### Creating a Module

To understand what a Module can give you, you’ll need to understand what the following function concept does:

(function () {

// code

})();

It declares a function, which then calls itself immediately. These are also known as [Immediately-Invoked-Function-Expressions](http://benalman.com/news/2010/11/immediately-invoked-function-expression)’s, in which the function creates new scope and creates “privacy”. JavaScript doesn’t have privacy, but creating new scope emulates this when we wrap all our function logic inside them. The idea then is to return only the parts we need, leaving the other code out of the global scope.

After creating new scope, we need to namespace our code so that we can access any methods we return. Let’s create a namespace for our anonymous Module.

var Module = (function () {

// code

})();

We then have Module declared in the global scope, which means we can call it wherever we like, and even pass it into another Module.

### Private methods

You’ll see and hear a lot about private methods in JavaScript, but it doesn’t strictly have them, but we can create it.

What are private methods you might be asking? Private methods are anything you don’t want users/devs/hackers to be able to see/call outside the scope they’re in. We might be making server calls and posting sensitive data, we don’t want to expose those functions publicly, they could post anything back then and take advantage of our code. So we can create closure and be more sensible (as best as we can with JavaScript) at protecting our code. It’s not all about protection however, there are also naming conflicts. I bet when you first started out writing jQuery/JavaScript, that you dumped all your code in one file and it was just function, function, function. Little did you know these were all global, and you probably suffered the consequence at some point. If so, you’ll learn why, and what to do to change it.

So let’s use our newly created Module scope to make our methods inaccessible outside of that scope. For beginners to the Module Pattern, this example will help understand how a private method would be defined:

var Module = (function () {

var privateMethod = function () {

// do something

};

})();

The above example declares our function privateMethod, which is locally declared inside the new scope. If we were to attempt calling it anywhere outside of our module, we’ll get an error thrown and our JavaScript program will break! We don’t want anyone to be able to call our methods, especially ones that might manipulate data and go back and forth to a server.

### Understanding “return”

Typical Modules will use return and return an Object to the Module, to which the methods bound to the Object will be accessible from the Module’s namespace.

A real light example of returning an Object with a function as a property:

var Module = (function () {

return {

publicMethod: function () {

// code

}

};

})();

As we’re returning an Object Literal, we can call them exactly like Object Literals:

Module.publicMethod();

For those who haven’t used the Object Literal syntax before, a standard Object Literal could look something like this:

var myObjLiteral = {

defaults: { name: 'Todd' },

someMethod: function () {

console.log(this.defaults);

}

};

// console.log: Object { name: 'Todd' }

myObjLiteral.someMethod();

But the issue with Object Literals is we can abuse the pattern and bolt “private” methods onto it, which as they’re part of the Object, will be accessible too. This is where the Module comes in to save us, by allowing us to define all our private stuff locally and only return “the good parts”.

Let’s look at a more Object Literal syntax, and a perfectly good Module Pattern and thereturn keyword’s role. Usually a Module will return an Object, but how that Object is defined and constructed is totally up to you. Depending on the project and the role/setup of the code, I may use one of a few syntaxes.

#### Anonymous Object Literal return

One of the easiest patterns is the same as we’ve declared above, the Object has no name declared locally, we just return an Object and that’s it:

var Module = (function () {

var privateMethod = function () {};

return {

publicMethodOne: function () {

// I can call `privateMethod()` you know...

},

publicMethodtwo: function () {

},

publicMethodThree: function () {

}

};

})();

### Locally scoped Object Literal

Local scope means a variable/function declared inside a scope. On the [Conditionizr](https://conditionizr.com/)project, we use a locally scoped namespace as the file is over 100 lines, so it’s good to be able to see what are the public and private methods without checking the returnstatement. In this sense, it’s much easier to see what is public, because they’ll have a locally scoped namespace attached:

var Module = (function () {

// locally scoped Object

var myObject = {};

// declared with `var`, must be "private"

var privateMethod = function () {};

myObject.someMethod = function () {

// take it away Mr. Public Method

};

return myObject;

})();

You’ll then see on the last line inside the Module that myObject is returned. Our globalModule doesn’t care that the locally scoped Object has a name, we’ll only get the actualy Object sent back, not the name. It offers for better code management.

### Stacked locally scoped Object Literal

This is pretty much identical as the previous example, but uses the “traditional” single Object Literal notation:

var Module = (function () {

var privateMethod = function () {};

var myObject = {

someMethod: function () {

},

anotherMethod: function () {

}

};

return myObject;

})();

I prefer the second approach we looked at, the Locally scoped Object Literal. Because here, we have to declare other functions before we use them (you should do this, usingfunction myFunction () {} hoists your functions and can cause issues when used incorrectly). Using var myFunction = function () {}; syntax lets us not worry about this, as we’ll declare them all before we use them, this also makes debugging easier as the JavaScript interpreter will render our code in the order we declare, rather than hoistingfunction declarations. I also don’t like this approach so much, because the “stacking” method can often get verbose looking, and there is no obvious locally scoped Object namespace for me to bolt public methods onto.

### Revealing Module Pattern

We’ve looked at the Module, and there’s a really neat variant which is deemed the “revealing” pattern, in which we reveal public pointers to methods inside the Module’s scope. This again, can create a really nice code management system in which you can clearly see and define which methods are shipped back to the Module:

var Module = (function () {

var privateMethod = function () {

// private

};

var someMethod = function () {

// public

};

var anotherMethod = function () {

// public

};

return {

someMethod: someMethod,

anotherMethod: anotherMethod

};

})();

I really like the above syntax, as it’s very declarative. For bigger JavaScript Modules this pattern helps out a lot more, using a standard “Module Pattern” can get out of control depending on the syntax you go for and how you structure your code.

### Accessing “Private” Methods

You might be thinking at some stage during this article, “So if I make some methods private, how can I call them?”. This is where JavaScript becomes even more awesome, and allows us to actually invoke private functions via our public methods. Observe:

var Module = (function () {

var privateMethod = function (message) {

console.log(message);

};

var publicMethod = function (text) {

privateMethod(text);

};

return {

publicMethod: publicMethod

};

})();

// Example of passing data into a private method

// the private method will then `console.log()` 'Hello!'

Module.publicMethod('Hello!');

You’re not just limited to methods, though. You’ve access to Objects, Arrays, anything:

var Module = (function () {

var privateArray = [];

var publicMethod = function (somethingOfInterest) {

privateArray.push(somethingOfInterest);

};

return {

publicMethod: publicMethod

};

})();

### Augmenting Modules

So far we’ve created a nice Module, and returned an Object. But what if we wanted to extend our Module, and include another smaller Module, which extends our original Module?

Let’s assume the following code:

var Module = (function () {

var privateMethod = function () {

// private

};

var someMethod = function () {

// public

};

var anotherMethod = function () {

// public

};

return {

someMethod: someMethod,

anotherMethod: anotherMethod

};

})();

Let’s imagine it’s part of our application, but by design we’ve decided to not include something into the core of our application, so we could include it as a standalone Module, creating an extension.

So far our Object for Module would look like:

Object {someMethod: function, anotherMethod: function}

But what if I want to add our Module extension, so it ends up with another public method, maybe like this:

Object {someMethod: function, anotherMethod: function, extension: function}

A third method is now available, but how do we manage it? Let’s create an aptly namedModuleTwo, and pass in our Module namespace, which gives us access to our Object to extend:

var ModuleTwo = (function (Module) {

// access to `Module`

})(Module);

We could then create another method inside this module, have all the benefits of private scoping/functionality and then return our extension method. My pseudo code could look like this:

var ModuleTwo = (function (Module) {

Module.extension = function () {

// another method!

};

return Module;

})(Module || {});

Module gets passed into ModuleTwo, an extension method is added and then returnedagain. Our Object is getting thrown about, but that’s the flexibility of JavaScript :D

I can then see (through something like Chrome’s Dev Tools) that my initial Module now has a third property:

// Object {someMethod: function, anotherMethod: function, extension: function}

console.log(Module);

Another hint here, you’ll notice I’ve passed in Module || {} into my second ModuleTwo, this is incase Module is undefined - we don’t want to cause errors now do we ;). What this does is instantiate a new Object, and bind our extension method to it, and return it.

### Private Naming Conventions

I personally love the Revealing Module Pattern, and as such, I have many functions dotting around my code that visually are all declared the same, and look the same when I’m scanning around. I sometimes create a locally scoped Object, but sometimes don’t. When I don’t, how can I distinguish between private variables/methods? The \_ character! You’ve probably seen this dotted around the web, and now you know why we do it:

var Module = (function () {

var \_privateMethod = function () {

// private stuff

};

var publicMethod = function () {

\_privateMethod();

};

return {

publicMethod: publicMethod

};

})();