How to Work with JavaScript's Prototyping Patterns

Create more extensible code using JavaScript's prototyping property

Dan Wahlin

Mar 18, 2012

RELATED: "HTML5 Tutorial: Build a Chart with JavaScript and the HTML5 Canvas" and "How to Structure JavaScript Code, Part 1"

JavaScript development provides several different patterns that can be used to structure code and make it more reusable, more maintainable, and less subject to naming collisions. Patterns such as the Revealing Module pattern (covered in "Structuring JavaScript Code in HTML5 Applications, Part 2"), Prototype pattern, Revealing Prototype pattern, and others can be used to structure code and avoid what I call "function spaghetti code." One of my favorite features offered by both the Prototype and the Revealing Prototype patterns is the extensibility they provide. They're quite flexible, especially compared to the Module or Revealing Module patterns, because of their use of JavaScript prototyping.

In this article I'll walk through the key concepts you need to know to get started with the JavaScript prototype property and explain the benefits it offers. Let's start by introducing two patterns that can be used to structure JavaScript code and see how they take advantage of the prototype property.

JavaScript Prototype Patterns

Both the Prototype and Revealing Prototype patterns rely on JavaScript prototyping. As a result, they offer several benefits over other patterns. For example, any functions defined are shared across object instances in memory. Another benefit is that for users of objects following these patterns, extending or overriding existing functionality is quite straightforward.

You can see examples of the Prototype and Revealing Prototype patterns in Figure 1 and Figure 2, respectively. Both patterns provide a way to structure JavaScript code in much the same way that classes structure code in object-oriented languages such as C# or Java.

```
var Calculator = function (eq) {
    //state goes here
    this.eqCtl = document.getElementById(eq);
};

Calculator.prototype = {
    add: function (x, y) {
        this.eqCtl.innerHTML = x + y;
    },
    subtract: function (x, y) {
        this.eqCtl.innerHTML = x - y;
    }
};
```

```
var Calculator = function (eq) {
    //state goes here
    this.eqCtl = document.getElementById(eq);
};

Calculator.prototype = function () {
    //private members
    var add = function (x, y) {
        this.eqCtl.innerHTML = x + y;
    },
    subtract = function (x, y) {
        this.eqCtl.innerHTML = x - y;
    };

    //public members
    return {
        add: add,
        subtract: subtract
    };
} ();
```

Looking through Figures 1 and 2, you can see that both patterns rely on the prototype functionality available in JavaScript. The Prototype pattern defines a constructor and assigns an object literal (the names and functions defined in the { } block of code) to the prototype property. The Revealing Prototype pattern also defines a constructor but assigns an anonymous function to the prototype property.

The Revealing Prototype pattern has the added functionality of being able to define public and private members within the Calculator because it returns an object literal. Any functions defined in the object literal that's returned from the anonymous function are publicly accessible, whereas ones that are not defined in the object literal are private and inaccessible by outside callers. (See the sidebar "Public/Private Member Functionality: A Question of Preference," at the end of this article, for my thoughts about this capability.)

What if you want to extend one of the Calculator objects shown in Figures 1 and 2 or override existing functions? When you use the Prototype or Revealing Prototype patterns, this is possible because both patterns rely on prototyping. We'll explore examples of these uses of prototyping in the following sections.

Getting Started with JavaScript Prototyping

Prototyping allows objects to inherit, override, and extend functionality provided by other objects, similarly to how inheritance, overriding, abstraction, and related technologies work in C#, Java, and other languages. By default, every object you create in JavaScript has a prototype property that can be accessed. To better understand prototyping, take a look at Figure 3. Rather than adding methods directly into the BaseCalculator constructor definition as we would do with a pattern such as the Revealing Module pattern, this example relies on separate prototype definitions to define two functions.

```
var BaseCalculator = function () {
    //Define a variable unique to each instance of BaseCalculator
    this.decimalDigits = 2;
};

//Extend BaseCalculator using prototype
BaseCalculator.prototype.add = function (x, y) {
    return x + y;
};

BaseCalculator.prototype.subtract = function (x, y) {
    return x - y;
};
```

The code in Figure 3 defines a BaseCalculator object with a variable named decimalDigits in the constructor. The code then extends the BaseCalculator object using the prototype property. Two functions are added: add() and subtract(). Both functions are defined using an anonymous function that accepts x and y parameters.

This type of definition can be simplified, as shown earlier with the Prototype pattern, by using an object literal to define the prototype functions -- see Figure 4.

```
var BaseCalculator = function() {
    //state goes here
    this.decimalDigits = 2;
};

BaseCalculator.prototype = {
    //private members
    add: function(x, y) {
        return x + y;
    },
    subtract: function(x, y) {
        return x - y;
    }
};
```

Once BaseCalculator is defined, you can inherit from it by doing the following:

```
var Calculator = function () {
    //Define a variable unique to each instance of Calculator
    this.tax = 5;
};

Calculator.prototype = new BaseCalculator();
```

Note that Calculator is defined with a constructor that includes a tax variable that's unique to each object instance. The Calculator's prototype points to a new instance of BaseCalculator(), allowing Calculator to inherit the add() and subtract() functions automatically. These functions are shared between both types and not duplicated in memory as instances are created, which is a nice feature provided by the prototype property.

Figure 5 shows an example of creating a new Calculator object instance.

```
var calc = new Calculator();
alert(calc.add(1, 1));

//variable defined in the BaseCalculator parent object is accessible from the deri
alert(calc.decimalDigits);
```

In Figure 5, BaseCalculator's decimalDigits variable is accessible to Calculator because a new instance of BaseCalculator was supplied to the Calculator's prototype. If you want to disable access to parent type variables defined in the constructor, you can assign BaseCalculator's prototype to Calculator's prototype, as shown next:

```
var Calculator = function () {
    this.tax= 5;
};

Calculator.prototype = BaseCalculator.prototype;
```

Because the BaseCalcuator's prototype is assigned directly to Calculator's prototype, the decimalDigits variable defined in BaseCalculator will no longer be accessible if you go through a Calculator object instance. The tax variable defined in Calculator would be accessible, of course. For example, the code in the following example will throw a JavaScript error when the code tries to access decimalDigits. This occurs because BaseCalculator's constructor is no longer being assigned to the Calculator prototype.

```
var calc = new Calculator();
alert(calc.add(1, 1));
alert(calc.decimalDigits);
```

Overriding with Prototype

If you're using either the Prototype pattern or Revealing Prototype pattern to structure code in JavaScript (or any other object that relies on prototyping), you can take advantage of the prototype property to override existing functionality provided by a type. This can be useful in scenarios where a library built by a third party is being used and you want to extend or override existing functionality without having to modify the library's source code. Or, you may write code that you want other developers on your team to be able to enhance or override. Here's an example of overriding the add() function provided in Calculator:

```
//Override Calculator's add() function
Calculator.prototype.add = function (x, y) {
    return x + y + this.tax;
};

var calc = new Calculator();
alert(calc.add(1, 1));
```

This code overrides the add() function provided by BaseCalculator and modifies it to add x, y, and an instance variable named myData together. The override applies to all Calculator object instances created *after* the override.

Working with *this*

JavaScript's *this* keyword can be a bit tricky to work with, depending on the context in which it's used. When *this* is used with patterns such as the Prototype or Revealing Prototype patterns, working with *this* can be challenging in some cases. Unlike in languages such as C# or Java, in JavaScript *this* can change context. For example, if a Calculator object named calc calls an add() function, *this* represents the Calculator object, which means you can easily access any variables defined in the constructor -- such as one named tax, by simply using this.tax. However, if add() makes a call to another function, *this* changes context and no longer represents the Calculator object. In fact, *this* will change to represent the window object, which means you can no longer access variables defined in the constructor of the Calculator object, such as tax.

There are several ways to handle this challenge. First, you can pass *this* as a parameter to other functions. Figure 6 shows an example of passing *this* between functions.

```
var Calculator = function (eq) {
    //state goes here
    this.eqCtl = document.getElementById(eq);
    this.lastNumber;
    this.equalsPressed;
    this.operatorSet;
};

Calculator.prototype = function () {
    //private members
    var add = function (x, y) {
        this.eqCtl.innerHTML = x + y;
    },
    subtract = function (x, y) {
        this.eqCtl.innerHTML = x - y;
    },
    setVal = function (val, thisObj) {
        thisObj.currNumberCtl.innerHTML = val;
    },
    setEquation = function (val, thisObj) {
        thisObj.eqCtl.innerHTML = val;
    },

    //Other functions omitted for brevity

    clearNumbers = function () {
        this.lastNumber = null;
        this.equalsPressed = this.operatorSet = false;
        setVal('0', this);
        setEquation('', this);
    };

    //public members
    return {
        add: add,
        subtract: subtract,
        clearNumbers: clearNumbers
    };
} ();
```

If a Calculator object calls a clearNumbers() function, you can easily access the Calculator object's constructor variables within the function. However, once clearNumbers() calls other functions such as setVal() or setEquation(), *this* changes context. To account for the change, the code in Figure 6 passes *this* as a parameter to each of the functions, and they then use it in the normal way. Although this type of code works, it pollutes your function parameters in some cases and becomes a little messy to work with (at least, in my opinion).

Another technique that can be used involves JavaScript's call() function. This function can be used to invoke functions and set the context of *this* while the call is being made. For example, if you want to call a function named setVal() and preserve the current value of *this* as the call is made, you can do the following:

```
setVal.call(this, 'yourParameterValue');
```

The current value of *this* will be passed along automatically to the setVal() function, and it can safely use this.tax in the case of a Calculator object.

Figure 7 uses the call() function to update the code shown in Figure 6.

```
var Calculator = function (eq) {
    //state goes here
    this.eqCtl = document.getElementById(eq);
    this.lastNumber;
    this.equalsPressed;
    this.operatorSet;
};

Calculator.prototype = function () {
    //private members
    var add = function (x, y) {
        this.eqCtl.innerHTML = x + y;
    },
    subtract = function (x, y) {
        this.eqCtl.innerHTML = x - y;
    },
    setVal = function (val) {
        this.currNumberCtl.innerHTML = val;
    },
    setEquation = function (val) {
        this.eqCtl.innerHTML = val;
    },

    //Other functions omitted for brevity

    clearNumbers = function () {
        this.lastNumber = null;
        this.equalsPressed = this.operatorSet = false;
        setVal.call(this, '0');
        setEquation.call(this, '');
    };

    //public members
    return {
        add: add,
        subtract: subtract,
        clearNumbers: clearNumbers
    };
} ();
```

The clearNumbers() function uses JavaScript's call() to invoke the setVal() and setEquation() functions and preserve the current value of *this* in the process. Notice that the setVal() and setEquation() functions no longer need the extra parameter as the functions shown in Figure 6 did and can simply use *this* to access Calculator object variables defined in the object's constructor. This simplifies the call by eliminating the need for the extra parameter and makes the code a lot cleaner compared to the code shown in Figure 6.

More Extensibility

There's more that you can do with JavaScript prototyping, so I recommend you read a post by Dmitry Soshnikov for additional details and more advanced examples of using the prototype property. Although the Prototype and Revealing Prototype patterns are just two of many patterns that are available to structure JavaScript code, if you're looking for JavaScript patterns that are extensible, they fit the bill nicely. For additional details about JavaScript prototyping as well as information about patterns that can be used to structure JavaScript code, check out my Structuring JavaScript Code course.

Related Articles

How to Structure JavaScript Code, Part 1
How to Build a jQuery HTML5 Web Application with Client-Side Coding
Improve Your JavaScript Coding with Data-Oriented Programming
Working with HTML5 Web Forms: Handling Dates and Other Input Types
Build Enterprise-Scale JavaScript Applications with TypeScript