<http://www.authenticsociety.com/blog/jQueryPluginTutorial_Beginner>

**How to Make a jQuery Plugin - Tutorial for Beginners**

Article written by [Greg Sidelnikov](http://www.authenticsociety.com/about.html)

|  |  |
| --- | --- |
|  |  |

Whether you are already familiar with Javascript development or just starting to learn more about dynamic web development, jQuery is an excellent framework to start learning more about today. People who come to jQuery from Javascript background begin to realize how much they have been missing. On this page we will learn how to make jQuery plugins, but several other important points will be made as well.

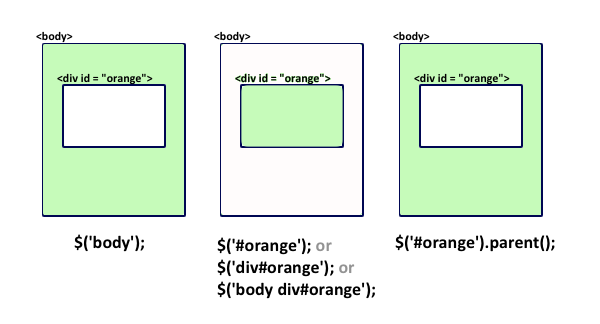
In this tutorial you will learn about 2 different **jQuery plugin** patterns (pattern A and pattern B) -- by the end of this tutorial, you should be able to grasp the basics behind writing custom jQuery plugins. jQuery books as well as a few online tutorials were used as references in writing this tutorial.

***You can't really understand jQuery until you understand CSS Selectors***

In jQuery your CSS selector code begins with the main jQuery object $ (dollar sign). It's just a function name, and it takes a paremeter - your CSS Selector.

Selectors work in the same way as regular CSS selectors you encounter in typical **style.css** files.

Most people only know the ID selector (#) and the class selector (.) but the story goes much...much further. For [jQuery CSS Selector Tutorial extravaganza](http://www.learnjquery.org/newsletter/Tutorial-2-jquery-css-selectors-walkthrough.html" \o "jquery selectors" \t "_blank) (<-**part 1**) please follow this link. But that's not all. I have written yet another [in-depth jQuery CSS Selector tutorial](http://www.learnjquery.org/newsletter/Tutorial-6-more-about-css-selectors.html) (<-**part 2**) just on the subject of selectors!



See the last example? I use the **parent()** function to go back one element in the DOM tree. Cool!

If you understand what's going on in this diagram, you can easily learn the rest of jQuery.

There are plenty of other jQuery tutorials on the Internet.

In addition I'd like to recommend [pre-order my jQuery book for 1/2 the price](http://www.learnjquery.org/) until March 31st, 2012 - hands down if you actually want to **learn jQuery** with tons of practical examples and HTML game, this is the best jQuery book for beginners. If you are ready to drop $25 or so bucks on a jQuery book, then this is it. I promise not to disappoint or your money back.

I own 5 jQuery books, but to be honest they were a little difficult to follow and I wanted to write a book that contains **practical** examples. Well, there you have it. It will be available on March 31st, 2012.

Moving on...I have found out that most jQuery plugin tutorials talk about the basics, but often skip detailed description of functionality.

When I was learning about writing jQuery plugins for the first time, I found many jQuery tutorials not thorough enough - I knew absolutely nothing, and the authors would give me syntax I did not understand - I was just trying to learn the very basics!

An expert jQuery developer may overlook the basics, because they are so obvious to them. But as a beginner, I was totally lost.

I am the writer of several online publications about web development. In my experience, time and time again I see that the advantages of detailed information on how to build your own jQuery plug-in can go a long way.

Once you make the commitment to learn and understand **the basic principles** of writing a jQuery plugin you will never want to go back to regular Javascript programming.

If for any reason at all, a thorough understanding of these principles will help you write a plugin you can't find anywhere else on the Internet. Many of the beginners try to copy and paste the code from cookie-cutter tutorials. Don't choose that path.

Many tutorial authors offer well-written jQuery code that works in all browsers, but copying it into your site will not help you to **understand how the code actually works**.

It has been determined (scientifically proven) that being genuinely curious about learning a subject will make a greater impact on the student's memory, versus when the student has no interest whatsoever in the subject. By the end of this tutorial you will learn the basics of jQuery plugin programming.

Some of the most common questions one may ask when you begin learning about how to write a new jQuery plugin are

* What are CSS Selectors?
* What is the difference between using *$.myfunction* and *$.fn.myfunction*?
* What does the dollar sign mean?
* What does the jQuery function *jQuery.extend* do and how to use it?
* How to initialize my jQuery plugin and pass it function parameters?
* How to provide *default* values for and how to *override* initialization parameters? If you are curious about in depth implementation of jQuery interface for adding your own code into existing framework, this tutorial is for you*!*

In this *jQuery plugin tutorial* I will walk you through the process of writing your own jQuery plugins. This process is the same every time you decide to create a new plugin. Once you understand how your JavaScript code can be integrated into the main jQuery object you will no longer need documentation and will be able to focus on programming plugins.

**Objects in JavaScript**

As you may already know, JavaScript is an object-oriented language. It would be improper to describe jQuery to someone who had not yet studied objects in JavaScript. The basic philosophy of object design in JavaScript is that when you declare (or *define*) a function, that function is by default a class and can be instantiated as an object.

We can use the *new* operator to instantiate copies of the same class. It just so happened that instead of the traditional keyword *class*, JavaScript prefers the keyword *function*. It is interesting to note that a class in JavaScript can be used both as a class from which other classes can be inherited (by implementing what is known as *prototype* inheritance) or the same exacty construct can be used as an actual function that we can call from our code.

In Javascript, everything is an object.

The so-called function closures are often used with [jQuery events](http://www.authenticsociety.com/jquery/list-all-jquery-events.html" \t "_blank). A function closure is the type of a function that doesn't have a reference, or a name. It is simply the object of that function.

**The main jQuery object**

The main jQuery object is simply defined with the keyword *function* as an object whose identifier name is $. For a deeper insight into what this really means, I have written another tutorial [What does the dollar sign mean?](http://www.authenticsociety.com/blog/JavaScript_DollarSign) Be sure to read it if you're still confused about the dollar sign identifier notation.

The jQuery object is equivalent to a global function, such as *window.$*. You see, in JavaScript, when you extend the window object (that must exist in all browsers by design), or in other words, attach new member functions with the dot (.) to the *window* object, it means that you will be able to call that object either by *window.myobj();* or simply *myobj();* You are allowed to do so because every function you attach to the window object can also be called from the global scope in your script. Internally, the jQuery object is created like so:

var jQuery = window.jQuery = window.$ = function(selector, context)

{

// ...

// other internal initialization code goes here

};

This precise declaration allows us to refer to the main jQuery object by the identifier name *jQuery*, or the dollar sign ($). You need to know that *jQuery*, *window.jQuery*, *window.$* or simply *$* can be used interchangeably, because as the declaration stated in the code block above tells us, it refers to the same object in memory.

Please note the *selector* and *context* parameters of the jQuery function object. The *selector* is usually a string that selects a number of elements from the DOM. It can also be the *this* object (a self-reference). The jQuery *selector* parameter accepts the same values you would expect to use in a CSS style definition. For example, consider the following jQuery object call:

// Select all elements of class: "someClass" and

// apply a red border to all of them

jQuery(".someClass").css("border", "1px solid red");

// Select an element with id: someId and insert dynamic html into it

jQuery("#someId").html("<b>So Bold!</b>");

// Exactly the same as above, but using $

$("#someId").html("<b>So Bold!</b>");

This is an example of how powerful a short jQuery statement can be. You dont need to worry about writing your own document.getElementById functions that tend to clutter the code. With just one line of code, jQuery selects *all* elements of the requested type by scanning the entire DOM and applies the desired effect.

Many people use jQuery just for the intuitive element-selector functionality, but if you like this so far, you are going to love the rest of the features at your command when you work with the jQuery framework. It's good to note that jQuery cares about cross-browser compatibility, only a small number of things won't work in all browsers, but it's very rare (I am refering to keyboard events in older versions of IE).

**jQuery Plugin Entry Point**

I would like to begin with a jQuery plugin example code as seen used by most people, followed by an explanation of what it means. But because I am targeting the absolute jQuery beginners who may not have enough experience with JavaScript, I'd like to clarify a few things first.

When learning a new language or a framework such as jQuery, you need to understand where the entry point of your plugin program is. Traditionally, for years prior to jQuery, some JavaScript programmers liked to execute their crucial code in the *window.onload* function as illustrated below:

// Override the onload event

window.onload = function()

{

// the page finished loading, do something here...

}

This code actually overrides the onload event of the HTML <body> tag. All this means to us is that our code will be executed soon as the page is *finished* loading. It makes sense because sometimes pages take time to load, or the downloading process is segmented by the browser architecture.

We would not want to compile and execute any JavaScript code on a page that is currently being loaded. The jQuery internal architecture also utilizes the *window.onload* event, but before it does so, it checks whether the entire DOM (document object model) has been loaded because it is very important.

It is not enough for jQuery to know that the page has been loaded, we must ensure that the DOM has been fully constructed. This is achieved by listening to the DOMContentLoaded in most browsers. Luckily, we don't need to worry about this at all because jQuery takes care of this for us internally.

To provide us with this functionality, jQuery gives us a new method called *ready* that we can call on the main jQuery object itself. When writing a jQuery plugin, we use the ready function to check whether we are 'ready' to execute our plugin code. Please note that this is not yet the plugin code, this will be *the entry point* of our plugin. You can think of this as a jQuery's version of window.onload function:

<script type = "text/javascript">

// Define the entry point

$(document).ready(function()

{

// The DOM (document object model) is constructed

// We will initialize and run our plugin here

});

</script>

Let's think for a moment that we are writing a plugin called *Shuffle*. Assuming the plugin provides two separate functions for initialization as well as execution of the base code, the code may have looked something like the following:

<script type = "text/javascript">

// One way to initialize plugin code

$(document).ready(function()

{

jQuery.Shuffle.initialize( "monalisa.jpg", 5, 8, 67, 1500);

jQuery.Shuffle.run();

});

</script>

More than often, the code above can be improvised. Is it required to use this specific format for initializing and executing our plugin? No. For me personally, because I come from C and C++ background writing computer games, I like to separate initialization and execution function calls like I show above.

**How to make a jQuery Plugin - Internal definition of a jQuery plugin**

The coding style is entirely up to you but also depends on what you are trying to accomplish. It is easy to think that a plugin must be written, initialized and executed in a certain way all the time but this is simply not true. The reason you see many different styles and syntactical differences in jQuery plugin code is that the programmers are trying to accomplish different things. Additional knowledge of JavaScript Inheritance and function closures may help here quite a bit.

First let's talk a little about the following syntax seen in many a plugin written by jQuery programmers. It is very easy to get confused about it especially for programmers with only intermediate knowledge of JavaScript. So, this may seem quite ambiguous at first.

<script type = "text/javascript">

(function($){ ... })(jQuery);

</script>

What is going on here? In a real-world scenario, the three dots would be replaced with actual code that we would like to be executed. Here we see a definition of an *anonymous* function (also known as*function closure*), that takes a parameter called dollar sign ($). The function is wrapped in parenthesis, but why? You see, the function in this example is an anonymous function that has no reference to itself, but by wrapping it in parenthesis, the JavaScript syntax allows us to reference to an anonymous function we just created.

We could have added a dot (.) after the function definition wrapped in parenthesis and called a member function that this function object supports. We could then add the dot at the end of that statements as well, and call yet another member function that can be executed on the type of the object returned by the previous function. This "function chaining" feature of JavaScript language is common throughout many other script languages, for example Perl. It is a common script-language feature because it makes your code shorter, more readable and intuitive.

So, by wrapping an anonymous function with parenthesis we can reference to that function's memory location without actually refering to the name of that function - and well, we can't do that because the function has no name!

Furthermore, not only can we reference to a function that has no name, we can also call that function in the same statement that created it. And that's exactly what's going on here. The nameless function is defined, wrapped in parenthesis, and then that function is immediately called. This is only one example of the *function closure* usage that you will see throughout jQuery and other advanced JavaScript code.

Why would someone do such a thing? There are several reasons. This is done to minimize code length and more importantly, to hide parameters that start with the dollar sign character from the global scope of the script. The real reason for doing *that* is for cases where you are using multiple frameworks that may use the dollar sign function object in the global scope of the program (*which is a very bad design idea in itself*).

But in a real-world scenario... when would we see that happen? That depends on circumstances and developer's choices.

Since in this tutorial we are not using any additional frameworks or outside code, this obscure and sometimes confusing syntax is not necessary and the chances of creating a conflict are zero. Let's take a look at a very basic plugin construction idea:

<script type = "text/javascript">

jQuery.func = doSomething;

jQuery.fn.func = doSomethingElse;

</script>

Where *doSomething* and *doSomethingElse* are simply previously defined function objects. However, this technique is not always desired because we are losing the ability to chain our jQuery plugin with other jquery API functions that already exist, or that are created by other developers. In the end, what it comes down to is that if you want your plugin have that *chaining* functionality, you must use the self-calling function mechanism as explained above. If chaining the functionality of your plugin with other people's code is not important to you, simply add your own objects and variables to the *$.*or *$.fn* objects. Also keep in mind that as discussed earlier, jQuery, the dollar sign (and even something like window.$) can be used interchangeably and refer to the same exact object in memory. jQuery documentation tells us to use jQuery object instead of the dollar sign. The dollar sign must be hidden, it must only be used internally in the jQuery architecture. Do not expose the dollar sign to the implementators of your plugin.

**jQuery Plugin Design Pattern `A`**

Now, let's consider the internal code of an arbitrary plugin that uses the self-referencing anonymous function call that takes the jQuery object as the only parameter. I wanted to include additional comments to explain what each part means. The comments obviously clutter the code, but I think it is slightly more intuitive than explaining each line by writing about it and I tried to format the comments in a conscious way.

It is easy to get confused about what the *this* keyword is refered to in this pattern in different places in the code. I created some local variables that I named *vari*, stored within the main jQuery object ($.vari), and a jquery fn object ($.fn.vari). These could have been function objects too, but for simplicity's sake I use basic variables. You can see these values displayed with alert functions in the following code. I did this to demonstrate which varibles the *this* keyword will refer to in different function scopes. (For those who may not know, in classic computer programming a scope can be defined by the opening and closing brackets {, and } which modifies the visibility behavior of variables. I don't want to go any further into explaining how JavaScript scope works because it is like a science of its own, but I hope this clears some fog for people who may not know how the scope works. If you do, that's great!

Let's take our first look at the classic jQuery plugin pattern. First, I want to show the pattern without the comment clutter.

(function($)

{

$.vari = "$.vari";

$.fn.vari = "$.fn.vari";

// $.fn is the object we add our custom functions to

$.fn.DoSomethingLocal = function()

{

return this.each(function()

{

alert(this.vari); // would output `undefined`

alert($(this).vari); // would output `$.fn.vari`

});

};

})(jQuery);

// $ is the main jQuery object, we can attach a global function to it

$.DoSomethingGlobal = function()

{

alert("Do Something Globally, where `this.vari` = " + this.vari);

};

$(document).ready(function()

{

$("div").DoSomethingLocal();

$.DoSomethingGlobal();

});

And here are some details:

// plugin-name.js - define your plugin implementation pattern

(function($) // The $ here signifies a parameter name

// As you can see from below, (jQuery) is

// immediately passed as the $ param

{

$.vari = "$.vari";

$.fn.vari = "$.fn.vari";

// 1.) Add a custom interface `DoSomethingLocal`

// Which will modify all selected elements!

// If you are a software engineer, think about this as

// a member function of the main jQuery class

$.fn.DoSomethingLocal = function()

{

// return the object back to the chained call flow

return this.each(function() // This is the main processor

// function that executes on

// each selected element

// (e.g: jQuery("div"))

{

// this ~ refers to a DOM element

// $(this) ~ refers to a jQuery object

// Here, the `this` keyword is a self-refence to the

// selected object `this.vari` is `undefined` because

// it refers to selected DOM elements. So, we can do

// something like: var borderStyle = this.style.border;

// While $(this).vari, or jQuery(this).vari refers

// to `$.fn.vari`

// You would use the $(this) object to perform

// any desired modification to the selected elements

// $(this) is simply a reference to the jQuery object

// of the selected elements

alert(this.vari); // would output `undefined`

alert($(this).vari); // would output `$.fn.vari`

});

};

})(jQuery); // pass the jQuery object to this function

// 2.) Or we can add a custom interface to the global jQuery

// object. In this case, it makes no sense to enumerate

// through objects with `each` keyword because this function

// will theoretically work in the `global` scope. If you are

// a professional software engineer, think about this

// as a [static function]

$.DoSomethingGlobal = function()

{

// this will output this.vari = $.vari

alert("Do Something Globally, where `this.vari` = " + this.vari);

};

// index.html - test the plugin

$(document).ready(function()

{

$("div").DoSomethingLocal();

$.DoSomethingGlobal();

});

There are two different types of interfaces you can add to the main jQuery object that is already defined by the framework. Please take a look at the example above where we have just added the function DoSomethingLocal and DoSomethingGlobal.

1.) DoSomethingLocal is defined as *$.fn.DoSomethingLocal*. You can add as many custom functions to the *jQuery.fn* (or simply *$.fn*) object as required by your plugin implementation. Any function you add to the $.fn object is assumed to work on references to an instance of an element/object that was selected. This is what the $.fn syntax means - we are simply adding a custom interface to a jQuery object that assumes selected elements.

2.) DoSomethingGlobal is applied directly to the global jQuery object as *$.DoSomethingGlobal*. A function attached to the jQuery framework in such way assumes that it will not work on selected elements but rather it will work in the global scope and contain practically any implementation.

**jQuery Plugin Design Pattern `B`**

Many people prefer wrapping the plugin code in an anonymous function, instantly call it by adding the parenthesis at the end of the statement and pass jQuery object to it like shown in the jQuery Plugin Pattern A above, but do we have to do that? We don't. Let's take a look at the alernative way of building jQuery plugins, which some people will find more straightforward and easy to read.

If you don't yet feel comfortable with the classic pattern presented above (it's a monster), you might as well play around with the following pattern until you become more familiar with jQuery. Please note that the following pattern must only be used in cases where you are not implementing *chainability*, in other words, you must assume that your plugin functions will not be returning a jQuery object. You must also point this out in your documentation.

// Plugin base object

$.gShuffle = function()

{

}

// Initializes plugin

$.gShuffle.initialize = function()

{

}

// Runs the plugin

$.gShuffle.run = function()

{

};

That's pretty much it. This pattern will be popular among programmers coming to Javascript from languages like C++, or possibly even PHP?