Posted in: Plugins (https://learn.jquery.com/plugins/)

How to Create a Basic Plugin

Sometimes you want to make a piece of functionality available throughout your code. For example, perhaps you want a single method you can call on a jQuery selection that performs a series of operations on the selection. In this case, you may want to write a plugin.

How jQuery Works 101: jQuery Object Methods

Before we write our own plugins, we must first understand a little about how jQuery works. Take a look at this code:

```
$( "a" ).css( "color", "red" );
```

This is some pretty basic jQuery code, but do you know what's happening behind the scenes? Whenever you use the `$` function to select elements, it returns a jQuery object. This object contains all of the methods you've been using (`.css()`, `.click()`, etc.) and all of the elements that fit your selector. The jQuery object gets these methods from the `$.fn` object. This object contains all of the jQuery object methods, and if we want to write our own methods, it will need to contain those as well.

Basic Plugin Authoring

Let's say we want to create a plugin that makes text within a set of retrieved elements green. All we have to do is add a function called `greenify` to `$.fn` and it will be available just like any other jQuery object method.

```
$.fn.greenify = function() {
    this.css( "color", "green" );
};

$( "a" ).greenify(); // Makes all the links green.
```

Notice that to use `.css()`, another method, we use `this`, not `$( this )`. This is because our `greenify` function is a part of the same object as `.css()`.

Chaining

This works, but there are a couple of things we need to do for our plugin to survive in the real world. One of jQuery's features is chaining, when you link five or six actions onto one selector. This is accomplished by having all jQuery object methods return the original jQuery object again (there are a few exceptions: `.width()` called without parameters returns the width of the selected element, and is not chainable). Making our plugin method chainable takes one line of code:

```
$.fn.greenify = function() {
    this.css( "color", "green" );
    return this;
}

$( "a" ).greenify().addClass( "greenified" );
```

Protecting the $ Alias and Adding Scope

The `$` variable is very popular among JavaScript libraries, and if you're using another library with jQuery, you will have to make jQuery not use the `$` with `jQuery.noConflict()`. However, this will break our plugin since it is written with the assumption that `$` is an alias to the `jQuery` function. To work well with other plugins, *and* still use the jQuery `$` alias, we need to put all of our code inside of an Immediately Invoked Function Expression (http://benalman.com/news/2010/11/immediately-invoked-function-expression/), and then pass the function `jQuery`, and name the parameter `$`:

```
(function ( $ ) {

    $.fn.greenify = function() {
        this.css( "color", "green" );
        return this;
    };

}( jQuery ));
```

In addition, the primary purpose of an Immediately Invoked Function is to allow us to have our own private variables. Pretend we want a different color green, and we want to store it in a variable.

```
(function ( $ ) {

    var shade = "#556b2f";

    $.fn.greenify = function() {
        this.css( "color", shade );
        return this;
    };

}( jQuery ));
```

Minimizing Plugin Footprint

It's good practice when writing plugins to only take up one slot within `$.fn`. This reduces both the chance that your plugin will be overridden, and the chance that your plugin will override other plugins. In other words, this is bad:

```
(function( $ ) {

    $.fn.openPopup = function() {
        // Open popup code.
    };

    $.fn.closePopup = function() {
        // Close popup code.
    };

}( jQuery ));
```

It would be much better to have one slot, and use parameters to control what action that one slot performs.

```
(function( $ ) {

    $.fn.popup = function( action ) {

        if ( action === "open") {
            // Open popup code.
        }

        if ( action === "close" ) {
            // Close popup code.
        }

    };

}( jQuery ));
```

Using the `each()` Method

Your typical jQuery object will contain references to any number of DOM elements, and that's why jQuery objects are often referred to as collections. If you want to do any manipulating with specific elements (e.g. getting a data attribute, calculating specific positions) then you need to use `.each()` to loop through the elements.

```
$.fn.myNewPlugin = function() {

    return this.each(function() {
        // Do something to each element here.
    });

};
```

Notice that we return the results of `.each()` instead of returning `this`. Since `.each()` is already chainable, it returns `this`, which we then return. This is a better way to maintain chainability than what we've been doing so far.

Accepting Options

As your plugins get more and more complex, it's a good idea to make your plugin customizable by accepting options. The easiest way to do this, especially if there are lots of options, is with an object literal. Let's change our greenify plugin to accept some options.

```
(function ( $ ) {

    $.fn.greenify = function( options ) {

        // This is the easiest way to have default options.
        var settings = $.extend({
            // These are the defaults.
            color: "#556b2f",
            backgroundColor: "white"
        }, options );

        // Greenify the collection based on the settings variable.
        return this.css({
            color: settings.color,
            backgroundColor: settings.backgroundColor
        });

    };

}( jQuery ));
```

Example usage:

```
$( "div" ).greenify({
    color: "orange"
});
```

The default value for `color` of `#556b2f` gets overridden by `$.extend()` to be orange.

Putting It Together

Here's an example of a small plugin using some of the techniques we've discussed:

```
(function( $ ) {

    $.fn.showLinkLocation = function() {

        this.filter( "a" ).each(function() {
            var link = $( this );
            link.append( " (" + link.attr( "href" ) + ")" );
        });

        return this;

    };

}( jQuery ));

// Usage example:
$( "a" ).showLinkLocation();
```

This handy plugin goes through all anchors in the collection and appends the `href` attribute in parentheses.

```
<!-- Before plugin is called: -->
<a href="page.html">Foo</a>

<!-- After plugin is called: -->
<a href="page.html">Foo (page.html)</a>
```

Our plugin can be optimized though:

```
(function( $ ) {

    $.fn.showLinkLocation = function() {

        this.filter( "a" ).append(function() {
            return " (" + this.href + ")";
        });

        return this;

    };

}( jQuery ));
```

We're using the `.append()` method's capability to accept a callback, and the return value of that callback will determine what is appended to each element in the collection. Notice also that we're not using the `.attr()` method to retrieve the `href` attribute, because the native DOM API gives us easy access with the aptly named `href` property.

Finding & Evaluating Plugins (https://learn.jquery.com/plugins/finding-evaluating-plugins/)

Advanced Plugin Concepts (https://learn.jquery.com/plugins/advanced-plugin-concepts/)

Last Updated

December 17, 2015

Suggestions, Problems, Feedback?

Open an Issue or Submi
(https://github.com/jquery/learn.jquery.com/tree/m