Encapsulation in JavaScript

While this is no longer late breaking news, it bears repeating: JavaScript is a powerful object oriented language, capable of being used to build sophisticated applications on both the client and the server. However, the more sophisticated the implementation, the bigger our responsibility to create maintainable and flexible code. In this article, I demonstrate how to use one of the pillars of object oriented programming, “encapsulation,” to help achieve these goals.

Encapsulation is the idea that the data of an object should not be directly exposed. Instead, callers that want to achieve a given result are coaxed into proper usage by invoking methods (rather than accessing the data directly).

The Problem

Let’s take a simple example where a “person” object contains a “fullName” attribute.

```javascript
var person = {
  "fullName": "Jason Shapiro",
};
alert(person.fullName); // Jason Shapiro
person.fullName = "Jim White";
alert(person.fullName); // Jim White
```

Everything looks OK, so far. We created the object, printed and changed the `fullName` without any issues. However, imagine that someone accidentally misuses this object, and sets the `fullName` to an invalid value, such as a number:

```javascript
person.fullName = 42;
alert(person.fullName); // 42
```

This is perfectly legal. As far as JavaScript is concerned, a variable can accept any type of data as its value. However, we may want to restrict the range of valid characters to satisfy how we use the `fullName` in our application. If we allow the caller to access and modify the data directly, we can’t force any validation logic to take place when they set the value. Furthermore, we may want to change how we store and/or construct the `fullName` value. Perhaps, at a later date, we decide that we want to break up the `fullName` variable into two separate variables: “firstName” and “lastName.” Again, allowing the caller direct access to the data is problematic: we can’t modify the internal data structure without breaking their accessor calls.
An Incomplete Solution

Let’s tackle the first issue (data validation) by modifying the object to contain a setFullName method. The goal of this method is to ensure no numeric characters are included when the name is set by the caller.

```
var person = {
    "fullName": "Jason Shapiro",
    "setFullName": function (newValue) {
        var reg = new RegExp(/d+/);
        if (reg.test(newValue)) {
            alert("Invalid Name");
        } else {
            this.fullName = newValue;
        }
    },
    "getFullName": function () {
        return this.fullName;
    }
};
alert(person.getFullName()); // Jason Shapiro
person.setFullName( "Jim White" );
alert(person.getFullName()); // Jim White
person.setFullName( 42 ); // Invalid Name
alert(person.getFullName()); // Jim White
```

Looks better. When the caller calls “setFullName” it ensures the name doesn’t contain a digit. But unfortunately, we’re only half way there. This strategy requires a wink and a nod from the caller ensuring that they promise not to call “fullName” directly. The code as written, however, won’t stop them from doing so:

```
person.setFullName( 42 ); // Invalid Name; the name is not changed.
person.fullName = 42; // No validation is executed; the name is changed and...
alert(person.getFullName()); // ...42 is printed.
```

The end goal here is to prevent callers from accessing fullName directly, which is where encapsulation comes into the picture. In order to hide our fullName data, we’ll need to learn two new concepts: function scope and closures.
Function Scope

Variables declared in a function are hidden from any code outside of its definition.

```
function scopeTest()
{
    var functionVariable = "Hello!";
    alert( functionVariable ) // "Hello!"
}
```

Therefore if we move the `fullName` variable inside of a function/method, callers won’t be able to invoke it directly. However, we can’t simply move the variable to the inside of the “setFullName” method; methods of an object can’t see each other’s local variables! In other words, if `fullName` was added to `setFullName`, it would not be available to any another method, such as `getFullName`. This is where closures help us out.

Closures

Simply put, a closure is an inner object which has access to all of the variables available in its parent function. So although methods of an object can’t see each other’s local variables, an inner object does have access to the local variables of its parent function. A bit of a confusing concept, so let’s see a code example. Here we have a function which is hiding the `fullName` variable from the outside world. The inner object (`theObj`), however, can access `fullName`:

```
var person = function () {
    var fullName = "Jason Shapiro";
    var reg = new RegExp(/\d+/);

    var theObj = {
        "setFullName" : function (newValue) {
            if( reg.test(newValue) ) {
                alert("invalid name");
            } else {
                fullName = newValue; // Legal! The object has access to "fullName"
            }
        },
        "getFullName" : function () {
            return fullName; // Legal! The object has access to "fullName"
        }
    }
    // End of the Object

    person.getFullName() // doesn't work!
}
```
We’re getting closer! We’ve created an anonymous function to hide the variables, and then defined the object as an inner object, in order to grant it access to the private variables. But how do we expose the inner object to the outside world? With the way the code is written in the above example, we can’t call “person.getFullName().” Nor for that matter could we call “person.theObj.getFullName().” since all of the variables, including “theObj” are private to the function. Thankfully the solution is simple: return the inner object when the anonymous function is called, and assign it to the outside variable (person).

```javascript
var person = function () {
  var fullName = "Jason Shapiro";
  var reg = new RegExp(/d+/);

  return {
    "setFullName" : function (newValue) {
      if (reg.test(newValue)) {
        alert("Invalid Name");
      } else {
        fullName = newValue;
      }
    },
    "getFullName" : function () {
      return fullName;
    }
  }; // end of the return

}(); // Note the '()', this means we're calling the function
// and using the *returned object,* instead of
// the function *itself* as the value of 'person.'

alert(person.getFullName()); // Jason Shapiro
person.setFullName( "Jim White" );
alert(person.getFullName()); // Jim White
person.setFullName( 42 ); // Invalid Name; the name is not changed.
person.fullName = 42; // Doesn't affect the private fullName variable.
alert(person.getFullName()); // Jim White is printed again.
```

The trick is to remember to call the anonymous function immediately after its definition, and assign the return value (the inner object) to the outside variable (person), rather than simply assigning the anonymous function itself to the outside variable. This is done by using the invocation operator, (), at the end of the function. Since this is easy to miss, and we want to strive for maximum readability, developers typically wrap the entire anonymous function & invocation operator with parentheses. While this isn’t required, it’s definitely recommended as a way to signal to other developers, “this function is being executed rather than simply assigned.”
4 var reg = new RegExp(/\d+/);
5 return {
6     "setFullName" : function (newValue) {
7         if (reg.test(newValue) ) {
8             alert("Invalid Name");
9         }
10     } else {
11         fullName = newValue;
12     },
13     "getFullName" : function () {
14         return fullName;
15     };
16     // end of the return
17 }();