a friendly API. Rather than examine any one particular library, let me present a *very simple* proof of concept **for illustration purposes (only)**:

```
var MyModules = (function Manager() {
    var modules = {};

    function define(name, deps, impl) {
        for (var i=0; i<deps.length; i++) {
            deps[i] = modules[deps[i]];
        }
        modules[name] = impl.apply( impl, deps );
    }

    function get(name) {
        return modules[name];
    }

    return {
        define: define,
        get: get
    };
})();
```

The key part of this code is `modules[name] = impl.apply(impl, deps)`. This is invoking the definition wrapper function for a module (passing in any dependencies), and storing the return value, the module's API, into an internal list of modules tracked by name.

And here's how I might use it to define some modules:

```
MyModules.define( "bar", [], function(){
    function hello(who) {
        return "Let me introduce: " + who;
    }

    return {
        hello: hello
    };
} );

MyModules.define( "foo", ["bar"], function(bar){
    var hungry = "hippo";

    function awesome() {
        console.log( bar.hello( hungry ).toUpperCase() );
    }

    return {
        awesome: awesome
    };
} );

var bar = MyModules.get( "bar" );
var foo = MyModules.get( "foo" );

console.log(
    bar.hello( "hippo" )
); // Let me introduce: hippo

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

Both the "foo" and "bar" modules are defined with a function that returns a public API. "foo" even receives the instance of "bar" as a dependency parameter, and can use it accordingly.

Spend some time examining these code snippets to fully understand the power of closures put to use for our own good purposes. The key take-away is that there's not really any particular "magic" to module managers. They fulfill both characteristics of the module pattern I listed above: invoking a function definition wrapper, and keeping its return value as the API for that module.

In other words, modules are just modules, even if you put a friendly wrapper tool on top of them.

Future Modules

ES6 adds first-class syntax support for the concept of modules. When loaded via the module system, ES6 treats a file as a separate module. Each module can both import other modules or specific API members, as well export their own public API members.

Note: Function-based modules aren't a statically recognized pattern (something the compiler knows about), so their API semantics aren't considered until run-time. That is, you can actually modify a module's API during the run-time (see earlier `publicAPI` discussion).

By contrast, ES6 Module APIs are static (the APIs don't change at run-time). Since the compiler knows *that*, it can (and does!) check during (file loading and) compilation that a reference to a member of an imported module's API *actually exists*. If the API reference doesn't exist, the compiler throws an "early" error at compile-time, rather than waiting for traditional dynamic run-time resolution (and errors, if any).

ES6 modules **do not** have an "inline" format, they must be defined in separate files (one per module). The browsers/engines have a default "module loader" (which is overridable, but that's well-beyond our discussion here) which synchronously loads a module file when it's imported.

Consider:

bar.js

```
function hello(who) {
    return "Let me introduce: " + who;
}

export hello;
```

foo.js

```
// import only `hello()` from the "bar" module
import hello from "bar";

var hungry = "hippo";
```

```
function awesome() {
    console.log(
        hello( hungry ).toUpperCase()
    );
}

export awesome;

// import the entire "foo" and "bar" modules
module foo from "foo";
module bar from "bar";

console.log(
    bar.hello( "rhino" )
); // Let me introduce: rhino

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

Note: Separate files **"foo.js"** and **"bar.js"** would need to be created, with the contents as shown in the first two snippets, respectively. Then, your program would load/import those modules to use them, as shown in the third snippet.

`import` imports one or more members from a module's API into the current scope, each to a bound variable (`hello` in our case). `module` imports an entire module API to a bound variable (`foo`, `bar` in our case). `export` exports an identifier (variable, function) to the public API for the current module. These operators can be used as many times in a module's definition as is necessary.

The contents inside the *module file* are treated as if enclosed in a scope closure, just like with the function-closure modules seen earlier.

Review (TL;DR)

Closure seems to the un-enlightened like a mystical world set apart inside of JavaScript which only the few bravest souls can reach. But it's actually just a standard and almost obvious fact of how we write code in a lexically scoped environment, where functions are values and can be passed around at will.

Closure is when a function can remember and access its lexical scope even when it's invoked outside its lexical scope.

Closures can trip us up, for instance with loops, if we're not careful to recognize them and how they work. But they are also an immensely powerful tool, enabling patterns like *modules* in their various forms.

Modules require two key characteristics: 1) an outer wrapping function being invoked, to create the enclosing scope 2) the return value of the wrapping function must include reference to at least one inner function that then has closure over the private inner scope of the wrapper.

Now we can see closures all around our existing code, and we have the ability to recognize and leverage them to our own benefit!