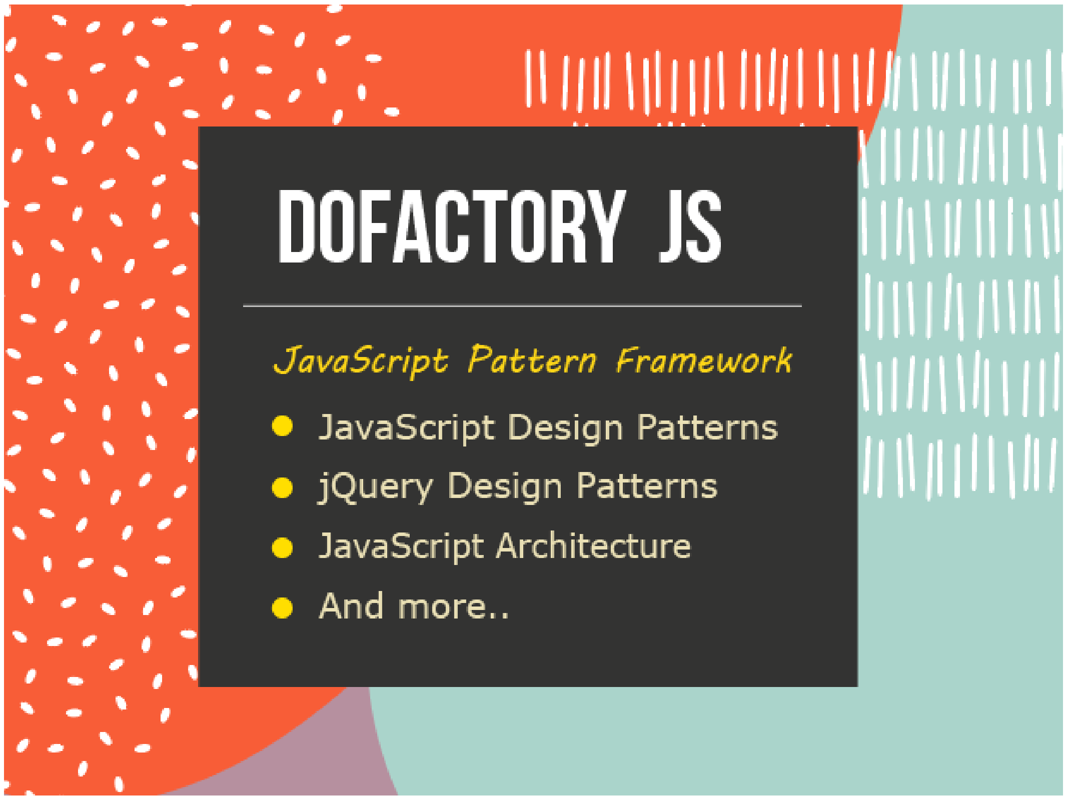
Dofactory JS 6.0

jQuery JavaScript Patterns



by

Data & Object Factory, LLC

www.dofactory.com

6. jQuery JavaScript Patterns

Index

Introduction

jQuery is by far the most popular JavaScript library in use today; 60% of the 10,000 most visited websites use it. The total number of deployments worldwide is around 20 million websites.

jQuery is cross-browser (supporting all popular browsers) and is designed to simplify navigation of the HTML document, select DOM elements, and make changes to the web page. On their website the authors state: "jQuery is a fast and concise JavaScript library that simplifies HTML document traversing, event handling, animation, and Ajax interactions for rapid web development". This describes it well.

It has been referred to as a *write less, do more* library which is absolutely correct. For web app developers the pre-jQuery days were considerably harder. Just to be able to select and manipulate elements on a page required a lot of messy code, especially in scenarios where multiple browsers and browser versions needed to be supported.

The jQuery selector syntax is based on CSS selectors which makes it succinct and highly effective. It takes very little code to select a complex set of page elements. Most web developers are familiar with CSS so that made the transition to jQuery relatively easy and painless.

The jQuery library is unique in that it focuses almost entirely on HTML manipulation. Other popular libraries are usually broader in scope, such as providing MV frameworks, data binding, HTML templating, and/or utility type functionalities. Many client side web projects use jQuery in combination with one other significant library, such as Backbone, Ember, or ExtJs. This is also true for modern HTML5/CSS3 projects (by the way: jQuery is fully CSS3 compliant).

Another advantage of using jQuery is that there is very strong community support. The library is maintained by an active group of JavaScript developers.

In this section we cover jQuery design patterns. We will dig into jQuery's source code and explore the patterns that have been used to build the library. There are many. You can learn from these and use the exact same patterns in your own projects whether you use jQuery or not. The best way to become an expert developer is to read and learn from code written by other expert developers.

Obviously, the developers of jQuery did not set out to cram as many patterns and best practices into their code as possible, but it is remarkable how many have been applied while building this powerful library. A real testament of what can be accomplished with patterns.

To get started we suggest you download the jQuery source code. It comes as a single JavaScript file with roughly 10,000 lines of code (including comments). Although jQuery is distributed as a single file, this is not how it is maintained. In github.com, the repository for jQuery, you'll find that the code base is partitioned in about 25 separate JavaScript files organized by functionality. This makes sense or else development and maintenance would be a nightmare.

Once downloaded, open the jQuery source file in your favorite editor. You will quickly recognize several commonly used JavaScript patterns. Scrolling down the file you may discover a few more. In the next several pages we will locate, identify, and discuss these patterns and best practice techniques. Again, learning from the masters is the best way to becoming an expert JavaScript developer yourself.

Module pattern

The Module pattern really stands out in jQuery because the entire code base resides in a single module. You can verify this by looking at the first line and then scrolling down to the very last line in the source file. The entire body of code is wrapped in a single immediate function (or IIFE) which makes up the module. Here are the relevant lines:

```
(function (window, undefined) {

  // ~ 10,000 lines of code ...

})( window );
```

The window argument is the global object. By passing it as an argument it becomes a locally available variable ensuring fast access. The parameter named undefined is to prevent hackers from redefining the built-in 'undefined' value.

All this is 'vanilla' Module and has been discussed in our Module discussion in the Moderns Patterns section.

Single var pattern

The first line within the module's function body is single var followed by about 2 dozen variable declarations and initializations. This is the Single var Pattern.

```
(function (window, undefined) {
var
    rootjQuery,
    readyList,

    document = window.document,
    location = window.location,
    navigator = window.navigator,

    _jQuery = window.jQuery,
    _$ = window.$,

    core_push = Array.prototype.push,
    core_slice = Array.prototype.slice,
    core_indexOf = Array.prototype.indexOf,
    core_toString = Object.prototype.toString,
    core_hasOwn = Object.prototype.hasOwnProperty,
    core_trim = String.prototype.trim,

    // more ...

})( window );
```

As you know variables get hoisted to the top, which can lead to unexpected results. The Single var pattern states that all variables in a function should be declared, and possibly initialized, by a single var at the beginning of the function body. The jQuery authors strictly adhere to this rule with only one var in each function.

Unfortunately, strictly enforcing the Single var pattern can sometimes lead to code that seems contrived and unnatural; take a look at jQuery's grep method:

```
grep: function( elems, callback, inv ) {
    var retVal,
        ret = [],
        i = 0,
        length = elems.length;

        inv = !!inv;
        // Go through the array, only saving the items
        // that pass the validator function
        for ( ; i < length; i++ ) {
            retVal = !!callback( elems[ i ], i );
            if ( inv !== retVal ) {
                 ret.push( elems[ i ] );
            }
        }

        return ret;
},
```

Here we have a single var with four variables, three of which are initialized. In the middle of this function is a 'for' loop that seems incomplete because the variables `i` and `length` are already defined and initialized at the top. You see these kinds of for loops throughout the library.

We prefer to have the looping variables declared and initialized where they are used (right in your face, so to speak) without having to guess whether `i` or `length` exist and/or have been initialized. Doing it this way would change the code to this:

```
grep: function( elems, callback, inv ) {
    var retVal,
        ret = [];

        inv = !!inv;
        // Go through the array, only saving the items
        // that pass the validator function
        for (var i = 0, length = elems.length; i < length; i++ ) {
            retVal = !!callback( elems[ i ], i );
            if ( inv !== retVal ) {
                 ret.push( elems[ i ] );
            }
        }
        return ret;
},
```

We think this code makes its intentions more clear (i.e. more in your face).

Double exclamation (!!)

The Double exclamation idiom converts an expression or a value that is *truthy* or *falsy* to a true Boolean value. This is discussed in the Essentials section. jQuery uses it in several places throughout its code base. The grep method uses it twice:

```
grep: function (elems, callback, inv) {
    var retVal,
        ret = [],
        i = 0,
        length = elems.length;
    inv = !!inv;

    // Go through the array, only saving the items
    // that pass the validator function
    for (; i < length; i++) {
        retVal = !!callback(elems[i], i);
        if (inv !== retVal) {
            ret.push(elems[i]);
        }
    }

    return ret;
},
```

In incoming `inv` argument is converted to a Boolean and the return value of the callback invocation also.

Overloading

Many languages support function overloading which is a feature that allows you to have multiple functions with the same name, but they all have a different set of parameters. For example `add(x, y)` and `add(x, y, z)` are overloaded functions. In typed languages the parameter type is also significant. We may have `add(x, y)` which adds two numbers and `add(s, t)` which concatenating two string values with `s` and `t` of type string.

JavaScript does not support overloading. However, you mimic it with a technique called *argument switching*. jQuery uses it extensively. In fact, the main jQuery method itself (which is aliased as $) is overloaded. It does different things depending on the arguments that are being passed. Here are some examples of different uses for jQuery (or $):

```
$(function() {
   // ...
});
```

This is a shorthand for `$(document).ready()` which runs when the DOM has been fully loaded. A function is passed into the `$()` function.

Here is another example:

```
$("#menu").addClass("active");
```

This is the regular selector syntax which returns a list of elements that meet the selector criteria. The `$()` function is passed a string.

And another example:

```
$("<p>Hello there!</p>").insertAfter("#intro");
```

This inserts a new DOM element after a specific location in the HTML document.

These are all examples of function overloading. The overloaded versions expect argument types that are either a function or a string. As you would expect the jQuery source code checks for the selector type. Here is an outline of this code (simplified):

```
init: function( selector, context, rootjQuery ) {

        // HANDLE: $(HTML)
        if (....) {
            return this;
        }

        // HANDLE: $(expr)
        if (....) {

            return this;
        }

        // HANDLE: $(function)
        if ( jQuery.isFunction( selector ) ) {
            return rootjQuery.ready( selector );
        }
        // ...
}
```

Interestingly, this goes beyond standard function overloading. If the selector type is a string value, the function needs to determine whether this is a CSS selector or an HTML string. This could be called *semantic overloading*.

Placeholder

The Placeholder Idiom uses an underscore (_) to indicate an unused parameter. jQuery uses this idiom also. Here are two code snippets from its source: the first one has a placeholder parameter in a callback function. The second uses it in a try-catch block:

```
// Listener
callback = function( _, isAbort ) {
    // etc.
}
```

And

```
// Need an extra try/catch for cross domain requests in Firefox 3
try {
    for ( i in headers ) {
        xhr.setRequestHeader( i, headers[ i ] );
    }
} catch( _ ) {}
```

Placeholder is a simple but helpful idiom.

Short-walk

Within the Single var area in jQuery there are several assignments of built-in JavaScript functions to local variables. Here is an extract:

```
core_push = Array.prototype.push,
core_slice = Array.prototype.slice,
core_indexOf = Array.prototype.indexOf,
core_toString = Object.prototype.toString,
core_hasOwn = Object.prototype.hasOwnProperty,
core_trim = String.prototype.trim,
```

These are implementations of the Short Walk idiom which is designed to speed up processing. Having these methods available as local variables will prevent the JavaScript engine from having to walk the scope chain every time these methods are needed.

A slight disadvantage is that you need to invoke these with `call` or `apply`, making the code a bit less legible, like so:

```
core_push.call( ret, arr );

core_indexOf.call( arr, elem, i );

core_toString.call(obj)
```

These lines are all from jQuery.

Minification

The above Short Walk idiom does more than provide faster access. It also helps in minifying the code. The local variables can get shortened to 1 or 2 character names, which would not be possible with built-in methods with longer names and object chains. This is the Minification Idiom.

Options hash

The Options Hash reduces the number of parameters in a function by grouping several into a single parameter object with a set of name/value pairs. The general usage pattern is like this:

```
var thing = new Thing(parm1, parm2, {
                       option1: "val1",
                       option2, "val2",
                       option3, "val3" });
```

The last argument is an object with 3 name/value pairs. Perhaps this last argument accepts, say, 10 different name/value pairs, but when not provided the Thing constructor function will assign default values for the 7 that are missing.

A great example of the Options Hash in jQuery is the Ajax method. Here is the method's description from the jQuery website:

```
jQuery.Ajax(url, [settings]);
------------------------------------------------------------------------------------
url:      A string containing the url to which the request is sent.
settings: A set of key/value pairs that configure the Ajax request. All settings are
optional.
```

The optional setting parameter is exactly the Options Hash idiom as we know it: it is used as a configuration parameter and all items are optional (there are about a dozen or so possible items in the jQuery's settings object).

Chaining Pattern

The Chaining pattern allows method calls to be chained together in a single statement. Chaining is deeply engrained in jQuery and any method that can support chaining does. In jQuery chaining is important because it limits selector processing which involves DOM traversing which can be rather slow. Here is an example of a chained jQuery statement:

```
$("#menu").addClass("highlight").css("margin", "2px").fadeIn("fast");
```

As an aside, the methods that operate on a jQuery selector set (a set of DOM elements) are commonly referred to as *commands*. In the above code we have three commands.

The selector selects the #menu element only once; all subsequent commands are applied to the selector's result set in a chain, one after the other.

In jQuery's code base we find numerous methods that return 'this' or 'this.each' which mostly represents the jQuery object which is the currently selected set of elements. Many methods use an internal helper method named jQuery.access; it has a 'chainable' argument in which the calling method indicates whether chaining is supported or not. Here is the skeleton code of jQuery.access and its use of the chainable flag:

```
access: function( elems, fn, key, value, chainable, emptyGet, pass ) {

    // ...

    return chainable ? ...

}
```

Chaining in jQuery can also be applied to events, like so:

```
$("#menu").click(function (e) {
        alert("You clicked me!");
    }).mouseenter(function (e) {
        $(this).css("backgroundColor", "Red");
    }).mouseleave(function (e) {
        $(this).css("backgroundColor", "Green");
    });
```

In summary, the Chaining pattern is very important to jQuery.

Namespace Pattern

jQuery makes use of namespaces. A goal of the Namespace pattern is to limit the number of variables introduced to the global namespace to just a single name. This single global item (object) then contains the entire code base for the application or library.

This is exactly how jQuery is organized: a single object, named jQuery, holds the entire body of code, including variables, properties, methods, and other (nested) namespaces. Example nested namespaces include `event`, and `support` which are referenced as `jQuery.event` and `jQuery.support` respectively.

Actually, jQuery cheats a bit on the namespace pattern as it adds not one but two names to the global namespace, the second one being $ which is just shorthand for jQuery. The $ is referred to as an *alias* of jQuery. In your code you can replace any $ occurrence with jQuery and vice versa; they are the same and this will not change your code.

The statement below is from jQuery. It shows how the global names are exported.

```
// Expose jQuery to the global object
window.jQuery = window.$ = jQuery;
```

You find this line at the bottom of the jQuery source file.

Another area where jQuery uses namespacing is with events. When attaching an event you can quality your event name with a namespace, something like this: `click.Framework.User`:

```
$("#name").on("click.Framework.User ", function() { alert("clonk"); };
```

This will add click to the Framework and User namespace. Namespaces with events are not really hierarchical; instead, each name is just a separate namespace. Our event is part of both the Framework namespace and the User namespace (and not the Framework.User namespace).

This mechanism allows you to selectively detach event handlers without affecting any other click handler:

```
$("#name").off("click.Framework"};
$("#name").off("click.User"};
```

The two lines above have the same effect, as our click event is part of both namespaces.

Lazy Load Pattern

The Lazy Load Pattern loads objects only when absolutely necessary. Its goal is to conserve memory and CPU cycles. jQuery uses this pattern in small ways. Here is a snippet from jQuery:

```
// not intended for public consumption.
// generates a queueHooks object, or returns the current one.
_queueHooks: function( elem, type ) {
    var key = type + "queueHooks";
    return jQuery._data( elem, key ) || jQuery._data( elem, key, {
        empty: jQuery.Callbacks("once memory").add(function() {
        jQuery.removeData( elem, type + "queue", true );
        jQuery.removeData( elem, key, true );
     })
});
```

This method is for private use only as indicated by its comment and the underscore (_) prefix which indicates that the variable or method is private.

The comments hint at the lazy loading aspect of the method: generate the object or if it exists return the current one. The return statement returns the result of jQuery._data and if that one does not exist (i.e. is falsy) then go off and create a new instance. This is a subtle use of Lazy Load.

Zero Timeout Pattern

In the Essentials Section of this program we discussed the event loop and the Zero Timeout pattern. This pattern is used to give the event loop time to catch up with pending events in the event queue. It an important pattern because it prevents the user interface from freezing up with long running scripts.

jQuery uses this pattern in several places. Here is a code snippet from jQuery's ready method:

```
// Make sure body exists, at least, in case IE gets a little overzealous.
if ( !document.body ) {
    return setTimeout( jQuery.ready, 1 );
}
```

Another instance in the `jQuery.ready.promise` method:

```
if ( document.readyState === "complete" || ( document.readyState !== "loading" &&
    document.addEventListener ) ) {
    // Handle it asynchronously to allow scripts the opportunity to delay ready
    setTimeout( jQuery.ready, 1 );
}
```

And here is a snippet from its Ajax functionality

```
} else if ( xhr.readyState === 4 ) {
    // (IE6 & IE7) if it's in cache and has been
    // retrieved directly we need to fire the callback
    setTimeout( callback, 0 );
}
```

And finally an asynchronous animation example

```
// Animations created synchronously will run synchronously
function createFxNow() {
     setTimeout(function() {
         fxNow = undefined;
     }, 0 );
     return ( fxNow = jQuery.now() );
}
```

In jQuery the Zero Timeout pattern is used to address some specific browser peculiarities as well as assist in asynchronous animations. It is interesting to note that both 0 and 1 millisecond are used for the delay. It really does not make a difference because JavaScript's internal minimum is apparently 4 milliseconds.

Singleton Pattern

Earlier we discussed the Module pattern in jQuery, which, as you know, is JavaScript's manifestation of the Singleton pattern. The jQuery code base lives in a single Module which executes only once. Here are the relevant lines:

```
(function (window, undefined) {

  // ~ 10,000 lines of code ...

})( window );
```

This ensures that a web page will have at most one instance of jQuery (or its alias $). As an aside, you can have multiple versions of jQuery running on a page with the help of the `jQuery.noConflict()` method.

Iterator Pattern

The Iterator Pattern allows traversal (or looping) over a collection of objects. Collections in JavaScript include arrays and the properties on objects. jQuery's implementation of the iterator pattern is the `each` method and handles both collection types as the example below demonstrates. Here is an example of where jQuery is used to iterate over an array:

```
var rhyme = ["Eeny", "Meeny", "Miny", "Moe"];
$.each(rhyme, function (index, value) {
    alert(index + ": " + value);    // => 0: Eeny, 1: Meeny, 2: Miny, 3: Moe
});
```

And here jQuery iterates over object properties:

```
var employee = {first: "Jonathan", last: "VanderHorn"};

$.each(employee, function (index, value) {
    alert(index + ": " + value);   // => first: Jonathan, last: VanderHorn
});
```

The source code of `each` shows that it is able to handle either type. These are the relevant lines in that method:

```
each: function( obj, callback, args ) {
    var name,
        i = 0,
        length = obj.length,
        isObj = length === undefined || jQuery.isFunction( obj );

    // ..

    if ( isObj ) {
        for ( name in obj ) {
            if ( callback.call( obj[ name ], name, obj[ name ] ) === false ) {
                break;
            }
        }
    } else {
        for ( ; i < length; ) {
            if ( callback.call( obj[ i ], i, obj[ i++ ] ) === false ) {
                break;
        }
    }
}
```

The iterator is not written from scratch with methods like: `current`, `next`, and `hasNext` as explained in the original GoF Iterator pattern. Instead it uses the built-in `for` and `for-in` constructs. In the code above we can see that the `if` block iterates over object properties (using for-in) and the else block iterates over arrays (using for).

Observer Pattern:

The Observer pattern allows functions to register themselves for notification when a particular event occurs. These functions are called event handlers. This pattern is at the core of all event-driven programming, including JavaScript. Not surprisingly then, the Observer pattern is a big part of jQuery: it has many dozens of objects, properties, and methods that are involved with events.

The newer `on` and `off` methods in jQuery are used to make it easy to attach and detach handlers to any event you specify. Here are some examples.

Attach a handler to the click event:

```
$("#menu").on("click", function (e) {
    alert("menu was clicked");
});
```

Remove all click handlers:

```
$("#menu").off("click");
```

The `trigger` method fires an event programmatically. All registered event handlers will execute.

```
$("#menu").trigger("click");
```

Proxy Pattern

The Proxy Pattern is an object that stands-in or represents another object. Searching for the keyword proxy in jQuery source file leads to a proxy method. This method takes a function and returns a new one which will always have a particular context (i.e., the `this` value in the function). The returned function is the proxy function object.

This is helpful in event handlers where there is a delay between calling and executing the event handler (and you don't want the overhead of building a closure). Consider the example below:

```
<div id="area" class="red"></div>

$("#area").on("click", function (e) {
    $(this).toggleClass("yellow");
});
```

Clicking the div element will toggle the area between red and yellow. The `$(this)` element refers to the element that was just clicked. Now, we wish to automate the toggling by using setInterval. Unfortunately this attempt fails:

```
<div id="area" class="red">The Area</div>

$("#area").on("click", function (e) {
    setInterval(function() {
        $(this).toggleClass("yellow");       // $(this) is the global object
    }, 1000);
});
```

Because of the delay, the function's context has changed and the `$(this)` value refers to the global window object. This is where the proxy method comes in. We create a proxy function that remembers (using its closure) the original `$(this)` context when it was first called. Now `setInterval` will call the proxy rather than the original function. Here is what this looks like:

```
<div id="area" class="red">The Area</div>
```

```
$("#area").on("click", function (e) {
    setInterval($.proxy(function () {
        $(this).toggleClass("yellow");
    }, this), 1000);
});
```

After the first click, the area toggles between red and yellow indefinitely (we did not build in a stop).

Note that you could have solved the above problem also by maintaining the original this value in a closure, like so:

```
<div id="area" class="red">The Area</div>

$("#area").on("click", function (e) {
    var that = this;
    setInterval(function () {
        that.toggleClass("yellow");
    }, 1000);
});
```

Fortunately, this feature has been standardized in ES5 (EcmaScript 5) in the form of the `bind` method (which was borrowed from the Prototype framework). It's an elegant way to assign and memorize a context to a function. Most modern browsers support it. Here is how it is used:

```
<div id="area" class="red">The Area</div>

$("#area").on("click", function (e) {
    setInterval(function () {
        $(this).toggleClass("yellow");
    }.bind(this), 1000);
});
```

The `bind` function creates a new function that, when called, has its `this` keyword set to the provided value. This is the proxy function that stands-in for the original function.

Façade Pattern

The Façade pattern provides an interface that shields a client from complex functionality. This definition appears to closely match jQuery's API which provides an easy-to-use interface to traverse the DOM, perform Ajax interactions, event handling, and animation. None of these things is simple and each requires a large and complex body of JavaScript code.

The jQuery library is cross-browser, that is, it supports all popular browsers. This involves a huge effort on the part of the authors of jQuery; not only are there several browsers to support (IE, Chrome, Safari, Firefox), but each has different versions with differences among them. There are subtle, and not so subtle, differences between all these browsers as well as bugs for which clever workarounds needed to be found.

In summary, the API of jQuery is an implementation of the Façade pattern.

Adapter Pattern

The Adapter pattern translates an object's interface (its properties and methods) to another interface. Adapters allow programming components to work together that otherwise wouldn't because of mismatched interfaces. The Adapter pattern is also referred to as the Wrapper Pattern.

In jQuery you attach an event handler to click, mouseover, keypress, focus, resize, scroll and other events like this:

```
$("#div").click(function() { alert("hello"); };

$("#name").focus(function() { alert("name please"); };
```

These are shorthand methods for the more generic bind method, which is used like this:

```
$("#div").bind("click", function() { alert("hello"); };
$("#name").bind("focus", function() { alert("name please"); };
```

In older versions of jQuery all shorthand methods were mapped to bind or trigger methods:

```
function click (fn) {
    return fn ? this.bind(name, fn) : this.trigger(name);
}
```

If an event handler (`fn`) is provided it is bound, but if none is provided it triggers the specific event. This mapping of shortcut methods to the more generic bind method is the Adapter Pattern in action.

This functionality continuous to be supported, but in recent versions two new methods `on` an `off` were introduced. They attach and detach event handlers respectively. The two methods unify all good practices of event handling and allow you to write nice tidy code. It is used just like the bind method above:

```
$("#div").on("click", function() { alert("hello"); };

$("#div").on("focus", function() { alert("name please"); };
```

Again the Adapter Pattern translates the call from `bind` to `on`. Here is a glimpse into jQuery source which shows how this is done:

```
jQuery.each(("blur focus focusin focusout load resize scroll unload click dblclick " +
    "mousedown mouseup mousemove mouseover mouseout mouseenter mouseleave " +
    "change select submit keydown keypress keyup error contextmenu").split(" "),
    function( i, name ) {
        // Handle event binding
        jQuery.fn[ name ] = function( data, fn ) {
            if ( fn == null ) {                      // argument switching
                fn = data;
                data = null;
            }

            return arguments.length > 0 ?
                this.on( name, null, data, fn ) :
                this.trigger( name );
       };
});
```

This code iterates over an array of shorthand method names and adds each one to jQuery.fn. They are mapped to either the on method or the trigger method depending on the arguments passed (trigger, as the name implies, triggers the event). Notice also that argument switching (function overload idiom) is used in these convenience methods.

Here is another jQuery Adapter in action. This time with the important ajax method:

```
ajax: function( url, options ) {
    // If url is an object, simulate pre-1.5 signature
    if ( typeof url === "object" ) {
        options = url;
        url = undefined;
    }
    // ...
}
```

This method is adapted to support/simulate older versions of the API. It does this through argument switching which allows it to support the current API and the older API. The fact that this is all handled in a single function is rather unusual.

jQuery Plugins

The jQuery library is an open system that from the beginning has promoted extensibility and reuse.

A common misconception is that jQuery plugins are only for open source projects to be shared with the community at large. If your motivation is to contribute your plugins or widgets to the open-source community then that is commendable, but don't forget that plugins can be very beneficial to your own projects as well.

Once you get the hang of constructing plugins you will realize that writing these is not much more difficult than writing regular JavaScript or jQuery code. The advantage you're getting is that they seamlessly integrate with jQuery: it is as if your enhancements shipped with the core library itself. A good example of this integration is chaining. Your custom methods can be chained together with any existing jQuery commands.

The flexibility of JavaScript allows you to overwrite, inherit, modify, and extend the jQuery library. However, jQuery provides an extension point, which is the `jQuery.fn` object where you are expected to add your plugin. By the way: `jQuery.fn` (or `$.fn`) is just an alias to `jQuery.prototype`, so at the end your plugin gets added as a method to jQuery's prototype object.

Common Plugin patterns

There are three commonly used patterns to structure a plugin, they are:

1. Function Plugin
2. Extend Plugin
3. Constructor Plugin

Function Plugin: The simplest (and most popular) way is to directly add a function to `$.fn`. Here is the code:

```
$.fn.myPlugin = function() {
    alert("My first plugin!");
};

$.myPlugin();          // => My first plugin!
```

Here is an example plugin you can use in the real-world:

```
$.fn.signal = function() {
    this.slideDown(300).delay(1000).slideUp(300);
};
```

And this is how you use it:

```
<div id="flash" style="height:100px;background:red;display:none;"></div>

$("#flash").signal();
```

This works great. It demonstrates how a simple plugin can make your jQuery life a whole lot easier. There is no point in sharing the above plugin as an open source project, but for your own purposes it is very useful.

A weakness of the above example is that it relies on $ being an alias for jQuery which may not always be the case. To protect against this you can wrap the plugin in an anonymous immediate function.

```
(function ($) {
    $.fn.myPlugin = function () {
        alert("My first plugin!");
    };
})(jQuery);

$.myPlugin();           // => My first plugin!
```

Much better: we can now be sure that within the closure the $ always refers to jQuery.

Extend Plugin: As an alternative you can use the jQuery's extend method:

```
(function ($) {
    $.extend($.fn, {                            // Extend
        myPlugin: function () {
            alert("My first plugin!");
        }
    });
})(jQuery);

$.myPlugin();           // => My first plugin!
```

The `extend` method extends `$.fn` with the properties in the object literal that is provided. Using `extend` allows you to assign multiple properties and methods in a single statement.

We should caution against adding multiple names to the `$.fn` namespace as this can lead to name collisions with jQuery itself or other plugins. You should consider wrapping your code base in its own namespace with a name that is unlikely to conflict with other names that may exist. You could use your project's name, company name, or domain name.

```
(function ($) {
    $.fn.MyCompany = {};
    $.extend($.fn MyCompany, {
        start: function () { alert("start "); },
        stop: function () { alert("stop "); },
        enable: function () { alert("enable "); },
        disable: function () { alert("disable "); },
    });
})(jQuery);

$.MyCompany.start();        // => start
$.MyCompany.enable();       // => enable
```

Here we created a namespace called MyCompany. It is the only name we're adding to `$.fn`. All our methods and properties are inside the MyCompany namespace and our impact on jQuery's `fn` namespace is now minimal.

Constructor Plugin: A third alternative is to use a private constructor function. Here is an example:

```
(function ($) {
    var Plugin = function (element) {        // Constructor
        this.element = element;
        this.$element = $(element);

        this.doSomething = function() {
            // ...
        }
    }

    $.fn.myPlugin = function () {
        return this.each(function() {
            if (!$.data(this, "myPlugin")) {
                $.data(this, "myPlugin", new Plugin(this));
            }
        };
    }
})(jQuery);
```

This pattern does more than just assigning a plugin object to `$.fn`. It also associates an object (a Plugin instance) with each element in jQuery's result set.

`Plugin` is a constructor function that accepts a DOM element as its argument. The function's name is `Plugin` but could be anything because it is never exposed to the outside world. The real plugin name is the one you assign to `$.fn` (`myPlugin` in our example). In `Plugin` we declare and initialize all necessary properties and methods.

The actual plugin function is added to `$.fn`. The context of the function is jQuery's result set which is returned to allow chaining with other jQuery methods. Just before returning `this` it iterates over each element in the result set using jQuery's built-in each method. The callback function that is passed into `each` checks if the element already has an attached `Plugin` instance; if not it creates one and stores it with the element. The `data` method in jQuery is a utility feature that stores arbitrary data with a specified element. Please note that within the callback function the value of `this` is bound to an element and not the result set.

By associating a `Plugin` instance to each element, we have the ability to maintain plugin state for each element. The state is nicely packaged in a single object, rather than a set of name/value pairs. It also makes the plugin easily reachable for each element, like so:

```
$("selector").data("myPlugin").doSomething();
```

Constructor Plugin Template

Following the three jQuery plugin patterns we'll now review a more comprehensive plugin that is based on the Constructor Plugin and that you can use as a template for your more advanced plugin projects. This template has evolved as a best-practice model and it is based on experiences from numerous members in the jQuery community. You'll recognize several other patterns and practices in this code.

Here's what the Plugin Template looks like:

```
;(function ($, win, undefined) {
    var name = "myPlugin",
        defaults = {
            option1: "value1",
            option2: "value2"
        };

    var Plugin = function (element, options) {
        this.element = element;
        this.$element = $(element);

        // handle different argument options
        if (typeof options === "object")
            this.settings = $.extend({}, defaults, options || {});
        } else if (typeof options === "string") {
            // any special string value processing ...
        }
        this._name = name;
        this._defaults = defaults;

        this.init();
    }

    Plugin.prototype.init = function () {
        // initialization code
    }

    $.fn[name] = function (options) {
        return this.each(function() {
            if (!$.data(this, name)) {
                $.data(this, name, new Plugin(this, options));
            }
        };
    }

})(jQuery, window);
```

The first thing you notice is the leading semicolon. The Leading Semicolon idiom prevents improperly closed code from interfering with our module.

The list of parameters in the immediate function (the Module Pattern) has increased to three. The $ parameter ensures that $ is an alias for jQuery within the module. The window parameter is the global object and is available as a local argument which reduces the scope traversing of the JavaScript runtime. Finally, the undefined parameter ensures that undefined really means undefined and not something else (in case some hacker decided to reassign its value).

Next, we have the Single var pattern in which two variables are declared and initialized: `name`, which is the name of the plugin and `defaults` which holds name/value pairs with default values for all options. This `defaults` variable is part of the Options Hash idiom we discussed in the Essentials section. Also note that the Single var is not 'pure' because there is another var for the Plugin function.

The `Plugin` function is a constructor function. It accepts a DOM element and a set of options. The function's name is Plugin but it can be anything else because its name is not exposed to the outside world. Remember, the real plugin name is stored in the name variable.

In Plugin we declare and initialize all necessary properties: `element` and `$element` (we save element as both a normal reference and a jQuery reference to speed up processing); the `settings` (built using the Options Hash idiom), and two private properties `_name` and `_defaults` which store the plugin name and default values locally. Finally, the init method is invoked which will initialize the plugin further now that all properties are properly set.

To conserve memory all Plugin methods are associated with its prototype rather than with the instance itself. This ensures that all methods are shared. Here we only have an `init` method but you can add additional methods if necessary.

At the end of the template we are adding the plugin function to jQuery's `$.fn`. This function has a single parameter named `options`. There is no need for an element parameter because the context (`this` in the function) refers to the result set which has all relevant elements. We will see an example of how this is used shortly.

The remainder of the plugin function is similar to the earlier Constructor Plugin Pattern. The only difference is that here we are passing an extra `options` argument into the new Plugin constructor call.

Mother of all Plugins

After you have written several plugins using the above boilerplate templates you may find that the process becomes rather repetitive. You are wondering if there is not a more generic solution to writing plugins: something along the lines of a base class (or prototype in JavaScript) which has all the boilerplate code and you just fill in the details.

It turns out there is indeed a more generic solution which has been referred to as the DOM-to-Object Bridge Pattern. This sounds awkward, so we will refer to it as the Mother of all Plugins pattern.

Actually, if you understand the prior template we discussed, then this pattern is fairly straightforward. You wrap the actual registration with $.fn in a new function called plugin. It accepts the object name which contains the plugin functionality and the rest is quite similar to the template discussed before. Here is the skeleton code:

```
;(function ($, win, undefined) {

    $.fn.plugin = function (name, object) {
        $.fn[name] = function (options) {
            return this.each(function() {
                if (!$.data(this, name)) {
                    $.data(this, name, create(object).init(this, options));
                }
            };
        },
    }
    function create (obj) {    // In newer browsers replace with Object.create
        function F() {};
        F.prototype = obj;
        return new F();
    }

})(jQuery, window);
```

Your plugin object (which does not have to deal with plugin plumbing) is an object literal that has the following structure (again similar to the Constructor Function plugin patterns):

```
var myPlugin = {
    init: function (element, options) {
        this.element = element;
        this.$element = $(element);
        this.settings = $.extend({}, this.defaults, options || {} );

        _init();
    },
    defaults: {
       color: "Yellow",
       speed: 100;
    },
    _init() {
        // internal initialization
    },
    sayColor() {               // public method
        alert(settings.color);
    }
};
```

Here is an example of how this is used:

```
$('selection').plugin("myPlug", myPlugin);     // register with plugin plumbing

$('selection').myPlug( {color: "Green" });
$('selection').myPlug.sayColor();              // => Green
```

Plugin File names

Plugins reside in their own JavaScript files and it is important that you follow generally accepted naming conventions. Here are examples of the source and the minified versions:

```
jquery.pluginname.js
jquery.pluginname.min.js
```

The names are always in lowercase.

If shared with the open source community, include a version number:

```
jquery.pluginname-2.1.js
jquery.pluginname-2.1.min.js
```

If your project team builds multiple plugins, include another namespace which will keep them together:

```
jquery.projectname.pluginname1.js
jquery.projectname.pluginname2.js
```