A Drip of JavaScript

Understanding the Module Pattern in JavaScript

Originally published in the A Drip of JavaScript newsletter.

Of all the design patterns you are likely to encounter in JavaScript, the module pattern is probably the most pervasive. But it can also look a little strange to developers coming from other languages.

Let's walk through an example to see how it works. Suppose that we have a library of utility functions that looks something like this:

```
var batman = {
    identity: "Bruce Wayne",

    fightCrime: function () {
        console.log("Cleaning up Gotham.");
    },

    goCivilian: function () {
        console.log("Attend social events as " + this.identity);
    }
};
```

This version of `batman` is perfectly serviceable. It can fight crime when you call upon it. However, it has a serious weakness. This `batman`'s `identity` property is publicly accessible.

Any code in your application could potentially overwrite it and cause `batman` to malfunction. For example:

```
// Some joker put this in your codebase
batman.identity = "a raving lunatic";

// Outputs: "Attend social events as a raving lunatic"
batman.goCivilian();
```

To avoid these sorts of situations we need a way to keep certain pieces of data private. Fortunately, JavaScript gives us just such a tool. We've even talked about it before: the immediately invoked function expression (IIFE).

A standard IIFE looks like this:

```
(function () {
    // Code goes here
})();
```

The advantage of the IIFE is that any `var`s declared inside it are inaccessible to the outside world. So how does that help us? The key is that an IIFE can have a return value just like any other function.

```
var batman = (function () {
    var identity = "Bruce Wayne";

    return {
        fightCrime: function () {
            console.log("Cleaning up Gotham.");
        },

        goCivilian: function () {
            console.log("Attend social events as " + identity);
        }
    };
```

```
})();

// Outputs: undefined
console.log(batman.identity);

// Outputs: "Attend social events as Bruce Wayne"
batman.goCivilian();
```

As you can see, we were able to use the IFFE's return value to make `batman`'s utility functions publicly accessible. And at the same time we were able to ensure that `batman`'s `identity` remains a secret from any clowns who want to mess with it.

You might be wondering when using the module pattern is a good idea. The answer is that it works well for situations like the one illustrated here. If you need to both enforce privacy for some of your data and provide a public interface, then the module pattern is probably a good fit.

It is worth considering, though whether you really need to enforce data privacy, or whether using a naming convention to indicate private data is a better approach. The answer to that question will vary on a case by case basis. But now you're equipped to enforce data privacy if necessary.

Thanks for reading!

Josh Clanton
