JAVASCRIPT

JavaScript Closures and the Module Pattern

2012/04/30

One of the most widely used design patterns in JavaScript is the module pattern. The module pattern makes use of one of the nicer features of JavaScript – closures – in order to give you some control of the privacy of your methods so that third party applications cannot access private data or overwrite it. In this post I'll teach you what a closure is, how it works, and how to utilize it to implement the module pattern in your own JavaScript code.

When you purchase anything through any of the above recommended links, you help support this blog. Thank you!

RECENT POSTS

GET STARTED WITH JAVASCRIPT ARRAYS

J Jens Look

Fashion Editorial photo
shoot in Ibiza with Jens Look. Behind the scenes videc

What is a Closure?

Closures are a construct of the JavaScript language. Within JavaScript all variables are accessible from the global scope except variables that are declared within a function using the `var` keyword.

```
1  variable1 = 1; // Global Sco
2  var variable2 = 2; // Not wi
3
4  function funcName() {
5      variable3 = 3; // No var
6      var variable4 = 4; // Lo
7  }
```

Within a function, you also have access to the global scope and every other scope above the function that you are in. In other words an inner function has

access to the variables that are within the function that wraps it.

```
1  var globalvar = 1; // Globa
2
3  function outer() {
4      var outervar = 2; // Sc
5
6      function inner() {
7          var innervar = 3; /
8          console.log(globalv
9          console.log(outerva
10         console.log(innerva
11     }
12
13     console.log(globalvar);
14     console.log(outervar);
15     console.log(innervar);
16 }
17
18 console.log(globalvar); //
19 console.log(outervar); // =
20 console.log(innervar); // =
```

Every real JavaScript programmer should know this if he or she wants to become great. Knowing this, you can come to the conclusion that there is a way to keep all of your code outside of the global namespace and you would be correct. This is especially helpful when you don't want to give anyone a chance to override any of your code without your permission. You can accomplish this by using an anonymous function (no name is given to it and it is not assigned to a variable) that

immediately executes itself. This is commonly known as a Self-Invoking Anonymous Function (SIAF), though it is probably more accurately referred to as an Immediately Invoked Function Expression (IIFE – pronounced "iffy") by Ben Alman.

```
1  (function() {
2      // This function immedia
3  }());
```

Right after the closing curly brace, just put an opening and closing parenthesis and the function will immediately be executed. The parentheses around the entire function expression aren't necessary for the code to run, but are generally used as a signal to other developers that this is an IIFE, not a standard function. Some people like to prefix with an exclamation point (!) or a semicolon (;), rather than wrapping the whole thing in parentheses.

Using Closures for the Module Pattern

Knowing what we know about closures, we can create objects using the module pattern. By returning an object or variable and assigning it to a variable

outside of the function, we can expose whatever we wish to the outside world, so we can have both public and private methods.

```
var Module = (function() {
    // Following function i
    function privateFunc() 

    // Return an object tha
    return {
        publicFunc: functio
            privateFunc(); 
        }
    };
}());
```

That's essentially the module pattern right there. You can also use the arguments to send in and shrink the name of commonly used assets:

```
var Module = (function($, w,
    // ...
    // return {...};
}(jQuery, window));
```

I sent in `jQuery` and `window`, which were abbreviated to `$` and `w`, respectively. Notice that I didn't send anything in for the third argument. This way `undefined` will be undefined, so it works perfectly. Some people do this with `undefined` because for whatever

reason it is editable. So if you check to see if something is `undefined`, but `undefined` has been changed, your comparison will not work. This technique ensures that it will work as expected.

The Revealing Module Pattern

The revealing module pattern is another way to write the module pattern that takes a bit more code, but is easier to understand and read sometimes. Instead of defining all of the private methods inside the IIFE and the public methods within the returned object, you write all methods within the IIFE and just "reveal" which ones you wish to make public within the `return` statement.

```
1  var Module = (function() {
2      // All functions now hav
3      var privateFunc = funct
4          publicFunc1();
5      };
6
7      var publicFunc1 = funct
8          publicFunc2();
9      };
10
11     var publicFunc2 = funct
12         privateFunc();
13     };
14
```

```
15      // Return the object tha
16      return {
17          publicFunc1: publicF
18          publicFunc2: publicF
19      };
20  }());
```

There are a few advantages to the revealing module pattern versus the normal module pattern:

1. All the functions are declared and implemented in the same place, thus creating less confusion.
2. Private functions now have access to public functions if they need to.
3. When a public function needs to call another public function they can call `publicFunc2()` rather than `this.publicFunc2()`, which saves a few characters and saves your butt if `this` ends up being something different than originally intended.

The only real downside to the revealing module pattern, as I said, is that you have to write a bit more code because you have to write the function and then write its name again in the `return` statement, though it could end up saving you code because you can skip the `this.` part.

Module Pattern for Extension

The last thing I wanted to talk about was using the module pattern for extending already-existing modules. This is done quite often when making plugins to libraries like jQuery, as you can see below.

```
1  (function($) {
2      $.pluginFunc = function(
3          ...
4      }
5  }(jQuery));
```

This code is pretty flexible because you don't even need the `var jQuery =` or the `return` statement near the end. jQuery will still be extended with the new method without them. It's actually probably bad for performance to return the entire jQuery object and assign it, however, if you want to assign jQuery to a new variable name at the same time that you're extending it, you can just change `jQuery` on the first line to whatever you want.

A Foregone Conclusion

That's all there is for today, friend. These are common techniques and features, so even if you don't make use of the knowledge from this post, keep it in the back of your mind just in case it comes up (which it probably will). Also, make sure to stop in again on Thursday to read about requestAnimationFrame: a new API coming out in browsers to make animations smoother and cleaner. Finally, don't forget to share and comment below. Thanks and Happy Coding!

Author: Joe Zimmerman

Joe Zimmerman has been doing web development ever since he found an HTML book on his dad's shelf when he was 12. Since then, JavaScript has grown in popularity and he has become passionate about it. He also loves to teach others though his blog and other popular blogs. When he's not writing code, he's spending time with his wife

and children and leading them in God's Word.

#JavaScript Comments Share
#design patterns #how to #plugin #security

We were unable to load Disqus. If you are a moderator please see our troubleshooting guide.