Structuring JavaScript Code in HTML5 Applications, Part 2

Use the Revealing Module Pattern to create structured, reusable JavaScript objects

Dan Wahlin

Oct 21, 2011

DOWNLOAD the CODE

As JavaScript continues to grow in popularity in HTML5 applications, it's important for developers to structure their code in a way that promotes reuse and allows for simplified maintenance. In my article last month, "Structuring JavaScript Code, Part 1," I discussed the standard technique used for structuring JavaScript code and talked about how closures can make things better through encapsulation. Figure 1 shows code from the previous article that's used to create a calculator, and Figure 2 shows the calculator in action. The code demonstrates the typical way to organize JavaScript variables and functions. Here I'll discuss an alternative to this coding structure that provides better encapsulation while eliminating variable and function-scoping issues.

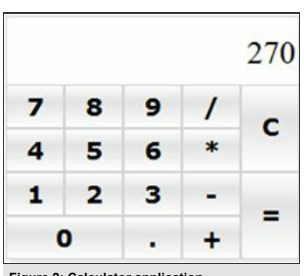

Figure 2: Calculator application

Dealing with Scoping Issues

Defining function after function in a JavaScript file certainly works but provides little in the way of encapsulation because functions aren't included in a container object. I refer to this style of coding as "function spaghetti code."

Looking through the code in Figure 1, you can easily see that functions aren't organized into any type of container object. Is the code really that bad, though -- after all, the functions get the job done, don't they? Although the functions can certainly be reused throughout a web application, variable scoping issues may come into play as other JavaScript files are included in an application, and function names may even be duplicated. This duplication can lead to variables changing unexpectedly as names collide. For example, if you create a variable named eqCtl in a JavaScript file, then include another file that also uses a variable named eqCtl, problems are likely to occur.

Although it doesn't have much structure, the code in Figure 1 does demonstrate a common technique for defining variables while saving a few keystrokes in the process. Toward the top of Figure 1, you'll see that multiple variables are defined, yet the var keyword is used only once:

```
var eqCtl,
    currNumberCtl,
    operator,
    operatorSet = false,
    equalsPressed = false,
    lastNumber = null;
```

Are the variables that follow eqCtl automatically placed in the global scope? Shouldn't the variables all have the var keyword applied to them, as shown next?

```
var eqCtl;
var currNumberCtl;
var operator;
var operatorSet = false;
var equalsPressed = false;
var lastNumber = null;
```

Because the variables aren't included in a closure, they're automatically placed in the global scope in this example, but that is to be expected. However, it's not because one var keyword is used. It's perfectly acceptable to define variables (inside or outside of a closure) with a single var keyword, then separate variable definitions with a comma. It is equivalent to placing the var keyword in front of each variable, as shown in the previous example. Although this technique is optional, you'll see that I use it throughout the code in this article.

Now that you've seen some "typical" JavaScript code, let's take a look at how the code can be organized into an object that keeps variables and functions out of the global scope while allowing functions to be exposed either publicly or privately. All of this can be achieved using a pattern called the Revealing Module Pattern. It is one of several patterns that can be used to provide more structure to JavaScript code and lead to the creation of more reusable objects. I'm a big fan of the pattern and used it throughout the Account at a Glance application that I built with my team.

Introducing the Revealing Module Pattern

The Revealing Module Pattern is based on a pattern called the Module Pattern. It makes code easier to read (in my opinion) and allows code to be organized in a more structured manner. The pattern starts with code similar to the following:

```
var Calculator = function () {
    /* Code goes here */
}();
```

You can see that a variable named Calculator is assigned to a function. Calculator represents the object that will be used in an application. The final parenthesis shown at the end of the code causes the function to be invoked immediately as the code loads.

Note: Some people prefer a lowercase name for Calculator because the *new* keyword isn't used to invoke it with the Revealing Module Pattern (as you'll see shortly). I prefer to use initial uppercase names for all my container objects. Ultimately it's up to you to decide on what conventions you'd like to follow in your JavaScript code.

Variables and functions that should be encapsulated within the Calculator object go in the "Code goes here" section. What's great about the pattern is that you can define members publicly or privately (as you can in C# or Visual Basic) even though JavaScript doesn't explicitly support that concept. This is done by adding a return statement at the end of the function that exposes the public members using a JavaScript object literal. Figure 3 shows an example of defining functions named myPrivateFunc() and add() along with a return object. This example exposes an add() function as a public member of the Calculator object while keeping the myPrivateFunc() function private to external callers.

```
var Calculator = function () {
    var myPrivateFunc = function() {
        /* private code */
    },
    add = function(x,y) {
        return x + y;
    };

    return {
        add: add
    };

}();
```

In this example, variables named myPrivateFunc and add are assigned to functions and placed within the Calculator object to create a closure (see "Structuring JavaScript Code, Part 1" if you're new to closures). Only the add variable will be exposed to outside callers of the Calculator object. This is accomplished by returning a JavaScript object literal from the Calculator function. The first value (the property name) in the return object is the name that callers will use, and the second (the value) represents the actual function they'll call. Although the two values are the same in this case, you can certainly change the name portion of the return object, if you want.

Now that you've seen a simple example, let's look at a more involved example of using the Revealing Module Pattern. The code in Figure 4 shows how the Calculator functionality shown earlier in Figure 1 can be refactored to follow the Revealing Module Pattern. Only the init, numberClick, setOperator, and clearNumbers functions are exposed publicly to outside callers.

Functions that are exposed publicly are defined in the return section of the Calculator object. In this example, the init, numberClick, setOperator, and clearNumbers functions are exposed by simply defining a JavaScript object literal that is returned when the main Calculator function is invoked. All the other functions and variables defined in the Calculator object are private. JavaScript doesn't support accessibility modifiers as C# or VB do, but this pattern provides a nice way to emulate that type of functionality.

Looking through the code, you may notice that a new function named init() was added; this function wasn't in Figure 1. This function is responsible for accepting any initialization data that the Calculator object needs for it to work correctly. As soon as the page loads, the Calculator object is created—but init() needs to be called to pass two HTML elements that it interacts with. The following code shows an example of calling init():

```
window.onload = function () {
    var eqCtl = document.getElementById('eq');
    var currNumberCtl = document.getElementById('currNumber');
    Calculator.init(eqCtl, currNumberCtl);
};
```

The eq and currNumber IDs referenced in the code can be found in this article's downloadable code samples. See the note at the end of the article for more information.

When the Calculator object shown in Figure 4 is initially parsed, the function is invoked right away, which creates one object in memory. What if you'd like to create and use multiple calculator objects on a single page? If you need to create multiple objects, you can do so using another technique with this pattern. Remove the final parenthesis that follows the Calculator object in Figure 4, then call the object, as shown in the following example:

```
var myCalc;
window.onload = function () {
    var eqCtl = document.getElementById('eq');
    var currNumberCtl = document.getElementById('currNumber');

    myCalc = Calculator(); //Invoke the object and assign to myCalc
    myCalc.init(eqCtl, currNumberCtl);
};
```

Once the Calculator function is invoked and assigned to the myCalc variable, the public functions exposed by the Calculator can be called through myCalc. The previous example uses myCalc to call init(). The myCalc variable is defined outside of onload, so that it can be accessed elsewhere in the script or within an HTML page.

A Useful Structuring Technique

The Revealing Module Pattern is currently one of my favorite JavaScript patterns for structuring JavaScript code because it's easy to use and very readable once you understand how it works (which is important for maintenance)—and it provides a simple way to expose public members to consumers while hiding private members. By using the Revealing Module Pattern, you can create reusable objects that avoid variable- and function-naming collisions, which is important when different JavaScript files are used in an application. To see additional examples of the Revealing Module Pattern in action, download the HTML5 Account at a Glance application. In "Tour the jQuery Framework," Mohammad Azam will introduce you to the jQuery framework, which provides developers with the ability to create animations, handle events, and develop AJAX applications with HTML5.

Related Articles

How to Structure JavaScript Code, Part 1

Structuring jQuery Ajax Calls in Web Applications 1

How to Build a jQuery HTML5 Web Application with Client-Side Coding

Build Enterprise-Scale JavaScript Applications with TypeScript

Using the HTML5 Canvas Tag