# How to Structure JavaScript Code, Part 1

**Techniques to encapsulate functionality in JavaScript similarly to using classes in C#**

Dan Wahlin

Sep 16, 2011

EMAIL SHARE Tweet G+1 Recommend 0 COMMENTS 0

DOWNLOAD the CODE

JavaScript has come a long way since the mid-90s when I first started working with it in Netscape 3 and Internet Explorer 3. Back in the day, I thought JavaScript was a bit painful to use, but over the years I've learned to love it and appreciate what it offers as a language. JavaScript is quite flexible and can perform a wide variety of tasks on both the client-side and server-side. In fact, I used to prefer it to VBScript on the server-side when writing classic ASP applications—and today we have server-side frameworks such as Node.js that are JavaScript based.

With the rise of HTML5 and new features such as the Canvas API and Scalable Vector Graphics (SVG), JavaScript is more important than ever when building applications. In "HTML5 Syntax and Semantics: Why They Matter," Michael Palermo and Daniel Egan explained the importance of using semantic elements in HTML5 applications. Similarly, as applications use more JavaScript, it's important that the code is structured in a way that's easy to work with and maintain to understand how to properly leverage JavaScript in your HTML5 applications.

Although JavaScript isn't designed with the concept of classes or object-oriented programming in mind, as with C# or Java, with a little work you can achieve similar results. In this article series I'll discuss popular patterns for structuring JavaScript to encapsulate functionality much as classes do, hide private members, and provide a better overall reuse strategy and maintenance story in applications. Several different JavaScript patterns exist, such as the Prototype Pattern, Revealing Module Pattern, and Revealing Prototype Pattern.

I'll use a calculator example throughout the articles to demonstrate different techniques that can be used for structuring JavaScript code. I decided on a calculator since it provides a simple starting point that everyone understands without needing a detailed explanation. An example of the calculator interface is shown in Figure 1.

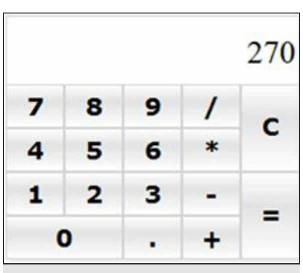


Figure 1: The calculator interface

I use specific add, subtract, multiply, and divide functions in the code that I will show mainly because I tend to avoid JavaScript's eval(), and I wanted to add enough functions to realistically demonstrate why taking the time to learn JavaScript code structuring techniques and patterns is worthwhile. In this first article in the series, I'll discuss the standard technique most people use when writing JavaScript code, examine the role of closures, and discuss different ways to define variables.

### Function Spaghetti Code

Most people (including myself) start out writing JavaScript code by adding function after function into a .js or HTML file. While there's certainly nothing wrong with that approach since it gets the job done, it can quickly get out of control when you're working with a lot of code. When lumping functions into a file, finding code can be difficult, refactoring code is a huge chore (unless you have a nice tool like JetBrains' ReSharper 6.0), variable scope can become an issue, and performing maintenance on the code can be a nightmare especially if you didn't originally write it. The code shown in Figure 2 demonstrates using the function-based approach to create a simple calculator.

Although this code can probably be refactored in some manner, you can see that it performs a few key calculator features, such as handling arithmetic operations, detecting when operators are selected, and performing calculations. Although everything shown in the code is standard JavaScript and works fine, as the number of functions grows things can quickly get out of hand. You can put the code in a file named calculator.js and then use it in as many pages as you'd like. However, if you come from an object-oriented language background, you'd probably like to encapsulate the functionality into the equivalent of a "class." Although classes aren't supported directly in JavaScript, you can emulate the functionality by using different types of patterns.

Another problem with this type of code is that any variables defined outside of functions are placed in the global scope by default. The script shown in Figure 2 adds six variables to the global scope (the functions get added as well, by the way). This means that they can more easily be stepped on or changed by anything in your script or another script that may be using the same variable names. It would be nice to localize the global variables and limit their scope to avoid variable and scope conflicts. Fortunately, that can be done using functions. However, if you define a variable in a function, it goes away after the function returns, right? That problem can be remedied by using closures, which are an important part of various JavaScript patterns.

### The Role of Closures

Most JavaScript patterns that allow code to be structured rely on an important technique called closures. Closures are important because they allow stateful objects to be created without relying on variables defined in the global scope. By using closures you can emulate features found in the class approach taken by object-oriented languages such as C# and Java and "modularize" your code.

A closure is created when a function has variables that are bound to it in such a way that even after the function has returned, the variables stick around in memory. So what's the magic that allows variables to be "bound" in such a way that they stick around even after a function returns? The answer is nested functions. When one function has a nested function inside of it, the nested function has access to the vars and parameters of the outer function and a "closure" is created behind the scenes. Douglas Crockford explains this with the following quote:

"What this means is that an inner function always has access to the vars and parameters of its outer function, even after the outer function has returned."

To better understand closures, examine the code shown in Figure 3, which represents a standard JavaScript function without any closures. When the myNonClosure function is invoked, the date variable will be assigned a new Date object. The function then returns the milliseconds. Calling the function multiple times will cause the date variable to be assigned a new value each time. This is, of course, the expected behavior.

```
function myNonClosure() {
    //variable will not be stored in a closure between calls
    //to the myNonClosure function
    var date = new Date();
    return date.getMilliseconds();
}
```

With a closure, a variable can be kept around even after a function returns a value. Figure 4 shows an example of a function named myClosure() that creates a closure. Looking through the code in Figure 4, you can see that a variable named date is assigned a Date object that is similar to the variable shown earlier.

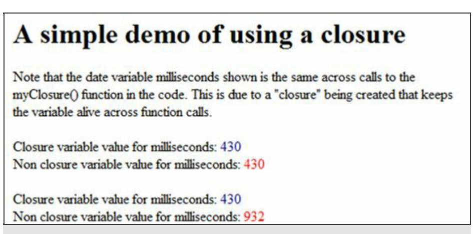
```
//closure example
function myClosure() {
    //date variable will be stored in a closure
    //due to the nested function referencing it
    var date = new Date();

    //nested function
    return function () {
        var otherDate = new Date();
        return "Closure variable value for milliseconds: " +
               " <span class='blue'>" +
                date.getMilliseconds() +
                "</span><br>Non closure variable value for " +
                "milliseconds: <span class='red'>" +
                otherDate.getMilliseconds() +
                "</span>";
    };
}
```

However, notice that myClosure returns a nested function that references the date variable. This creates a closure, causing the date variable to be kept around even after a value has been returned from the function. To see this in action, run the code in Figure 5.

```
window.onload = function () {
    //Using a closure
    var output = document.getElementById('Output'),
        closure = myClosure();
    output.innerHTML = closure();
    setTimeout(function() {
        output.innerHTML += '<br><br>' + closure();
    }, 1500);
};
```

The code in Figure 5 first references the myClosure() function and stores it in a variable named closure. The nested function is then called with the closure() call (note that the name "closure" could be anything; I chose it simply to make its purpose obvious), which invokes the function and returns the current milliseconds value. Next, a timeout is set to execute closure() again after 1.5 seconds have elapsed. Figure 6 shows the results of running the code. They demonstrate how the date variable is kept around even across multiple calls to the myClosure function. This is an important feature of JavaScript that is leveraged by the different JavaScript patterns.



A simple demo of using a closure

Note that the date variable milliseconds shown is the same across calls to the myClosure() function in the code. This is due to a "closure" being created that keeps the variable alive across function calls.

Closure variable value for milliseconds: 430
Non closure variable value for milliseconds: 430

Closure variable value for milliseconds: 430
Non closure variable value for milliseconds: 932

Figure 6: Example of how the values of variables used within nested functions are preserved even after functions return

Figure 7 shows a final example of a closure for you to study. It follows one of the popular JavaScript patterns being used—the Revealing Module Pattern, which I will discuss in detail in an upcoming article. Note that the myNestedFunc variable references a nested function that accesses the date variable.

```
var myClosure2 = function () {
    var date = new Date(),
        myNestedFunc = function () {
            return "Closure for myNestedFunc: " + date.getMilliseconds();
        };
    return {
        myNestedFunc: myNestedFunc
    };
} ();
```

This code is called using the following syntax:

```
output.innerHTML += '<br><br>'+ myClosure2.myNestedFunc();
```

### Getting Closure

In this article you've seen the standard way that JavaScript is normally written. You've also seen the role that closures play in JavaScript and how they can be used to create stateful objects. In "Structuring JavaScript Code in HTML5 Applications, Part 2," I'll discuss the Revealing Module Pattern and how you can use it along with the concept of closures to convert function spaghetti code into a more structured object that provides simplified maintenance and better reuse. If you'd like a sneak peek at the Revealing Module Pattern and other patterns I plan to discuss in the series, download the sample code available with this article.

PRINT REPRINT SAVE EMAIL SHARE Tweet G+1 Recommend 0

Please Log In or Register to post comments.

## Related Articles

Structuring JavaScript Code in HTML5 Applications, Part 2

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

Cross-Origin Resource Sharing for Windows Azure Cloud Apps

Structuring jQuery Ajax Calls in Web Applications 1

Build Enterprise-Scale JavaScript Applications with TypeScript

DevProConnections.com

Home Web Development Mobile Development Database Development Windows Development Azure Development Visual Studio

Site Features
Awards
Community Sponsors
Media Center
RSS
Sitemap
Site Archive
View Mobile Site

Penton
Privacy Policy
Terms of Service

Follow Us

Search DevProConnections.com

Dev Pro

Related Sites
SharePoint Pro SQL Server Pro SuperSite for Windows Windows IT Pro IT/Dev Connections myITforum