JavaScript design patterns – Part 1: Singleton, composite, and façade

This is the first article in a series about common design patterns used in JavaScript. Design patterns are proven ways of programming to help make your code more maintainable, scalable, and decoupled, all of which are necessary when creating large JavaScript applications, especially in large groups.

In Part 2 of this series, you're introduced to another 3 design patterns: adapter, decorator, and factory. Part 3 discusses 3 more design patterns: proxy, observer, and command.

Singleton

The singleton pattern is what you use when you want to ensure that only one instance of an object is ever created. In classical object-oriented programming languages, the concepts behind creating a singleton was a bit tricky to wrap your mind around because it involved a class that has both static and non-static properties and methods. I'm talking about JavaScript here though, so with JavaScript being a dynamic language without true classes, the JavaScript version of a singleton is excessively simple.

Why do you need the singleton?
Before I get into implementation details, I should discuss why the singleton pattern is useful for your applications. The ability to ensure you have only one instance of an object can actually come in quite handy. In server-side languages, you might use a singleton for handling a connection to a database because it's just a waste of resources to create more than one database connection for each request. Similarly, in front-end JavaScript, you might want to have an object that handles all AJAX requests be a singleton. A simple rule could be: if it has the exact same functionality every time you create a new instance, then make it a singleton.

This isn't the only reason to make a singleton, though. At least in JavaScript, the singleton allows you to namespace objects and functions to keep them organized and keep them from cluttering the global namespace, which as you probably know is a horrible idea, especially if you use third party code. Using the singleton for name-spacing is also referred to as the module design pattern.

Show me the singleton
To create a singleton, all you really need to do is create an object literal.

```
var Singleton = {
    prop: 1,
    another_prop: 'value',
    method: function() {…},
    another_method: function() {…}
};
```

You can also create singletons that have private properties and methods, but it's a little bit trickier as it involves using a closure and a self-invoking anonymous function. Inside a function, some local functions and/or variables are declared. You then create and return an object literal, which has some methods that refererence the variables and functions that you declared within the larger function's scope. That outer function is immediatly executed by placing () immediately after the function declaration and the resulting object literal is assigned to a variable. If this is confusing, then take a look over the following code and then I'll explain it some more afterward.

```
var Singleton = (function() {
    var private_property = 0,
        private_method = function () {
            console.log('This is private');
        }

    return {
        prop: 1,
        another_prop: 'value',
        method: function() {
            private_method();
            return private_property;
        },
        another_method: function() {…}
    }
}());
```

The key is that when a variable is declared with var in front of it inside a function, that variable is only accessible inside the function and by functions that were declared within that function (the functions in the object literal for example). The return statement gives us back the object literal, which gets assigned to Singleton after the outer function executes itself.

Namespacing with the singleton

In JavaScript, namespacing is done by adding objects as properties of another object, so that it is one or more layers deep. This is useful for organizing code into logical sections. While the YUI JavaScript library does this to a degree that I feel is nearly excessive with numerous levels of namespacing, in general it is considered best practice to limit nesting namespaces to only a few lavels or less. The code below is an example of namespacing.

```
var Namespace = {
    Util: {
        util_method1: function() {…},
        util_method2: function() {…}
    },
    Ajax: {
        ajax_method: function() {…}
    },
    some_method: function() {…}
};

// Here's what it looks like when it's used
Namespace.Util.util_method1();
Namespace.Ajax.ajax_method();
Namespace.some_method();
```

The use of namespacing, as I said earlier, keeps global variables to a minumum. Heck, you can even have entire applications attached to a single object namespace named app if that's your perogative. If you'd like to learn a little more about the singleton design pattern as well as its applicability in namespacing, then go ahead and check out the "JavaScript Design Patterns: Singleton" article on my personal blog.

Composite

If you read through the section about the singleton pattern and thought, "well, that was simple," don't worry, I have some more complicated patterns to discuss, one of which is the composite pattern. Composites, as the word implies, are objects composed of multiple parts that create a single entity. This single entity serves as the access point for all the parts, which, while simplifying things greatly, can also be deceiving because there's no implicit way to tell just how many parts the composite contains.

Structure of the composite

The composite is explained best using an illustration. In Figure 1, you can see two different types of objects: containers and galleries are the composites and images are the leaves. A composite can hold children and generally doesn't implement much behavior. A leaf contains most of the behavior, but it cannot hold any children, at least not in the classical composite example.

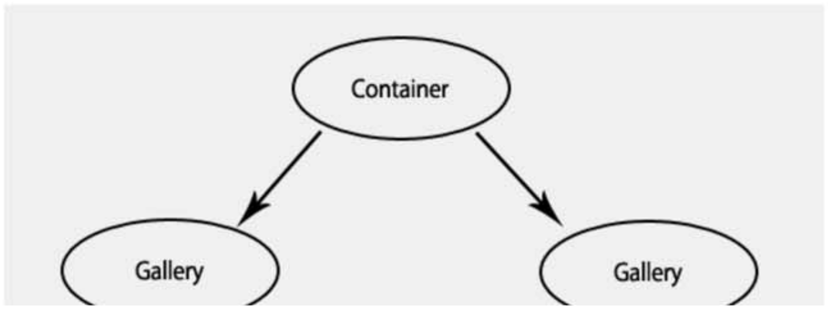

Image
Image
Image
Image

Figure 1. The structure of a composite

As another example, I'm 100% certain you've seen the composite pattern in action before but never really thought about it. The file structure in computers is an example of the composite pattern. If you delete a folder, all of its contents get deleted too, right? That's essentially how the composite pattern works. You can call a method on a composite object higher up on the tree and the message will be delivered down through the hierarchy.

Composite coding example

This example creates an image gallery as an example of the composite pattern. You will have just three levels: album, gallery, and image. The album and galleries will be composites and the images will be the leaves, just like in Figure 1. This is a more defined structure than a composite needs to have, but for this example, it makes sense to limit the levels to only being composites or leaves. A standard composite doesn't limit which levels of the hierarchy that leaves can be on, nor do they limit the number of levels.

To start things off, you create the `GalleryComposite` "class" used for both the album and the galleries. Please note that I am using jQuery for the DOM manipulation to simplify things.

```
var GalleryComposite = function (heading, id) {
    this.children = [];

    this.element = $('<div id="' + id + '" class="composite-gallery"></div>')
    .append('<h2>' + heading + '</h2>');
}

GalleryComposite.prototype = {
    add: function (child) {
        this.children.push(child);
        this.element.append(child.getElement());
    },

    remove: function (child) {
        for (var node, i = 0; node = this.getChild(i); i++) {
            if (node == child) {
                this.children.splice(i, 1);
                this.element.detach(child.getElement());
                return true;
            }

            if (node.remove(child)) {
                return true;
            }
        }

        return false;
    },

    getChild: function (i) {
        return this.children[i];
    },

    hide: function () {
        for (var node, i = 0; node = this.getChild(i); i++) {
            node.hide();
        }

        this.element.hide(0);
    },
    show: function () {
        for (var node, i = 0; node = this.getChild(i); i++) {
            node.show();
        }

        this.element.show(0);
    },

    getElement: function () {
        return this.element;
    }
}
```

There's quite a bit there, so how about I explain it a little bit? The `add`, `remove`, and `getChild` methods are used for constructing the composite. This example won't actually be using `remove` and `getChild`, but they are helpful for creating dynamic composites. The `hide`, `show`, and `getElement` methods are used to manipulate the DOM. This composite is designed to be a *representation* of the gallery that is shown to the user on the page. The composite can control the gallery elements through `hide` and `show`. If you call `hide` on the album, the entire album will disappear, or you can call it on just a single image and just the image will disappear.

Now create the `GalleryImage` class. Notice that it uses all of the exact same methods as the `GalleryComposite`. In other words, they implement the same interface, except that the image is a leaf so it doesn't actually do anything for the methods regarding children, as it cannot have any. Using the same interface is required for the composite to work because a composite element doesn't know whether it's adding another composite element or a leaf, so if it tries to call these methods on its children, it needs to work without any errors.

```
var GalleryImage = function (src, id) {
    this.children = [];

    this.element = $('<img />')
    .attr('id', id)
    .attr('src', src);
}

GalleryImage.prototype = {
    // Due to this being a leaf, it doesn't use these methods,
    // but must implement them to count as implementing the
    // Composite interface
    add: function () { },

    remove: function () { },

    getChild: function () { },

    hide: function () {
        this.element.hide(0);
    },

    show: function () {
        this.element.show(0);
    },

    getElement: function () {
        return this.element;
    }
}
```

Now that you have the object prototypes built, you can use them. Below you can see the code that actually builds the image gallery.

```
var container = new GalleryComposite('', 'allgalleries');
var gallery1 = new GalleryComposite('Gallery 1', 'gallery1');
var gallery2 = new GalleryComposite('Gallery 2', 'gallery2');
var image1 = new GalleryImage('image1.jpg', 'img1');
var image2 = new GalleryImage('image2.jpg', 'img2');
var image3 = new GalleryImage('image3.jpg', 'img3');
var image4 = new GalleryImage('image4.jpg', 'img4');

gallery1.add(image1);
gallery1.add(image2);

gallery2.add(image3);
gallery2.add(image4);

container.add(gallery1);
container.add(gallery2);

// Make sure to add the top container to the body,
// otherwise it'll never show up.
container.getElement().appendTo('body');
container.show();
```

That's all there is to the composite! If you want to see a live demo of the gallery, you can visit the demo page on my blog. You can also read the "JavaScript Design Patterns: Composite" article on my blog for a little more information on this pattern.

Façade

The façade pattern is the final design pattern in this article, which is just a function or other piece of code that simplifies a more complex interface. This is actually quite common, and, one could argue, most functions are actually made for this very purpose. The goal of a façade is to simplify a larger piece of logic into one simple function call.

Examples of the façade

You just might be using the façade pattern all the time without realizing that you're using any design pattern at all. Just about every single library that you use in any programming language uses the facade pattern to some degreebecause generally their purpose is to make complex things simpler.

Let's look at jQuery for an example. jQuery has a single function (`jquery()`) to do so many things; It can query the DOM, create elements, or just convert DOM elements into jQuery objects. If you just look at querying the DOM, and take a peek at the number of lines of code used to create that capability, you'd probably say to yourself, "I'm glad I didn't have to write this," because it is very long and complex code. They have successfully used the
façade pattern to convert hundreds of lines of complex code into a single, short function call for your convenience.

Finishing up

The façade pattern is pretty easy to understand but if you are interested you can learn more from the JavaScript Design Patterns: Façade post on my personal blog.