JavaScript Patterns JumpStart Guide

Clean up your JavaScript Code

Dan Wahlin

JavaScript Patterns JumpStart Guide

Edited by Andrea Grimm

About the Author

Welcome to the JavaScript Patterns JumpStart Guide! My name is Dan Wahlin and I'll be walking you through a lot of great information that will change how you write JavaScript and result in more maintainable applications. I've been building JavaScript applications since the 90s and love writing about and working with the language. If you're interested in keeping up with all of the great enhancements in the JavaScript language as well as various frameworks, check out my blog or follow me on Twitter.

Blog
http://weblogs.asp.net/dwahlin

Twitter
@DanWahlin

Table of Contents

About the Author

Sample Code

Introduction

- Function Spaghetti Code
- What are Closures?
- Namespaces and Global Scope
- Defining Variables

The Prototype Pattern

Revealing Module Pattern

Creating Multiple Objects with the Revealing Module Pattern

Revealing Prototype Pattern

Working with the JavaScript *this* Keyword

Using the JavaScript *prototype* Property to Structure and Extend Code

- Getting Started with JavaScript Prototyping
- Overriding with Prototype

Conclusion

Sample Code

A lot of effort went into formatting the code shown in this book to make it easy to read and follow. However, different devices display text in different ways so I've made the book's sample code available at http://tinyurl.com/StructuringJSCode so that you can view it in your favorite editor as well.

Introduction

JavaScript has come a long way since the mid-90s when I first started working with it in Netscape Navigator. Back in the day I thought JavaScript was 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 SVG, JavaScript is more important than ever when building applications. As applications use more JavaScript, it's important the code is structured in a way that's easy to work with and maintain.

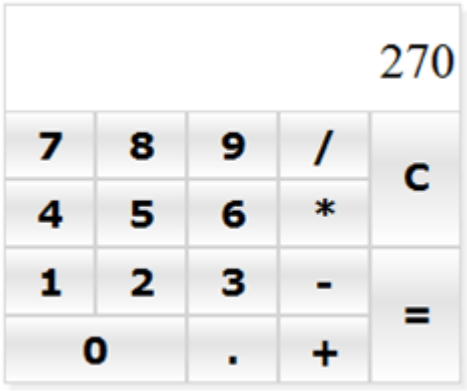
Although JavaScript isn't designed with the concept of classes or object-oriented programming in mind as with C# or Java (at least not until ECMAScript 6 comes out and is supported by all of the browsers), with a little work you can achieve similar results. In this guide I'll discuss a few popular techniques, strategies, and patterns for structuring JavaScript to encapsulate functionality much like classes do, hide private members, and provide a better overall re-use strategy and maintenance story in applications. These techniques work well across all of the major browser out there.

The patterns that will be discussed include the following:

- Prototype Pattern
- Revealing Module Pattern
- Revealing Prototype Pattern

Are there more patterns? Certainly! However, the goal of this jumpstart guide to focus on some of the commonly used patterns that are out there.

I'll use truck and calculator examples throughout the sections that follow to demonstrate different patterns that can be used for structuring JavaScript code. The truck will be used to show basic code and break things down in simple terms while the calculator is more robust and will have more code for you to study. I decided on a calculator since it provides a simple starting point that everyone understands without needing a detailed explanation, while also providing enough complexity to demonstrate something more realistic. An example of the calculator interface is shown next.



In this first section we'll look at the technique most people use when writing JavaScript code, examine the role of closures, review namespaces and global scope, and discuss different ways of defining variables. Let's start off by looking at some typical "function spaghetti code".

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 working with a lot of code. When lumping functions into a file, finding code can be difficult, refactoring code is a huge chore, variable scope can become an issue, and performing maintenance on the code can be a nightmare, especially if you didn't originally write it.

Let's start out with a simple example. The following code defines a variable and some functions a truck might have:

```
var engine = 'V8';

function start() {
    alert('Truck started engine ' + engine);
}

function stop() {
    alert('Truck stopped');
}
```

Pretty simple right? We have a single *engine* variable, a *start()* function that uses the *engine* variable, and a *stop()* function. This is how a lot of JavaScript applications start out. They're quite simple in the beginning but quickly grow out of control. What happens if another part of the application includes a script that defines an *engine* variable or a *start()* or *stop()* function? That's when bad things start to happen.

Let's look at another example that's a little more robust. The following code sample demonstrates using the function based approach to create a simple calculator. It's not perfect, but it gets the job done for the pattern samples that will follow.

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

function add(x, y) {
    return x + y;
}

function subtract(x, y) {
    return x - y;
}
```

```
function multiply(x, y) {
    return x * y;
}

function divide(x, y) {
    if (y == 0) {
        alert("Can't divide by 0");
        return 0;
    }
    return x / y;
}

function setVal(val) {
    currNumberCtl.innerHTML = val;
}

function setEquation(val) {
    eqCtl.innerHTML = val;
}

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

function setOperator(newOperator) {
    if (newOperator == '=') {
        equalsPressed = true;
        calculate();
        setEquation('');
        return;
    }

    //Handle case where = was pressed
    //followed by an operator (+, -, *, /)
    if (!equalsPressed) calculate();
    equalsPressed = false;
    operator = newOperator;
    operatorSet = true;
    lastNumber =
      parseFloat(currNumberCtl.innerHTML);
    var eqText = (eqCtl.innerHTML == '') ?
        lastNumber + ' ' + operator + ' ' :
        eqCtl.innerHTML + ' ' + operator + ' ';
    setEquation(eqText);
}

function numberClick(e) {
    var button = (e.target) ? e.target : e.srcElement;
    if (operatorSet == true ||
        currNumberCtl.innerHTML == '0') {
        setVal('');
        operatorSet = false;
```

```
    }
    setVal(currNumberCtl.innerHTML +
           button.innerHTML);
    setEquation(eqCtl.innerHTML +
                button.innerHTML);
}

function calculate() {
    if (!operator || lastNumber == null) return;
    var currNumber = parseFloat(currNumberCtl.innerHTML),
        newVal = 0;
    switch (operator) {
        case '+':
            newVal = add(lastNumber,
                         currNumber);
            break;
        case '-':
            newVal = subtract(lastNumber,
                              currNumber);
            break;
        case '*':
            newVal = multiply(lastNumber,
                              currNumber);
            break;
        case '/':
            newVal = divide(lastNumber,
                            currNumber);
            break;
    }
    setVal(newVal);
    lastNumber = newVal;
}
```

Although this code can probably be refactored in some manner or another, you can see 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 you'd probably like to encapsulate the functionality into the equivalent of a "class". Although classes aren't supported directly in JavaScript (at least not until ECMAScript 6), you can emulate the functionality 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 calculator script shown above adds 6 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'd 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 the JavaScript patterns that you'll be introduced to in this guide.

What are Closures?

JavaScript patterns rely on a key concept in JavaScript 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.

A closure is created when a function has variables 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 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.

To better understand closures, examine the following code representing a standard JavaScript function without any closures:

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

}
```

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.

With a closure, a variable can be kept around even after a function returns a value. An example of a function named *myClosure()* that creates a closure is shown next.

```
//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: " +
       date.getMilliseconds() +
       "<br>Non closure variable " +
       "value for milliseconds: " +
       otherDate.getMilliseconds();
    };
```

```
}
```

Looking through the code, you can see that a variable named *date* is assigned a *Date* object, which is similar to the variable shown earlier. However, notice that *myClosure* returns a nested function, which 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, the following code can be run:

```
window.onload = function () {
    //Using a closure
    var output =
        document.getElementById('Output'),
        closure = myClosure();

    output.innerHTML = closure();

    setTimeout(function () {
        output.innerHTML +=
          '<br><br>' + closure();
    }, 1500);

};
```

Here's a step-by-step look at what the code is doing:

1. The code first references the *myClosure()* function and stores it in a variable named *closure*.
2. The nested function is then called with the *closure()* call, which invokes the function and returns the current milliseconds. Note that the name "closure" could be anything. I chose it simply to make its purpose more obvious.
3. A timeout is set to execute *closure()* again after 1.5 seconds have elapsed.

The results of running the code are shown next. 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 patterns that will be shown.

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

Here's a final example of a closure that follows one of the patterns discussed later in this guide. Note that the *myNestedFunc* variable references a nested function that accesses the *date* variable.

```
var myClosure2 = function () {
    var date = new Date(),

    myNestedFunc = function () {
       return date.getMilliseconds();
    };

    return {
        myNestedFunc: myNestedFunc
    };

}();
```

This code is called using the following syntax:

```
output.innerHTML += myClosure2.myNestedFunc();
```

Namespaces and Global Scope

JavaScript variables and functions are added to the global scope by default. Although that's not necessarily a big deal for smaller applications, it quickly becomes more challenging with larger applications and is something you should avoid in general. For example, if you define a variable named *total* in code using the JavaScript *var* keyword it goes into the global scope by default and becomes a property of the *window* object:

```
var total = 50;
alert(window.total);
```

So what's the problem with this code? If another globally defined variable named *total* appears later in the code it will overwrite the previous one. This includes variables defined in scripts that team members contribute or 3rd party scripts used in an application.

The patterns discussed in this guide rely on functions and closures to shield variables and functions from the global scope. However, some variables or functions will still ultimately be associated with the *window* object. How do you fix the problem and prevent code from stepping on other code? There are a few solutions available such as using anonymous container functions and defining namespaces.

One technique that can be used to shield variables and functions from the global scope is to wrap the variables and functions with an anonymous function that is self-invoked as shown next:

```
//Take variables and functions
//out of the global scope by
//wrapping them in an outer function
(function () {

    var total = 50;
    alert(total);
    alert(window.total);

})();
```

In this example the *total* variable will be taken out of the global scope due to the anonymous function container. Running the code will result in the first alert showing a value of *50* while the second alert will show *undefined* since *total* isn't in the global scope and associated with the *window* object.

Another technique that can be used to take variables and functions out of the global scope is namespaces. A namespace is an object that acts as a container for various members such as properties and functions (JavaScript doesn't officially support modules or namespaces – at least until ECMASCript 6 - but we can fake it). An example of defining a namespace and adding a property and functions into it is shown next:

```
//A namespace is in the global
//scope but acts as a container
//for variables and functions
var acmeCorp = acmeCorp || {};

//Add properties and functions
//into the namespace
acmeCorp.engine = 'V8';

acmeCorp.start = function () {
    alert('Truck started ' + acmeCorp.engine);
};

acmeCorp.stop = function () {
    alert('Truck stopped');
};

acmeCorp.start();
acmeCorp.stop();
alert(window.engine); //Will be undefined
```

In this example a new namespace named *acmeCorp* is created. The code first checks to see if it has already been defined elsewhere in the code. If it already exists the value is assigned to the *acmeCorp* variable. If it doesn't exist then an empty object literal container is created using the {} syntax. From there an *engine* property along with *start()* and *stop()* functions are added into the *acmeCorp* namespace.

When the code is run the first alert will show "Truck started V8" and the second alert will show "Truck stopped". The final alert will display *undefined* since the *window* object doesn't have a

property named engine. By creating a namespace, property and function definitions can be taken out of the global scope.

In the previous code the *acmeCorp* namespace is still in the global scope which may or may not be appropriate for a given application (as mentioned, it should be avoided whenever possible). To take *acmeCorp* out of the global scope you can wrap it with an anonymous function:

```
(function () {

    var acmeCorp = acmeCorp || {};
    acmeCorp.engine = 'V6';

    acmeCorp.start = function () {
        alert('Truck started ' +
              acmeCorp.engine);
    };

    acmeCorp.stop = function () {
        alert('Truck stopped');
    };

    acmeCorp.start();
    acmeCorp.stop();
    alert(window.engine); //Will be undefined

})();
```

Defining Variables

Before jumping into JavaScript patterns, let's wrap up this section by taking a quick look at variables. Defining variables in JavaScript is one of the more simple aspects of the language, of course, but there are a few different ways to do it. For example, the following code is completely valid and what most people getting started with JavaScript do:

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

Although this code works fine, tools such as JSLint (http://jslint.com) will tell you to define the variables differently. Here's one of the messages JSHint may show:

```
var operator;
```

Combine this with the previous 'var' statement.

There's another option that can be used that will clean up the code somewhat. In the patterns that follow, you'll see code similar to the following when defining variables:

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

This code only uses the JavaScript *var* keyword once then separates variables with a comma. The code ultimately does the same thing as the previous example with variables, but it reduces the size of the script and is more readable once you get used to it.

Now that you've seen how to work with closures and different options for defining variables, let's get started with the process of cleaning up JavaScript code by applying patterns.

The Prototype Pattern

In the previous section, you were introduced to the "function spaghetti code" concept and some of the problems it introduces. In this section, you'll learn about the Prototype Pattern and how it relies on built-in functionality in the JavaScript language—namely the prototype property.

The Prototype Pattern can be broken out into two main sections including a constructor and a prototype. Prototyping allows functions and properties to be associated with objects. However, instead of each object instance getting a copy of all functions/properties each time an object is created, only one set of functions/properties exists across all objects, resulting in less memory consumption. In other words, functions and properties are defined once per prototype rather than once per object.

Let's review the truck code shown earlier:

```
var engine = 'V8';

function start() {
    alert('Truck started engine ' +
           engine);
}

function stop() {
    alert('Truck stopped');
}
```

To convert this code to use the Prototype Pattern you'll first define a constructor and put the *engine* variable inside of it:

```
var Truck = function (engine) {
    this.engine = engine;
};
```

Once the constructor is defined a prototype needs to be created. All of the functions go inside of an object literal assigned to the prototype. Any references by functions in the prototype to properties defined in the constructor must use the *this* keyword:

```
Truck.prototype = {

    start: function() {
        alert('Car started engine ' +
              this.engine);
    },

    stop: function() {
        alert('Car stopped');
```

```
    }

};
```

Prototype functions are shared across all instances of objects so they're really efficient on memory usage. Here's the final truck code converted to the Prototype Pattern:

```
var Truck = function (engine) {
    this.engine = engine;
};

Truck.prototype = {
    start: function() {
        alert('Car started engine ' +
              this.engine);
    },

    stop: function() {
        alert('Car stopped');
    }
};
```

Here's another example of the Prototype Pattern being applied to the calculator functionality shown earlier. To start using the Prototype Pattern you need to first create a constructor for the calculator and define properties inside of it:

```
var Calculator = function (tb, eq) {
    this.eqCtl =
      document.getElementById(eq);
    this.currNumberCtl =
      document.getElementById(tb);
    this.operator = null;
    this.operatorSet = false;
    this.equalsPressed = false;
    this.lastNumber = null;

};
```

The constructor can accept one or more parameters and define any variables the object needs. Note that the variables are scoped to the object, rather than to the global scope. The only thing in the global scope at this point is the *Calculator* variable which can be taken out of the global scope using an anonymous function and/or namespace as discussed earlier:

Once the constructor is defined, a prototype can be created using the *prototype* keyword. Although you can create a prototype for each function, it's more convenient (and less typing) to take advantage of object literal syntax, where the function name represents the property name and the value represents the function. An example of defining two functions in a prototype is shown next.

Notice that each function is separated with a comma.

```
Calculator.prototype = {
    add: function (x, y) {
        return x + y;
    },

    subtract: function (x, y) {
        return x - y;
    }
}
```

The original function-based calculator code shown earlier can be refactored to follow the Prototype Pattern as shown next. A prototype is created for the *Calculator* object and functions/properties are defined within the prototype using JavaScript object literal syntax.

```
Calculator.prototype = {
    add: function (x, y) {
        return x + y;
    },

    subtract: function (x, y) {
        return x - y;
    },

    multiply: function (x, y) {
        return x * y;
    },

    divide: function (x, y) {
        if (y == 0) {
            alert("Can't divide by 0");
            return 0;
        }
        return x / y;
    },

    setVal: function (val) {
        this.currNumberCtl.innerHTML = val;
    },

    setEquation: function (val) {
        this.eqCtl.innerHTML = val;
    },

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

```
setOperator: function (newOperator) {
    if (newOperator == '=') {
        this.equalsPressed = true;
        this.calculate();
        this.setEquation('');
        return;
    }

    //Handle case where = was pressed
    //followed by an operator (+, -, *, /)
    if (!this.equalsPressed) this.calculate();
    this.equalsPressed = false;
    this.operator = newOperator;
    this.operatorSet = true;
    this.lastNumber = parseFloat(
      this.currNumberCtl.innerHTML);
    var eqText = (this.eqCtl.innerHTML == '') ?
                 this.lastNumber + ' ' +
                 this.operator + ' ' :
                 this.eqCtl.innerHTML + ' ' +
                 this.operator + ' ';
    this.setEquation(eqText);
},

numberClick: function (e) {
    var button = (e.target) ? e.target : e.srcElement;
    if (this.operatorSet == true ||
        this.currNumberCtl.innerHTML == '0') {
        this.setVal('');
        this.operatorSet = false;
    }
    this.setVal(this.currNumberCtl.innerHTML +
      button.innerHTML);
    this.setEquation(this.eqCtl.innerHTML +
      button.innerHTML);
},

calculate: function () {
    if (!this.operator ||
        this.lastNumber == null) return;
    var displayedNumber =
       parseFloat(this.currNumberCtl.innerHTML),
       newVal = 0;
    switch (this.operator) {
        case '+':
            newVal = this.add(this.lastNumber,
                              displayedNumber);
            break;
        case '-':
            newVal = this.subtract(this.lastNumber,
                                   displayedNumber);
            break;
        case '*':
            newVal = this.multiply(this.lastNumber,
                                   displayedNumber);
```

```
                break;
            case '/':
                newVal = this.divide(this.lastNumber,
                                     displayedNumber);
                break;
        }
        this.setVal(newVal);
        this.lastNumber = newVal;
    }
};
```

To use the *Calculator* object, create a new instance and pass the names of the HTML container objects into the constructor (the container objects identify the IDs of controls used to display calculations—see the HTML that follows below):

```
var calc = new Calculator('currNumber', 'eq');
```

The HTML used to render the calculator can reference the *calc* object created when the page loads. The following code demonstrates how this can be done to handle events as different div elements are clicked.

```
<div class="Calculator">
    <div class="CalculatorHead">
        <div id="eq"></div>
        <div id="currNumber">0</div>
    </div>
    <div>
        <div onclick="calc.numberClick(event);">7</div>
        <div onclick="calc.numberClick(event);">8</div>
        <div onclick="calc.numberClick(event);">9</div>
        <div onclick="calc.setOperator('/');">/</div>
        <div onclick="calc.clearNumbers();">C</div>
    </div>
    <div>
        <div onclick="calc.numberClick(event);">4</div>
        <div onclick="calc.numberClick(event);">5</div>
        <div onclick="calc.numberClick(event);">6</div>
        <div onclick="calc.setOperator('*');">*</div>
    </div>
    <div>
        <div onclick="calc.numberClick(event);">1</div>
        <div onclick="calc.numberClick(event);">2</div>
        <div onclick="calc.numberClick(event);">3</div>
        <div onclick="calc.setOperator('-')">-</div>
        <div onclick="calc.setOperator('=');">=</div>
    </div>
    <div>
        <div onclick="calc.numberClick(event);">0</div>
        <div onclick="calc.numberClick();">.</div>
        <div onclick="calc.setOperator('+');">+</div>
```

```
    </div>
</div>
```

Note: If you're using a library such as jQuery you'd probably wire-up the click event to a function using code. To keep the demonstration simple, the *onclick* attribute is added directly to the different elements.

The Prototype Pattern provides a nice way to structure JavaScript code, but there are several other roads you can travel, if desired. In the next section, you'll be introduced to the Revealing Module Pattern and see how it can be used to structure JavaScript code.

Revealing Module Pattern

The Prototype Pattern discussed earlier works well and is quite efficient, but it's not the only game in town. One of my favorite JavaScript patterns is the Revealing Module Pattern since its cleaner (in my opinion) and has less reliance on the JavaScript *this* keyword. I also like the fact it doesn't separate code into constructor and prototype sections.

Although it doesn't offer the benefit of sharing function implementations across objects through JavaScript's prototype feature, it's definitely a viable option. I typically use it for scripts I refer to as "page scripts". These are scripts that have the specific purpose of controlling functionality in a given page as opposed to a component, library, or control script.

The Revealing Module Pattern is based on a pattern referred to as the Module Pattern. It makes reading code easier and allows it to be organized in a more structured manner. The pattern starts with code like the following to define a variable, associate it with a function, and then invoke the function immediately as the script loads. The final parentheses shown at the end of the code cause it to be invoked.

```
var truck = function () {

    /* Code goes here */

}();
```

Here's an example of the truck code shown earlier converted to follow the Revealing Module Pattern. Notice that the self-invoking parentheses are used to invoke the function and pass a value of *V8* into it:

```
var truck = function (eng) {
    var engine = eng,
    start = function () {
        alert('Truck started ' + engine);
    },
    stop = function () {
        alert('Truck stopped');
    };

}('V8');
```

All of the members inside of the *Truck* object are private by default so they're not accessible at this point. To make them accessible, return an object literal that maps public names to private names. Here's an example of exposing two public members named *start* and *stop* that map to the private *start()* and *stop()* functions:

```
var truck = function (eng) {
    //private members
    var engine = eng,
    start = function () {
```

```
        alert('Truck started ' + engine);
    },
    stop = function () {
        alert('Truck stopped');
    };

    //public members
    return {
        start: start,
        stop: stop
    };
}('V8');
```

How do you use the *truck* object? Because the function is self-invoked, you can call it directly:

```
truck.start();
```

Now that you've seen how the truck code can be converted to the Revealing Module Pattern, let's take a look at how the calculator code looks once converted to the pattern:

```
var calculator = function() {
    var eqCtl,
    currNumberCtl,
    operator,
    operatorSet = false,
    equalsPressed = false,
    lastNumber = null,

    init = function(equals, currNumber) {
        eqCtl = equals;
        currNumberCtl = currNumber;
    },

    add = function(x, y) {
        return x + y;
    },

    subtract = function(x, y) {
        return x - y;
    },

    multiply = function(x, y) {
        return x * y;
    },

    divide = function(x, y) {
        if (y == 0) {
            alert("Can't divide by 0");
            return 0;
```

```
    }
    return x / y;
},

setVal = function(val) {
    currNumberCtl.innerHTML = val;
},

setEquation = function(val) {
    eqCtl.innerHTML = val;
},

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

setOperator = function(newOperator) {
    if (newOperator == '=') {
       equalsPressed = true;
       calculate();
       setEquation('');
       return;
    }

    //Handle case where = was pressed
    //followed by an operator (+, -, *, /)
    if (!equalsPressed) calculate();
    equalsPressed = false;
    operator = newOperator;
    operatorSet = true;
    lastNumber = parseFloat(
      currNumberCtl.innerHTML);
    var eqText = (eqCtl.innerHTML == '') ?
        lastNumber + ' ' + operator + ' ' :
        eqCtl.innerHTML + ' ' + operator + ' ';
    setEquation(eqText);
},

numberClick = function(e) {
    var button = (e.target) ? e.target : e.srcElement;
    if (operatorSet == true ||
        currNumberCtl.innerHTML == '0') {
        setVal('');
        operatorSet = false;
    }
    setVal(currNumberCtl.innerHTML +
      button.innerHTML);
    setEquation(eqCtl.innerHTML +
      button.innerHTML);
},

calculate = function() {
```

```
        if (!operator ||
            lastNumber == null) return;
        var currNumber = parseFloat(
              currNumberCtl.innerHTML),
            newVal = 0;
        switch (operator) {
        case '+':
            newVal = add(lastNumber,
                         currNumber);
            break;
        case '-':
            newVal = subtract(lastNumber,
                              currNumber);
            break;
        case '*':
            newVal = multiply(lastNumber,
                              currNumber);
            break;
        case '/':
            newVal = divide(lastNumber,
                            currNumber);
            break;
        }
        setVal(newVal);
        lastNumber = newVal;
    };

    return {
        init: init,
        numberClick: numberClick,
        setOperator: setOperator,
        clearNumbers: clearNumbers
    };
}();
```

In this example the *init()*, *numberClick()*, *setOperator()*, and *clearNumbers()* functions are exposed publicly by simply defining a JavaScript object literal that is returned when the main *calculator* function is invoked. All of the other functions and variables defined in the *calculator* object are private by default. JavaScript doesn't support accessibility modifiers like C# or Java, but this pattern provides a nice way to emulate that type of functionality. It would be perfectly acceptable to prefix the private members with a "_" character if desired. This is a naming convention used by some developers to signify that a variable or function is private.

Looking through the code, you may notice a new function named *init()* was added that wasn't in the previous examples. It's responsible for accepting any initialization data the *calculator* object needs 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. An example of calling *init()* is shown next:

```
var eqCtl = document.getElementById('eq'),
    currNumberCtl =
      document.getElementById('currNumber');

calculator.init(eqCtl, currNumberCtl);
```

It's important to note the parentheses at the end of the calculator object's code causes the function to be immediately invoked as mentioned earlier. This creates a singleton—a single object in memory. That's why the *init()* function can be called directly as opposed to explicitly instantiating *calculator*.

Why is the *calculator* object defined with a lower-case "c" instead of an upper-case "C" as with the Prototype Pattern? Why was the *truck* object shown earlier defined with a lower-case "t"? A general rule is to make the first letter in an object's name lower-case if the object doesn't have to be instantiated and to make the first letter upper-case if it needs to be instantiated. By following that convention, it's easy to look at an object and know how to call it properly. This type of naming convention isn't a hard and fast rule that has to be followed, but it's something that is recommended.

The Revealing Module Pattern is one of my favorite patterns for structuring JavaScript code mainly because it's easy to use, very readable (which is important for maintenance), and provides a simple way to expose public members to consumers. It'd be nice if you could combine the Revealing Module Pattern with the Prototype Pattern to get the benefits provided by prototyping while also having the ability to define public/private members. The good news is you can do that with the Revealing Prototype Pattern, which you'll learn about a little later. Before jumping into that pattern, let's first talk about how the Revealing Module Pattern can be used to create multiple instances of an object.

Creating Multiple Objects with the Revealing Module Pattern

In the previous section, you saw how the Revealing Module Pattern can be used to structure JavaScript code while also allowing members of an object to be made public or private. A question that often comes up is, "Can the Revealing Module Pattern handle multiple instances on a page?"

With the prototype pattern, you simply "new" up as many instances as you need. But based on the code shown in the previous section, the Revealing Module Pattern looks like it's limited to a single instance per page. Looking back at the *truck* and *calculator* objects and the way they're self-invoked, it would definitely appear that you can only create one instance per page:

```
var truck = function (eng) {
    //private members
    var engine = eng,
    start = function () {
        alert('Truck started ' + engine);
    },
    stop = function () {
        alert('Truck stopped');
    };

    //public members
    return {
        start: start,
        stop: stop
    };
}('V8');
```

```
var calculator = function () {
    var eqCtl,
    currNumberCtl,
    operator,
    operatorSet = false,
    equalsPressed = false,
    lastNumber = null,

    init = function (equals, currNumber) {
        eqCtl = equals;
        currNumberCtl = currNumber;
    },

     //Code omitted for the sake of brevity


    return {
        init: init,
        numberClick: numberClick,
        setOperator: setOperator,
        clearNumbers: clearNumbers
```

```
    };
} ();
```

When the code for *truck* and *calculator* is initially parsed, the functions are invoked immediately which creates one *truck* and one *calculator* in memory. Although this works well and ensures that functions aren't duplicated in memory, what if you'd like to create and use multiple *truck* or *calculator* objects on a single page?

There's another technique that can be used with the Revealing Module Pattern when multiple objects need to be created in a page or script. This is done by removing the final parentheses in the code above and making the first letter upper-case to keep with the convention discussed earlier:

```
var Truck = function (eng) {
    //private members
    var engine = eng,
    start = function () {
        alert('Truck started ' + engine);
    },
    stop = function () {
        alert('Truck stopped');
    };

    //public members
    return {
        start: start,
        stop: stop
    };
}; //Parentheses removed



var Calculator = function () {
    var eqCtl,
    currNumberCtl,
    operator,
    operatorSet = false,
    equalsPressed = false,
    lastNumber = null,

    init = function (equals, currNumber) {
        eqCtl = equals;
        currNumberCtl = currNumber;
    },


     //Code omitted


    return {
        init: init,
        numberClick: numberClick,
        setOperator: setOperator,
        clearNumbers: clearNumbers
```

```
    };
}; //Parentheses removed
```

Once the parentheses are removed, the objects can be instantiated:

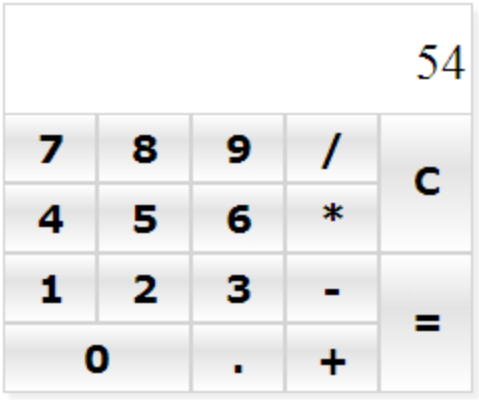
```
//Invoke the object and assign to truck
var truck = new Truck('V8');
truck.start();


var eqCtl = document.getElementById('eq'),
currNumberCtl =
   document.getElementById('currNumber');

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

In this example a new *truck* object is created and its *start()* function is called. A *Calculator* object is then instantiated and assigned to the *myCalc* variable. The public functions exposed by *Truck* and *Calculator* can be called using the respective objects that are created. For example, the previous code uses *myCalc* to call the *Calculator* object's *init()* function.

Different values can be passed to the call to *Calculator()* following this same pattern if multiple objects are needed in a page or script. This allows multiple calculators to be displayed in a page:



Keep in mind that each call to *Calculator()* (or *Truck()*) places a new copy of each function in memory, but the impact is quite minimal in this case. If you're worried about multiple copies of functions being placed in memory as objects are created but still want the public/private functionality, consider using the Revealing Prototype Pattern since it leverages JavaScript prototyping. That's covered next!

Revealing Prototype Pattern

If you like the features offered by the Revealing Module Pattern but want to take advantage of JavaScript's prototype functionality and the benefits it offers, the Revealing Prototype Pattern may be what you need. It offers the benefits of the Revealing Module Pattern but also provides a way to share function implementations across object instances through prototyping. The pattern is a combination of the Prototype Pattern and Revealing Module Pattern.

To start using the Revealing Prototype Pattern, you'll first create a constructor and define any variables that are unique to the object instance. Here's an example of defining a constructor for *Truck*:

```
var Truck = function (eng) {
    this.engine = eng;
};
```

Here's what the *Calculator* looks like. This code should look familiar because at this point it's following the Prototype Pattern:

```
var Calculator = function (cn, eq) {
    this.currNumberCtl = cn;
    this.eqCtl = eq;
};
```

This example allows two DOM elements to be passed, which are stored in the *currNumberCtl* and eqCtl variables.

Once the constructor is created, you can define the prototype. Unlike the Prototype Pattern which assigns a JavaScript object literal to the prototype property, the Revealing Prototype Pattern assigns a function which is immediately invoked as with the Revealing Module Pattern:

```
Truck.prototype = function () {

    //code

}();

Calculator.prototype = function () {

    //code

}();
```

The full constructor and prototype definition for *Truck* and *Calculator* are shown next. Looking through the code, you'll notice that it's quite similar to the Revealing Module Pattern except that it

assigns the container function to the prototype.

```
var Truck = function (eng) {
    this.engine = eng;
};

Truck.prototype = function () {
    var start = function () {
        alert('Truck started ' + this.engine);
    },
    stop = function () {
        alert('Truck stopped');
    };

    //public members
    return {
        start: start,
        stop: stop
    };

}();



var Calculator = function (cn, eq) {
    this.eqCtl = document.getElementById(eq);
    this.currNumberCtl =
      document.getElementById(cn);
    this.operator = null;
    this.operatorSet = false;
    this.equalsPressed = false;
    this.lastNumber = null;
};

Calculator.prototype = function () {
    var add = function (x, y) {
        return x + y;
    },

    subtract = function (x, y) {
        return x - y;
    },

    multiply = function (x, y) {
        return x * y;
    },

    divide = function (x, y) {
        if (y == 0) {
            alert("Can't divide by 0");
            return 0;
        }
        return x / y;
    },
```

```
setVal = function (val) {
    this.currNumberCtl.innerHTML = val;
},

setEquation = function (val) {
    this.eqCtl.innerHTML = val;
},

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

setOperator = function (newOperator) {
    if (newOperator == '=') {
        this.equalsPressed = true;
        calculate.call(this);
        setEquation.call(this, '');
        return;
    }

    //Handle case where = was pressed
    //followed by an operator (+, -, *, /)
    if (!this.equalsPressed) calculate.call(this);
    this.equalsPressed = false;
    this.operator = newOperator;
    this.operatorSet = true;
    this.lastNumber = parseFloat(
       this.currNumberCtl.innerHTML);
    var eqText = (this.eqCtl.innerHTML == '') ?
                 this.lastNumber + ' ' +
                 this.operator + ' ' :
                 this.eqCtl.innerHTML + ' ' +
                 this.operator + ' ';
    setEquation.call(this, eqText);
},

numberClick = function (e) {
    var button = (e.target) ? e.target : e.srcElement;
    if (this.operatorSet == true ||
        this.currNumberCtl.innerHTML == '0') {
        setVal.call(this, '');
        this.operatorSet = false;
    }
    setVal.call(this,this.currNumberCtl.innerHTML +
      button.innerHTML);
    setEquation.call(this, this.eqCtl.innerHTML +
      button.innerHTML);
},

calculate = function () {
```

```
        if (!this.operator ||
            this.lastNumber == null) return;
        var displayedNumber = parseFloat(
           this.currNumberCtl.innerHTML),
           newVal = 0;
        switch (this.operator) {
            case '+':
               newVal = add(this.lastNumber,
                            displayedNumber);
                break;
            case '-':
                newVal = subtract(this.lastNumber,
                                  displayedNumber);
                break;
            case '*':
                newVal = multiply(this.lastNumber,
                                  displayedNumber);
                break;
            case '/':
                newVal = divide(this.lastNumber,
                                displayedNumber);
                break;
        }
        setVal.call(this, newVal);
        this.lastNumber = newVal;
    };

    return {
        numberClick: numberClick,
        setOperator: setOperator,
        clearNumbers: clearNumbers
    };
} ();
```

Looking through the code, you'll see that instead of assigning a JavaScript object literal to the prototype (as with the Prototype Pattern), a function is assigned. It also allows you take advantage of public/private visibility functionality through the return block, so only functions accessible to external objects are exposed.

There's something interesting that happens with variables though, especially if you plan on creating more than one *Calculator* object in a page. Looking at the public functions, you'll see that the *this* keyword is used to access the *currNumberCtl* and *eqCtl* variables defined in the constructor. This works great since the caller of the public functions will be the *Calculator* object instance, which of course has the two variables defined. However, when one of the public functions calls a private function such as *setVal()*, the context of *this* changes and you'll no longer have access to the two variables. That's where JavaScript's *call()* or *apply()* functions come in handy (more on these later). In this example, the *call()* function is used where appropriate to ensure *this* is preserved.

An instance of the *Calculator* can be invoked, using the following code:

```
var cn = document.getElementById('currNumber'),
    eq = document.getElementById('eq'),
    calc = new Calculator(cn, eq);
```

The Revealing Prototype Pattern combines the best features of the Prototype Pattern and Revealing Prototype Pattern into a single technique. However, like the Prototype Pattern, it relies on JavaScript's *this* keyword which can be tricky to deal with at times. Let's take a closer look at how you can deal with *this*.

Working with the JavaScript *this* Keyword

JavaScript's *this* keyword can be a bit tricky to work with, depending on the context in which it's used. When it's used with patterns such as the Prototype or Revealing Prototype, working with *this* can be challenging in some cases. Unlike languages such as C# or Java, *this* can change context. For example, if a *Calculator* object named *calc* calls an *add()* function, then *this* represents the *Calculator* object, which means you can easily access any variables defined in the object, such as a variable named *tax* by simply using *this.tax*.

```
//Using this inside of add()
//gets you to the calc object
calc.add(2, 2);
```

However, if *add()* makes a call to another function, then *this* changes context and no longer represents the *Calculator* object. In fact, *this* will change to the *window* object, which means you can no longer access variables defined in the *Calculator* object such as *tax*. This presents a bit of a problem that's especially challenging if you've never dealt with it before.

There are several ways to handle this challenge. First, you can pass *this* as a parameter to other functions. An example of passing *this* between functions is shown next:

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

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

    subtract = function (x, y) {
        this.eqCtl.innerHTML = x - y + this.tax;
    },

    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;

        //Pass the Calculator object that
        //called clearNumbers() to other
        //functions as a parameter
        setVal('0', this);
        setEquation('', this);
    };

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

If a *Calculator* object calls a *clearNumbers()* function, then 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 passes *this* as a parameter to each of the functions, and they then use it like normal. Although this type of code works, it pollutes your function parameters in some cases and becomes a little messy to work with. I don't recommend going with this approach.

Another technique that can be used involves JavaScript's *call()* function. The *call()* 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, then you can do the following:

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

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. JavaScript also has an *apply()* function that is very similar to *call()*. The main difference is that *apply()* allows an array of parameter values to be passed:

```
//Separate params with a comma
setVal.call(this, 'param1value', 'param2value');

//Params sent using an array
setVal.apply(this, ['param1value', 'param2value']);
```

The following code uses the *call()* function to update the code shown earlier. The *clearNumbers()* function uses JavaScript's *call()* function to invoke the *setVal()* and *setEquation()* functions and preserve the current value of *this* in the process. Notice the *setVal()* and *setEquation()* functions no longer need the extra parameter, as the functions shown earlier 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.

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

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

    subtract = function (x, y) {
        this.eqCtl.innerHTML =
          x - y + this.tax;
    },

    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;

        //Set context of this to the Calculator
        //object that called clearNumbers()
        setVal.call(this, '0');
        setEquation.call(this, '');
    };

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

As a quick side note, another example of where JavaScript's *this* keyword gets tricky is inside of jQuery event handlers. For example, assume that an *init()* function is called that adds a *click* event to DOM elements. What if you want to get to a value of *this* representing the container *Calculator*

object while inside of the *click* event handler function? The context of *this* changes to the anchor tag, making it challenging. Two potential options that can be used are shown next.

The first option is to store the value of *this* as a variable outside of the *click* event handler function. By doing this, a closure is created, and you can still access the original value of *this* when the event fires.

```
Calculator.prototype = {
    //private members
    init: function () {
        //Option 1 for working with this
        var calcObject = this;
        $('a').click(function () {
            //Can't simply use this or $(this)
            //since this represents the anchor
            //now calcObject will represent the
            //Calculator object
            calcObject.highlight($(this));
        });
    },

    highlight: function (anchor) {
        anchor.toggleClass('highlight');
    }
};
```

Another option is to use an overloaded version of various jQuery event handlers, such as *on()* to pass *this* and make it accessible through the *event* object.

```
Calculator.prototype = {
    //private members
    init: function () {
        //Option 2 for working with this
        //Pass this into the on() call
        $('a').on('click', { calcObject: this },
         function (event) {
            //Access the original value of this
            //through the event object's data property
            event.data.calcObject.highlight($(this));
         });
    },

    highlight: function (anchor) {
        anchor.toggleClass('highlight');
    }
};
```

Notice *this* is assigned to an object literal property named *calcObject* in the *on()* parameters. Once the click event fires, the event object passed to the callback function can be used to get to the data property, which exposes the *calcObject* value that was passed in.

Although working with JavaScript's *this* keyword can be challenging in some scenarios, there are several different techniques that can be used to make it easier to work with. Now that you've seen how to handle *this*, let's wrap up by discussing how the different prototype patterns can be extended.

Using the JavaScript Prototype Property to Structure and Extend Code

Up to this point, you've seen there are several different patterns that can be used in JavaScript to structure code and make it more re-useable, more maintainable, and less subject to naming collisions. Patterns like the Revealing Module Pattern, Prototype Pattern, Revealing Prototype Pattern, and others can be used to structure code and avoid function spaghetti code. One of the great features offered by both the Prototype and the Revealing Prototype patterns is the extensibility they provide. They're quite flexible, especially compared to the Revealing Module Pattern.

Because both patterns rely on JavaScript prototyping, functions are shared across objects instances in memory, and it's straightforward for users of objects following these patterns to extend or override existing functionality. Here's a quick review of the different prototype-style patterns:

Prototype Pattern Example

```
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;
    }
};
```

Revealing Prototype Pattern Example

```
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 the two examples, you can see both patterns rely on the prototype functionality available in JavaScript. The Revealing Prototype Pattern has the added functionality of being able to define public and private members within the *Calculator*.

Sidebar: Some people love the public/private feature, and other people think all JavaScript code should be accessible to callers. It really depends on your background and what you want. I personally like having the public/private member functionality available since if you're working with editors that support code help, you only see the functions or variables you should call as you're typing. It's all personal preference, though, and if you'd prefer to add an underscore (as mentioned earlier) in front of a variable (such as *_firstName*) or function to mark it as private, that's certainly another option that's popular in the JavaScript world.

What if you want to extend one of the *Calculator* objects shown above or override existing functions? When using the Prototype or Revealing Prototype patterns, this is possible since they both rely on prototyping. If you're new to prototyping, here's a quick introduction to using the prototype property and different ways it can be used in your JavaScript code.

Getting Started with JavaScript Prototyping

Prototyping allows objects to inherit, override, and extend functionality provided by other objects in a similar manner as inheritance, overriding, abstraction, and related technologies do in C#, Java, and other languages. Every object you create in JavaScript has a prototype property by default that can be accessed. To better understand prototyping, take a look at the example below. Rather than adding methods directly into the *BaseCalculator* constructor definition, as with patterns like 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;
};
```

This code 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 including *add(x,y)* and *subtract(x,y)*. This type of definition can be simplified, as shown earlier,

with the Prototype Pattern by using an object literal to define the prototype functions:

```
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 (alternatively you can also use *Object.create()* if it's supported in the browser):

```
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 great feature provided by the prototype property. An example of creating a new *Calculator* object instance is shown next:

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

//Variable defined in the BaseCalculator
//parent object is accessible from
//the derived Calculator object's constructor

alert(calc.decimalDigits);
```

In the previous code, *BaseCalculator's decimalDigits* variable is accessible to *Calculator* since 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, as opposed to assigning a new *BaseCalculator* instance:

```
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* will be accessible. For example, the following code will throw a JavaScript error when the code tries to access *decimalDigits*. This is due to *BaseCalculator's* constructor 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 3rd 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. An example of overriding the *add()* function provided in Calculator is shown next:

```
//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 *tax* together. The override applies to all *Calculator* object instances created after the override. Any time you'd like to minimize the number of functions loaded into memory or need the ability to override existing functionality, you can take advantage of the prototype property.

Conclusion

Applying JavaScript patterns to your code can really help clean it up and make it more re-useable and maintainable. I hope that this guide has provided you with a solid look at some of the different

JavaScript patterns and how they can be used to structure code. Once you learn the patterns you'll never go back to writing function spaghetti code ever again!

A Quick Note about Custom Onsite/Online Training

If you're interested in training on JavaScript and other technologies, please contact us! We cover a wide range of technologies such as:

- AngularJS
- JavaScript
- JavaScript Patterns
- jQuery Programming
- Web Technology Fundamentals
- Node.js
- Responsive Design
- .NET Technologies
- C# Programming
- C# Design Patterns
- ASP.NET MVC
- ASP.NET Web API
- WCF
- End-to-end application development classes are also available

We fly all around the world to give our training classes and train developers at companies like Intel, Microsoft, UPS, Goldman Sachs, Alliance Bernstein, Merck, various government agencies, and many more. Online classes are also available. Please contact us at info@TheWahlinGroup.com if you're interested in onsite or online training for your developers.