Chapters

About jQuery (//learn.jquery.com/about-jquery/)
Using jQuery Core (//learn.jquery.com/using-jquery-core/)
Frequently Asked Questions (//learn.jquery.com/using-jquery-core/faq/)
Events (//learn.jquery.com/events/)
Effects (//learn.jquery.com/effects/)
Ajax (//learn.jquery.com/ajax/)
Plugins (//learn.jquery.com/plugins/)
Performance (//learn.jquery.com/performance/)
Code Organization (//learn.jquery.com/code-organization/)
Deferreds (//learn.jquery.com/code-organization/deferreds/)
jQuery UI (//learn.jquery.com/jquery-ui/)
Widget Factory (//learn.jquery.com/jquery-ui/widget-factory/)
Using jQuery UI (//learn.jquery.com/jquery-ui/environments/)
jQuery Mobile (//learn.jquery.com/jquery-mobile/)

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

Advanced Plugin Concepts

Provide Public Access to Default Plugin Settings

An improvement we can, and should, make to the code above is to expose the default plugin settings. This is important because it makes it very easy for plugin users to override/customize the plugin with minimal code. And this is where we begin to take advantage of the function object.

```
// Plugin definition.
$.fn.hilight = function( options ) {

    // Extend our default options with those provided.
    // Note that the first argument to extend is an empty
    // object - this is to keep from overriding our "defaults" object.
    var opts = $.extend( {}, $.fn.hilight.defaults, options );

    // Our plugin implementation code goes here.

};

// Plugin defaults - added as a property on our plugin function.
$.fn.hilight.defaults = {
    foreground: "red",
    background: "yellow"
};
```

Now users can include a line like this in their scripts:

```
// This needs only be called once and does not
// have to be called from within a "ready" block
$.fn.hilight.defaults.foreground = "blue";
```

And now we can call the plugin method like this and it will use a blue foreground color:

```
$( "#myDiv" ).hilight();
```

As you can see, we've allowed the user to write a single line of code to alter the default foreground color of the plugin. And users can still selectively override this new default value when they want:

```
// Override plugin default foreground color.
$.fn.hilight.defaults.foreground = "blue";

// ...

// Invoke plugin using new defaults.
$( ".hilightDiv" ).hilight();

// ...

// Override default by passing options to plugin method.
$( "#green" ).hilight({
    foreground: "green"
});
```

Provide Public Access to Secondary Functions as Applicable

This item goes hand-in-hand with the previous item and is an interesting way to extend your plugin (and to let others extend your plugin). For example, the implementation of our plugin may define a function called "format" which formats the hilight text. Our plugin may now look like this, with the default implementation of the format method defined below the hilight function:

```
// Plugin definition.
$.fn.hilight = function( options ) {

    // Iterate and reformat each matched element.
    return this.each(function() {

        var elem = $( this );

        // ...

        var markup = elem.html();

        // Call our format function.
        markup = $.fn.hilight.format( markup );

        elem.html( markup );

    });

};

// Define our format function.
$.fn.hilight.format = function( txt ) {
    return "<strong>" + txt + "</strong>";
};
```

We could have just as easily supported another property on the options object that allowed a callback function to be provided to override the default formatting. That's another excellent way to support customization of your plugin. The technique shown here takes this a step further by actually exposing the format function so that it can be redefined. With this technique it would be possible for others to ship their own custom overrides of your plugin – in other words, it means others can write plugins for your plugin.

Considering the trivial example plugin we're building in this article, you may be wondering when this would ever be useful. One real-world example is the Cycle Plugin (http://malsup.com/jquery/cycle/). The Cycle Plugin is a slideshow plugin which supports a number of built-in transition effects – scroll, slide, fade, etc. But realistically, there is no way to define every single type of effect that one might wish to apply to a slide transition. And that's where this type of extensibility is useful. The Cycle Plugin exposes a "transitions" object to which users can add their own custom transition definitions. It's defined in the plugin like this:

```
$.fn.cycle.transitions = {

    // ...

};
```

This technique makes it possible for others to define and ship transition definitions that plug-in to the Cycle Plugin.

Keep Private Functions Private

The technique of exposing part of your plugin to be overridden can be very powerful. But you need to think carefully about what parts of your implementation to expose. Once it's exposed, you need to keep in mind that any changes to the calling arguments or semantics may break backward compatibility. As a general rule, if you're not sure whether to expose a particular function, then you probably shouldn't.

So how then do we define more functions without cluttering the namespace and without exposing the implementation? This is a job for closures. To demonstrate, we'll add another function to our plugin called "debug". The debug function will log the number of selected elements to the console. To create a closure, we wrap the entire plugin definition in a function (as detailed in the jQuery Authoring Guidelines).

```
// Create closure.
(function( $ ) {

    // Plugin definition.
    $.fn.hilight = function( options ) {
        debug( this );
        // ...
    };

    // Private function for debugging.
    function debug( obj ) {
        if ( window.console && window.console.log ) {
            window.console.log( "hilight selection count: " + obj.length );
        }
```

```
    };

    // ...

// End of closure.

})( jQuery );
```

Our "debug" method cannot be accessed from outside of the closure and thus is private to our implementation.

Bob and Sue

Let's say Bob has created a wicked new gallery plugin (called "superGallery") which takes a list of images and makes them navigable. Bob's thrown in some animation to make it more interesting. He's tried to make the plugin as customizable as possible, and has ended up with something like this:

```
jQuery.fn.superGallery = function( options ) {

    // Bob's default settings:
    var defaults = {
        textColor: "#000",
        backgroundColor: "#fff",
        fontSize: "1em",
        delay: "quite long",
        getTextFromTitle: true,
        getTextFromRel: false,
        getTextFromAlt: false,
        animateWidth: true,
        animateOpacity: true,
        animateHeight: true,
        animationDuration: 500,
        clickImgToGoToNext: true,
        clickImgToGoToLast: false,
        nextButtonText: "next",
        previousButtonText: "previous",
        nextButtonTextColor: "red",
        previousButtonTextColor: "red"
    };

    var settings = $.extend( {}, defaults, options );

    return this.each(function() {
        // Plugin code would go here...
    });

};
```

The first thing that probably comes to your mind (OK, maybe not the first) is the prospect of how huge this plugin must be to accommodate such a level of customization. The plugin, if it weren't fictional, would probably be a lot larger than necessary. There are only so many kilobytes people will be willing to spend!

Now, our friend Bob thinks this is all fine; in fact, he's quite impressed with the plugin and its level of customization. He believes that all the options make for a more versatile solution, one which can be used in many different situations.

Sue, another friend of ours, has decided to use this new plugin. She has set up all of the options required and now has a working solution sitting in front of her. It's only five minutes later, after playing with the plugin, that she realizes the gallery would look much nicer if each image's width were animated at a slower speed. She hastily searches through Bob's documentation but finds no *animateWidthDuration* option!

Do You See The Problem?

It's not really about how many options your plugin has; but what options it has!

Bob has gone a little over the top. The level of customization he's offering, while it may seem high, is actually quite low, especially considering all the possible things one might want to control when using this plugin. Bob has made the mistake of offering a lot of ridiculously specific options, rendering his plugin much more difficult to customize!

A Better Model

So it's pretty obvious: Bob needs a new customization model, one which does not relinquish control or abstract away the necessary details.

The reason Bob is so drawn to this high-level simplicity is that the jQuery framework very much lends itself to this mindset. Offering a *previousButtonTextColor* option is nice and simple, but let's face it, the vast majority of plugin users are going to want way more control!

Here are a few tips which should help you create a better set of customizable options for your plugins:

Don't Create Plugin-specific Syntax

Developers who use your plugin shouldn't have to learn a new language or terminology just to get the job done.

Bob thought he was offering maximum customization with his *delay* option (look above). He made it so that with his plugin you can specify four different delays, "quite short," "very short," "quite long," or "very long":

```
var delayDuration = 0;

switch ( settings.delay ) {

    case "very short":
        delayDuration = 100;
        break;

    case "quite short":
        delayDuration = 200;
        break;

    case "quite long":
        delayDuration = 300;
        break;

    case "very long":
        delayDuration = 400;
        break;

    default:
        delayDuration = 200;

}
```

Not only does this limit the level of control people have, but it takes up quite a bit of space. Twelve lines of code just to define the delay time is a bit much, don't you think? A better way to construct this option would be to let plugin users specify the amount of time (in milliseconds) as a number, so that no processing of the option needs to take place.

The key here is not to diminish the level of control through your abstraction. Your abstraction, whatever it is, can be as simplistic as you want, but make sure that people who use your plugin will still have that much-sought-after low-level control! (By low-level I mean non-abstracted.)

Give Full Control of Elements

If your plugin creates elements to be used within the DOM, then it's a good idea to offer plugin users some way to access those elements. Sometimes this means giving certain elements IDs or classes. But note that your plugin shouldn't rely on these hooks internally:

A bad implementation:

```
// Plugin code
$( "<div class='gallery-wrapper' />" ).appendTo( "body" );

$( ".gallery-wrapper" ).append( "..." );
```

To allow users to access and even manipulate that information, you can store it in a variable containing the settings of your plugin. A better implementation of the previous code is shown below:

```
// Retain an internal reference:
var wrapper = $( "<div />" )
    .attr( settings.wrapperAttrs )
    .appendTo( settings.container );

// Easy to reference later...
wrapper.append( "..." );
```

Notice that we've created a reference to the injected wrapper and we're also calling the `.attr()` method to add any specified attributes to the element. So, in our settings it might be handled like this:

```
var defaults = {
    wrapperAttrs : {
        class: "gallery-wrapper"
    },
    // ... rest of settings ...
};

// We can use the extend method to merge options/settings as usual:
// But with the added first parameter of TRUE to signify a DEEP COPY:
var settings = $.extend( true, {}, defaults, options );
```

The *$.extend()* method will now recurse through all nested objects to give us a merged version of both the defaults and the passed options, giving the passed options precedence.

The plugin user now has the power to specify any attribute of that wrapper element so if they require that there be a hook for any CSS styles then they can quite easily add a class or change the name of the ID without having to go digging around in plugin source.

The same model can be used to let the user define CSS styles:

```
var defaults = {
    wrapperCSS: {},
    // ... rest of settings ...
};

// Later on in the plugin where we define the wrapper:
var wrapper = $( "<div />" )
    .attr( settings.wrapperAttrs )
    .css( settings.wrapperCSS ) // ** Set CSS!
    .appendTo( settings.container );
```

Your plugin may have an associated stylesheet where developers can add CSS styles. Even in this situation it's a good idea to offer some convenient way of setting styles in JavaScript, without having to use a selector to get at the elements.

Provide Callback Capabilities

What is a callback? – A callback is essentially a function to be called later, normally triggered by an event. It's passed as an argument, usually to the initiating call of a component, in this case, a jQuery plugin.

If your plugin is driven by events then it might be a good idea to provide a callback capability for each event. Plus, you can create your own custom events and then provide callbacks for those. In this gallery plugin it might make sense to add an "onImageShow" callback.

```
var defaults = {

    // We define an empty anonymous function so that
    // we don't need to check its existence before calling it.
    onImageShow : function() {},

    // ... rest of settings ...

};

// Later on in the plugin:

nextButton.on( "click", showNextImage );

function showNextImage() {

    // Returns reference to the next image node
    var image = getNextImage();

    // Stuff to show the image here...

    // Here's the callback:
    settings.onImageShow.call( image );
}
```

Instead of initiating the callback via traditional means (adding parenthesis) we're calling it in the context of `image` which will be a reference to the image node. This means that you have access to the actual image node through the `this` keyword within the callback:

```
$( "ul.imgs li" ).superGallery({
    onImageShow: function() {
        $( this ).after( "<span>" + $( this ).attr( "longdesc" ) + "</span>" )
    },

    // ... other options ...
});
```

Similarly you could add an "onImageHide" callback and numerous other ones. The point of callbacks is to give plugin users an easy way to add additional functionality without digging around in the source.

Remember, It's a Compromise

Your plugin is not going to be able to work in every situation. And equally, it's not going to be very useful if you offer no or very few methods of control. So, remember, it's always going to be a compromise. Three things you must always take into account are:

Flexibility: How many situations will your plugin be able to deal with?

Size: Does the size of your plugin correspond to its level of functionality? I.e. Would you use a very basic tooltip plugin if it was 20k in size? – Probably not!

Performance: Does your plugin heavily process the options in any way? Does this affect speed? Is the overhead caused worth it for the end user?

‹ How to Create a Basic Plugin (https://learn.jquery.com/plugins/basic-plugin-creation/)

Writing Stateful Plugins with the jQuery UI Widget Factory › (https://learn.jquery.com/plugins/stateful-plugins-with-widget-factory/)

Last Updated

December 17, 2015

Suggestions, Problems, Feedback?

Open an Issue or Sub (https://github.com/jquery/learn.jquery.com/tree/mas