<http://docs.jquery.com/Plugins/Authoring>

**Plugins/Authoring**

So you've become comfortable with jQuery and would like to learn how to write your own plugins. Great! You're in the right spot. Extending jQuery with plugins and methods is very powerful and can save you and your peers a lot of development time by abstracting your most clever functions into plugins. This post will outline the basics, best practices, and common pitfalls to watch out for as you begin writing your plugin.

|  |
| --- |
| **CONTENTS**   * [1 Getting Started](http://docs.jquery.com/Plugins/Authoring#Getting_Started) * [2 Context](http://docs.jquery.com/Plugins/Authoring#Context) * [3 The Basics](http://docs.jquery.com/Plugins/Authoring#The_Basics) * [4 Maintaining Chainability](http://docs.jquery.com/Plugins/Authoring#Maintaining_Chainability) * [5 Defaults and Options](http://docs.jquery.com/Plugins/Authoring#Defaults_and_Options) * [6 Namespacing](http://docs.jquery.com/Plugins/Authoring#Namespacing)   + [6.1 Plugin Methods](http://docs.jquery.com/Plugins/Authoring#Plugin_Methods)   + [6.2 Events](http://docs.jquery.com/Plugins/Authoring#Events)   + [6.3 Data](http://docs.jquery.com/Plugins/Authoring#Data) * [7 Summary and Best Practices](http://docs.jquery.com/Plugins/Authoring#Summary_and_Best_Practices) |

[[edit](http://docs.jquery.com/action/edit/Plugins/Authoring?section=1)]

**Getting Started**

To write a jQuery plugin, start by adding a new function property to the jQuery.fn object where the name of the property is the name of your plugin:

jQuery.fn.myPlugin = function() {

// Do your awesome plugin stuff here

};

But wait! Where's my awesome dollar sign that I know and love? It's still there, however to make sure that your plugin doesn't collide with other libraries that might use the dollar sign, it's a best practice to pass jQuery to an IIFE (Immediately Invoked Function Expression) that maps it to the dollar sign so it can't be overwritten by another library in the scope of its execution.

(function( $ ) {

$.fn.myPlugin = function() {

// Do your awesome plugin stuff here

};

})( jQuery );

Ah, that's better. Now within that closure, we can use the dollar sign in place of jQuery as much as we like.

[[edit](http://docs.jquery.com/action/edit/Plugins/Authoring?section=2)]

**Context**

Now that we have our shell we can start writing our actual plugin code. But before we do that, I'd like to say a word about context. In the immediate scope of the plugin function, the this keyword refers to the jQuery object the plugin was invoked on. This is a common slip up due to the fact that in other instances where jQuery accepts a callback, the this keyword refers to the native DOM element. This often leads to developers unnecessarily wrapping the this keyword (again) in the jQuery function.

(function( $ ){

$.fn.myPlugin = function() {

// there's no need to do $(this) because

// "this" is already a jquery object

// $(this) would be the same as $($('#element'));

this.fadeIn('normal', function(){

// the this keyword is a DOM element

});

};

})( jQuery );

$('#element').myPlugin();

[[edit](http://docs.jquery.com/action/edit/Plugins/Authoring?section=3)]

**The Basics**

Now that we understand the context of jQuery plugins, let's write a plugin that actually does something.

(function( $ ){

$.fn.maxHeight = function() {

var max = 0;

this.each(function() {

max = Math.max( max, $(this).height() );

});

return max;

};

})( jQuery );

var tallest = $('div').maxHeight(); // Returns the height of the tallest div

This is a simple plugin that leverages [.height()](http://docs.jquery.com/action/edit/Manipulation/height) to return the height of the tallest div in the page.

[[edit](http://docs.jquery.com/action/edit/Plugins/Authoring?section=4)]

**Maintaining Chainability**

The previous example returns an integer value of the tallest div on the page, but often times the intent of a plugin is simply modify the collection of elements in some way, and pass them along to the next method in the chain. This is the beauty of jQuery's design and is one of the reasons jQuery is so popular. So to maintain chainability in a plugin, you must make sure your plugin returns thethis keyword.

(function( $ ){

$.fn.lockDimensions = function( type ) {

return this.each(function() {

var $this = $(this);

if ( !type || type == 'width' ) {

$this.width( $this.width() );

}

if ( !type || type == 'height' ) {

$this.height( $this.height() );

}

});

};

})( jQuery );

$('div').lockDimensions('width').css('color', 'red');

Because the plugin returns the this keyword in its immediate scope, it maintains chainability and the jQuery collection can continue to be manipulated by jQuery methods, such as [.css](http://docs.jquery.com/CSS/css). So if your plugin doesn't return an intrinsic value, you should always return the this keyword in the immediate scope of the plugin function. Also, as you might assume, arguments you pass in your plugin invocation get passed to the immediate scope of the plugin function. So in the previous example, the string 'width' becomes the type argument for the plugin function.

[[edit](http://docs.jquery.com/action/edit/Plugins/Authoring?section=5)]

**Defaults and Options**

For more complex and customizable plugins that provide many options, it's a best practice to have default settings that can get extended (using [$.extend](http://docs.jquery.com/Utilities/jQuery.extend)) when the plugin is invoked. So instead of calling a plugin with a large number of arguments, you can call it with one argument which is an object literal of the settings you would like to override. Here's how you do it.

(function( $ ){

$.fn.tooltip = function( options ) {

// Create some defaults, extending them with any options that were provided

var settings = $.extend( {

'location'  : 'top',

'background-color' : 'blue'

}, options);

return this.each(function() {

// Tooltip plugin code here

});

};

})( jQuery );

$('div').tooltip({

'location' : 'left'

});

In this example, after calling the tooltip plugin with the given options, the default location setting gets overridden to become 'left', while the background-color setting remains the default'blue'. So the final settings object ends up looking like this:

{

'location'  : 'left',

'background-color' : 'blue'

}

This is a great way to offer a highly configurable plugin without requiring the developer to define all available options.

[[edit](http://docs.jquery.com/action/edit/Plugins/Authoring?section=6)]

**Namespacing**

Properly namespacing your plugin is a very important part of plugin development. Namespacing correctly assures that your plugin will have a very low chance of being overwritten by other plugins or code living on the same page. Namespacing also makes your life easier as a plugin developer because it helps you keep better track of your methods, events and data.

[[edit](http://docs.jquery.com/action/edit/Plugins/Authoring?section=7)]

**Plugin Methods**

Under no circumstance should a single plugin ever claim more than one namespace in thejQuery.fn object.

(function( $ ){

$.fn.tooltip = function( options ) {

// THIS

};

$.fn.tooltipShow = function( ) {

// IS

};

$.fn.tooltipHide = function( ) {

// BAD

};

$.fn.tooltipUpdate = function( content ) {

// !!!

};

})( jQuery );

This is a discouraged because it clutters up the $.fn namespace. To remedy this, you should collect all of your plugin's methods in an object literal and call them by passing the string name of the method to the plugin.

(function( $ ){

var methods = {

init : function( options ) {

// THIS

},

show : function( ) {

// IS

},

hide : function( ) {

// GOOD

},

update : function( content ) {

// !!!

}

};

$.fn.tooltip = function( method ) {

// Method calling logic

if ( methods[method] ) {

return methods[ method ].apply( this, Array.prototype.slice.call( arguments, 1 ));

} else if ( typeof method === 'object' || ! method ) {

return methods.init.apply( this, arguments );

} else {

$.error( 'Method ' + method + ' does not exist on jQuery.tooltip' );

}

};

})( jQuery );

// calls the init method

$('div').tooltip();

// calls the init method

$('div').tooltip({

foo : 'bar'

});

// calls the hide method

$('div').tooltip('hide');

// calls the update method

$('div').tooltip('update', 'This is the new tooltip content!');

This type of plugin architecture allows you to encapsulate all of your methods in the plugin's parent closure, and call them by first passing the string name of the method, and then passing any additional parameters you might need for that method. This type of method encapsulation and architecture is a standard in the jQuery plugin community and it used by countless plugins, including the plugins and widgets in [jQueryUI](http://jqueryui.com/" \o "http://jqueryui.com/).

[[edit](http://docs.jquery.com/action/edit/Plugins/Authoring?section=8)]

**Events**

A lesser known feature of the [bind](http://docs.jquery.com/Events/bind) method is that is allows for namespacing of bound events. If your plugin binds an event, its a good practice to namespace it. This way, if you need to [unbind](http://docs.jquery.com/Events/unbind) it later, you can do so without interfering with other events that might have been bound to the same type of event. You can namespace your events by appending “.<namespace>” to the type of event you're binding.

(function( $ ){

var methods = {

init : function( options ) {

return this.each(function(){

$(window).bind('resize.tooltip', methods.reposition);

});

},

destroy : function( ) {

return this.each(function(){

$(window).unbind('.tooltip');

})

},

reposition : function( ) {

// ...

},

show : function( ) {

// ...

},

hide : function( ) {

// ...

},

update : function( content ) {

// ...

}

};

$.fn.tooltip = function( method ) {

if ( methods[method] ) {

return methods[method].apply( this, Array.prototype.slice.call( arguments, 1 ));

} else if ( typeof method === 'object' || ! method ) {

return methods.init.apply( this, arguments );

} else {

$.error( 'Method ' + method + ' does not exist on jQuery.tooltip' );

}

};

})( jQuery );

$('#fun').tooltip();

// Some time later...

$('#fun').tooltip('destroy');

In this example, when the tooltip is initialized with the init method, it binds the reposition method to the resize event of the window under the namespace 'tooltip'. Later, if the developer needs to destroy the tooltip, we can unbind the events bound by the plugin by passing its namespace, in this case 'tooltip', to the unbind method. This allows us to safely unbind plugin events without accidentally unbinding events that may have been bound outside of the plugin.

[[edit](http://docs.jquery.com/action/edit/Plugins/Authoring?section=9)]

**Data**

Often times in plugin development, you may need to maintain state or check if your plugin has already been initialized on a given element. Using jQuery's [data](http://docs.jquery.com/action/edit/Utilities/data) method is a great way to keep track of variables on a per element basis. However, rather than keeping track of a bunch of separate data calls with different names, it's best to use a single object literal to house all of your variables, and access that object by a single data namespace.

(function( $ ){

var methods = {

init : function( options ) {

return this.each(function(){

var $this = $(this),

data = $this.data('tooltip'),

tooltip = $('<div />', {

text : $this.attr('title')

});

// If the plugin hasn't been initialized yet

if ( ! data ) {

/\*

Do more setup stuff here

\*/

$(this).data('tooltip', {

target : $this,

tooltip : tooltip

});

}

});

},

destroy : function( ) {

return this.each(function(){

var $this = $(this),

data = $this.data('tooltip');

// Namespacing FTW

$(window).unbind('.tooltip');

data.tooltip.remove();

$this.removeData('tooltip');

})

},

reposition : function( ) { // ... },

show : function( ) { // ... },

hide : function( ) { // ... },

update : function( content ) { // ...}

};

$.fn.tooltip = function( method ) {

if ( methods[method] ) {

return methods[method].apply( this, Array.prototype.slice.call( arguments, 1 ));

} else if ( typeof method === 'object' || ! method ) {

return methods.init.apply( this, arguments );

} else {

$.error( 'Method ' + method + ' does not exist on jQuery.tooltip' );

}

};

})( jQuery );

Using data helps you keep track of variables and state across method calls from your plugin. Namespacing your data into one object literal makes it easy to access all of your plugin's properties from one central location, as well as reducing the data namespace which allows for easy removal if need be.

[[edit](http://docs.jquery.com/action/edit/Plugins/Authoring?section=10)]

**Summary and Best Practices**

Writing jQuery plugins allows you to make the most out of the library and abstract your most clever and useful functions out into reusable code that can save you time and make your development even more efficient. Here's a brief summary of the post and what to keep in mind when developing your next jQuery plugin:

* Always wrap your plugin in a closure: (function( $ ){ /\* plugin goes here \*/ })( jQuery );
* Don't redundantly wrap the this keyword in the immediate scope of your plugin's function
* Unless you're returning an intrinsic value from your plugin, always have your plugin's function return the this keyword to maintain chainability.
* Rather than requiring a lengthy amount of arguments, pass your plugin settings in an object literal that can be extended over the plugin's defaults.
* Don't clutter the jQuery.fn object with more than one namespace per plugin.
* Always namespace your methods, events and data.