Evolution of a JS Controller Scheme

Posted At : February 19, 2009 4:53 PM | Posted By : Jon Hartmann
Related Categories: Examples & Tutorials, Javascript

It took me quite a while to discover the power and flexibility of a controller scheme for Javascript, but thanks to some awesome videos and tutorials, I've come up with a setup that works quite well, and I thought I'd share it with you. I'm going to start with a basic object, and evolve it into what I use, so you can understand why each thing is done, and what happens at each stage.

In this article I focus on the controller itself, not on how it might impact other JS. In real live you'll want a more friendly method of attaching event handlers.

Make an Object

Ok, first thing you know is that you need an object for your controller. This will let you put all of our code in one spot as methods of the object.

```
<script>
    var Controller = {};
</script>
```

Great start. Now lets add an initialization method and wire it up to window.onload so that we can have the initialization code fire after our elements are available.

```
<script>
    var Controller = {}
    Controller.init = function () {
        alert('I Loaded')
    }

    window.onload = Controller.init;
</script>
```

Ok, so now you've got an init method, lets set it up to accept a configuration value and add a method to display one of the configuration values.

```
<script>
    var config = {
        userID: 1234
    };

    var Controller = {}
    Controller.init = function (config) {
        this.options = config;
    }
    Controller.getUserID = function () {
        return this.options.userID;
    }

    window.onload = function () {
        Controller.init(config);
        alert(Controller.getUserID());
    };
</script>
```

Ok, looks good. Now lets have our init method find a button on our page and wire up an event to it. My button looks like this:

```
<button id="button1">I'm a Button</button>
```

And my updated controller will look like this:

```
<script>
    var config = {
        userID: 1234
    };

    var Controller = {}
    Controller.init = function (config) {
        this.options = config;

        var button = document.getElementById('button1');
        button.onclick = Controller.handleClick;
    }
    Controller.getUserID = function () {
        return this.options.userID;
    }
    Controller.handleClick = function () {
        alert(Controller.getUserID());
    }

    window.onload = function () {
        Controller.init(config);
    };
</script>
```

Ok, nifty. You've got things working fine and you're done right? I mean whats wrong with it?

The Catch

The problem is what you have is composed wholly of public methods. Your configuration can be changed at any time, and your methods aren't really integrated together. If you needed to access the button1 in another method you'd only have two options: code the element lookup again, or save the button value as a public attribute of Controller. Not ideal.

Closures

The answer you're looking for is closures. Closures are probably one of the most powerful aspects of Javascript, and one of the most confusing. Basically, closure says that anything inside a block can access values of that block, even after the block has executed.

Huh? Well its easier just to show you. Clear out that controller script (but leave your button) and give in the following code:

```
<script>
    function setup () {
        var x = 'I\'m inside the closure';

        document.getElementById('button1').onclick = function () {
            alert(x)
        }
    }

    window.onload = setup;
</script>
```

Now when you hit the button you get an alert stating "I'm inside the closure". Pretty straight forward until you look at how this code executes.

1. The function setup() is created and set to fire when window.onload occurs.
2. The window finishes loading and setup() is called. Setup assigns a local variable x to the string "I'm inside the closure" and sets the onclick event of button1 to a function.
3. When you click the button, the function () { alert(x); } is called and the alert is created.

Ok, did you catch that? x is a local variable inside setup(); once setup() finishes execution, it shouldn't exist any more, so clicking the button should cause an error. Well, it should if not for closures. Even though the function setup() has finished execution, the function () { alert(x); } still has access because the function was declared inside the function that also set x.

Now What

What can closures do for us and our controller object? It allows us to have private variables and private methods. In fact, you already made a private variable called x, since it doesn't exist outside of its enclosure. Making a private function is as easy as rearranging the code we just used:

```
<script>
    function setup () {
        var x = 'I\'m inside the closure';

        function handleClick () {
            alert(x)
        }

        document.getElementById('button1').onclick = handleClick;
    }

    window.onload = setup;
</script>
```

Now you've got a private method handleClick() that alerts a private variable x. Lets take a new look at our old controller with closures in mind.

```
<script>
    var config = {
        userID: 1234
    };

    var Controller = function (config) {
        var options = config;
        var button = document.getElementById('button1');

        function getUserID () {
            return options.userID;
        }

        function handleClick () {
            alert(getUserID());
        }
        button.onclick = handleClick;
    }

    window.onload = function () {
        Controller(config);
    };
</script>
```

Much better. Now we have two private methods, and the config is kept neatly inside the main object. All done right?

Public Methods

We've managed to hide all of our methods and variables away inside our enclosure, but we've introduced another problem; all of our information is hidden away inside our enclosure! If we wanted to have a method, say setUserID() that worked on the config, our only option would be to directly wire it to an event, otherwise its trapped inside our enclosure. How can we break out of the enclosure? Pass something back.

Passing Back Public Methods

You can break out of the enclosure by passing back an object that contains the public methods. Since the object is created inside the enclosure, it still has access to the methods. We still need a function to create the enclosure, but lets make it pass back an object that exposes setUserID().

```
<script>
    var config = {
        userID: 1234
    };
    var Controller = function (config) {
        var options = config;
        var button = document.getElementById('button1');

        function getUserID () {
            return options.userID;
        }

        function setUserID (userID) {
            options.userID = userID;
        }

        function handleClick () {
            alert(getUserID());
        }
        button.onclick = handleClick;

        return {
            setUserID: setUserID
        };
    }

    window.onload = function () {
        Controller = Controller(config);

        Controller.setUserID(6789);
    };
</script>
```

Congratulations! You've exposed a private method as a public one. Only problem is you have this ugly Controller = Controller(config); construct. What if we could let the controller set itself up?

```
<script>
    var config = {
        userID: 1234
    };

    var Controller = function (config) {
        ...

        return {
            ...
        };
    }(config);
</script>
```

That would be nice, but you're init code can't happen until the window loads right? Wait, what happened to that init method any way? Lets bring it back.

```
<script>
    var config = {
        userID: 1234
    };

    var Controller = function (config) {
        var options = config;

        function init () {
            var button = document.getElementById('button1');
            button.onclick = handleClick;
        }

        function getUserID () {
            return options.userID;
        }

        function setUserID (userID) {
            options.userID = userID;
        }

        function handleClick () {
            alert(getUserID());
        }

        return {
            init: init,
            setUserID: setUserID
        };
    }(config);

    window.onload = function () {
        Controller.init();

        Controller.setUserID(6789);
    };
</script>
```

Nice. But I'm not done with you yet. We've wound up with two constructors, one which takes in the config, and one that handles initial setup. Lets pull the config out of the auto-calling function and move it to the init() method;

```
<script>
    var config = {
        userID: 1234
    };

    var Controller = function () {
        function init (config) {
            var options= config;
            var button = document.getElementById('button1');
            button.onclick = handleClick;
        }

        function getUserID () {
            return options.userID;
        }

        function setUserID (userID) {
            options.userID = userID;
        }

        function handleClick () {
            alert(getUserID());
        }

        return {
            init: init,
            setUserID: setUserID
        };
    }();

    window.onload = function () {
        Controller.init(config);

        Controller.setUserID(6789);
    };
</script>
```

Works fine... I'm not sure why though. Although init() is inside the enclosure, setUserID() shouldn't have access to the config variable as far as I understand it, since config is not a part of an enclosure of setUserID(). Ignoring that headache for a second, lets thing about this... say we have tons of functions that create vars in this controller, how would a person see what variables are declared? They'd have to move through all of the methods to find them all. Lets pull the variable declarations for things we might want to reuse up into the constructor to make sure they are easy to see and enclosed.

```
<script>
    var config = {
        userID: 1234
    };

    var Controller = function () {
        // Private variables
        var options,
            button;

        function init (config) {
            options = config;
            button = document.getElementById('button1');
            button.onclick = handleClick;
        }

        function getUserID () {
            return options.userID;
        }

        function setUserID (userID) {
            options.userID = userID;
        }

        function handleClick () {
            alert(getUserID());
        }

        return {
            init: init,
            setUserID: setUserID
        };
    }();

    window.onload = function () {
        Controller.init(config);

        Controller.setUserID(6789);
    };
</script>
```

Ok, now which methods inside the constructor are public and which are private? We could just reorder and label them, but we would still have all of these init: init declarations for our return object. Lets pull the public methods down to where they are returned.

```
<script>
```

```
    var config = {
        userID: 1234
    };

    var Controller = function () {
        // Private variables
        var options,
            button;

        // Private Methods
        function getUserID () {
            return options.userID;
        }

        function handleClick () {
            alert(getUserID());
        }

        // Public Methods
        return {
            init: function (config) {
                options = config;
                button = document.getElementById('button1');
                button.onclick = handleClick;
            },
            setUserID: function (userID) {
                options.userID = userID;
            }
        };
    }();

    window.onload = function () {
        Controller.init(config);

        Controller.setUserID(6789);
    };
</script>
```

Ok, now we are in business. If you'll indulge me in one final tweak; listing the ID of the button right there in the init method bothers me. Its not something that would change often enough to need to be in the config, but it potentially references to it might be strewn through out the controller making it a pain if you do need to change it. Lets pull it up into a special configuration variable to house things that are static, but might need changing all at once.

```
<script>
    var config = {
        userID: 1234
    };

    var Controller = function () {
        // Static Values
        var configuration = {
            IDs: {
                button: 'button1'
            }
        };

        // Private variables
        var options,
            button;
        // Private Methods
        function getUserID () {
            return options.userID;
        }

        function handleClick () {
            alert(getUserID());
        }

        // Public Methods
        return {
            init: function (config) {
                options = config;
                button = document.getElementById(configuration.IDs.button);
                button.onclick = handleClick;
            },
            setUserID: function (userID) {
                options.userID = userID;
            }
        };
    }();

    window.onload = function () {
        Controller.init(config);

        Controller.setUserID(6789);
    };
</script>
```

Ok, so the you go. You've got a scheme for a controller that has static values, public and private methods, and private variables. You're ready to rock.

My Basic Controller

Although you can winnow the above examples down into a usable base, I figured I'd save you some time. This is what I use as the base for all of my controllers.

```
<script>
    var Controller = function () {
        var config = {};

        return {
            init: function () {


            }
        };
    }();
    window.onload = Controller.init;
</script>
```

Update: Peter requested that I give links to the videos that I referenced at the beginning of the post, and I'm more then happy to oblige.

- Douglas Crockford on Javascript
- Javascript Maintainablity Lecture by Chris Heilmann