

Working with jQuery, Part 3: Intermediate JQuery: Creating your own plug-in

Skill Level: Intermediate

[Michael Abernethy \(mike@abernethysoft.com\)](mailto:mike@abernethysoft.com)
Product Development Manager
Optimal Auctions

26 May 2009

jQuery lets you create your own plug-ins to extend the functions of jQuery—and to give back to the jQuery community. This article steps you through the process for creating your own plug-in and as well as getting it listed on the jQuery plug-in community Web pages.

Introduction

In a previous article in this series, [Working with jQuery, Part 2: Intermediate JQuery: The UI project](#), I discussed using plug-ins in your own jQuery code to increase the effectiveness of your Web applications. However, these plug-ins don't write themselves. They are written and tested by developers like you and me, donating their free time to the jQuery community to make it better. And we do this all for free, living only for the love of our own code. This article focuses on how you can give something back to this great community, by writing your own plug-in and checking it into the plug-in pages hosted by jQuery. This lets everyone use the plug-in you create, bettering the entire jQuery development community. Consider this my charity work for the year.

While writing the plug-in for this article, I found that the plug-in creation, and the framework for creating it, is extremely straightforward and easy. The hard work is actually thinking of something that hasn't already been done by someone else, and writing the "grunt work" JavaScript code that actually does things. Because of the straightforward plug-in structure, which is easy for beginners and flexible for advanced coders, the number of plug-ins has grown rapidly.

Of course, in doing the research for this article, I also found that every author has their own style for writing a plug-in and that jQuery allows several different styles for writing them. I will concentrate on the easiest one here, and the one recommended by jQuery itself, but will point out any differences or options as they pop up.

Describing your plug-in

Step 1 in creating a plug-in is, of course, thinking of a good idea. Chances are, like most good ideas, someone else has already beaten you to it. For the plug-in I'll be developing here, it's not a novel concept, but at the time of this article I couldn't find it anywhere in the jQuery plug-in community. I know I would personally get a lot of use out of it.

My plug-in is a NumberFormatter plug-in. Number formatting is likely familiar to anyone who's worked with server-side code like Java™ or PHP and internationalization. It's common knowledge that not everyone in the world formats their numbers in the same way. For example, not everyone uses miles for distance. A number that we would write in the U.S. as "1,250,500.75" (I just grabbed this number from my tax form), would be written differently in different countries: "1.250.500,75" in Germany, "1 250 500,75" in France, "1'250'500.75" in Switzerland, and "125,0500.75" in Japan. The number is exactly the same, but it's just written using a different format when presented to users of the Web application.

So, the question emerges, when you're writing an international application, how would you present these numbers to people in different countries? The solution, of course, is to use server-side formatting, which is pretty common. Java has a robust formatting library to make formatting numbers easy. When the page is set up on the server with numbers, the server can take care of formatting these numbers. However, many times you might not be on the server, so you need a way to format numbers on the client, without ever talking to the server.

Here's a typical use case for what I'm describing. You have an input field in your Web application that asks a person for their salary. In the U.S., the user can type in varied forms of input - "\$65000", "65,000", "65000", "65,000.00". All these numbers are exactly the same, but you want to control how these numbers look on the screen, in order to create a better user experience. You *could* call the server after the number is entered, but this could get annoying if you have a lot of number fields with different formats. Plus, if you can take care of this problem on the client, and offer instantaneous feedback to the user, why wouldn't you do that?

So, I've established the gap in JavaScript/jQuery functionality I am trying to fill. My plug-in is going to offer number formatting on the client, giving others a way to offer internationalization for their Web applications without having to talk to the server. As an added bonus, my plug-in will also do the opposite function; the plug-in will allow developers to parse numbers, getting a number out of a formatted text string. This

can be used for mathematical operations on the client. Additionally, I will mimic the functionality in the Java `DecimalFormatter` class, to maintain a commonality between the client code and the standard server-side method of doing number formatting.

Step 1 result: I've established a need for a plug-in, and have defined the specs that I will use in filling the need.

Plug-in rules

The jQuery team has established a number of general rules that they want plug-in authors to follow to create a common and expected environment for the plug-in users. Because I consider the jQuery team to be much smarter than me, who am I to disagree with any of the rules, right? For that reason, I'll outline the rules here, and will also strive to follow them in every step of my own plug-in.

- Name your file `"jquery.<your plug-in name>.js"`
Well, this makes sense, because you want someone to look at the file and know right away that it's a jQuery plug-in, and which plug-in it is.
Check. I will call my plug-in `"jquery.numberformatter.js."`
- All new methods are attached to the `jQuery.fn` object, all new functions to the `jQuery` object
This may be confusing at this stage, and I will discuss this more in the next section because this is the most important rule for actual coding.
Check. I will only attach methods/functions to those two objects.
- `"this"` is a reference to the jQuery object
This is to facilitate plug-in writing by letting all plug-in authors know what object they will receive from jQuery when they reference `"this."`
Check. I will use `"this"` only as a reference to the jQuery object.
- All methods/functions defined in the plug-in must have a `;"` (semicolon) at the end or code minimizers will break.
Because it is a best practice to minimize JavaScript files, breaking the minimizer would be bad, and chances are your plug-in would get dumped quickly
Check. All of the methods/functions will have a `;"` at the end.
- All methods must return the jQuery object, unless otherwise noted
The daisy-chaining of jQuery methods is quite popular, and if you write a plug-in that breaks this chain, it literally "breaks the chain"

Check. My `format()` method will return the jQuery object, and even though the `parse()` method will not return the jQuery object, I have documented it in many places that this function breaks the chain. (After all, it would be difficult/impossible to return a Number object and not break the chain.)

- You should always use `this.each()` to iterate over the matched elements, as this is a reliable and efficient way to loop through objects. For performance and stability reasons, they recommend all methods use this means to loop through matched elements. Check. My methods will only loop through matched elements in this manner.
- Always use "jQuery" instead of "\$" in your plug-in code. This is important because it allows users who have a conflict with "\$" (those who may be using another JavaScript library) the ability to change their pseudonym for jQuery in one place using the "`var JQ = jQuery.noConflict();`" function. However, as I have looked through many plug-ins, I have found that this rule is broken quite often, which is unfortunate. If the developer ever needs to change the jQuery pseudonym, it likely means that the broken plug-ins will get dumped. Check. In my plug-in, I will only use the jQuery and not its "\$" pseudonym.

So, those are the only rules/recommendations that are placed on your plug-in code. The truth of the matter is, they are de facto enforced, because if you start breaking the rules of the plug-ins, your plug-in won't get used very often, and it will get bad reviews from your peers. What will result is a plug-in that gets rarely used, and one you probably wasted your time on. Therefore, it's important to follow the rules. This not only helps your peers out, and makes their coding consistent, it also gives your plug-in a greater chance of success.

Step 2 result: I will follow all the rules of creating a jQuery plug-in

Writing the plug-in

Now it's time to start writing the code! The first step in starting your plug-in is to decide how you want to structure your plug-in. There are two choices to start off with: do you want it to be a method or a function? "What's the difference?" you might ask.

A method, as I said above, needs to be attached to the `jQuery.fn` object, and a function needs to be attached to the jQuery object. Oh, that clears *everything* up, right? Well, probably not if you're relatively new to jQuery. Instead, think of it this

way. A method gives your code the ability to loop through all the selected elements that are passed into the plug-in. As a result, your plug-in can accept any type of HTML element, and it's up to your plug-in to define how to work with each element. Thus, your plug-in method can accept any jQuery selector, anything from "p" to "#mySpecificPageElement". This would be desirable if you want to have some flexibility in your plug-in, allowing users to pass in any type of page element. The burden is on you, the plug-in developer, to deal with everything properly. Contrast that with a function, which *doesn't* take any selected elements as arguments. It's simply a function that will be applied to the entire page. The burden shifts to the plug-in developer, who has to define which page elements they want the plug-in to interact with, and ignore the others. Let's take a look at the difference in some code.

Listing 1. jQuery plug-in method/function

```
// This is a method because you can pass any type of selector to the method and it
// will take some action on the results of the selector. In this case, it will
// doSomething() on the page element with an ID of myExample
$("#myExample").doSomething();

// This is also a method, even though you are passing the entire page body to
// the method, because you are still passing a selector
$("body").doSomethingElse();

// This is a function, because you are NOT passing any selector to the function
// The plug-in developer must determine what page elements they want to take action on.
// This is usually accomplished by the plug-in developer requiring the page elements
// to contain a certain class name.

<div class="anotherThing">

// This hypothetical plug-in developer would document that his plug-in only works
// on elements with the class "anotherThing"
$.anotherThing();
```

So, judging from these descriptions, it looks like your plug-in should use a method, because you want to allow its users to tell you what page elements they want formatted. Listing 2 shows how your plug-in code looks now.

Listing 2. Your method definition

```
jQuery.fn.format = function();

// You would call your plug-in like this (at this point)
$("#myText").format();
```

Of course, your function can't simply be a one-size-fits-all plug-in, because you're dealing with internationalization and you can't automatically figure out what country you want the text formatted for or the format you want. So, you must modify your plug-in slightly to accept some options. You will need two options in your formatting method: the format the number should use (for example, #,### vs. #,###.00) and the locale (the locale being a simple 2-letter country code to determine which international number formatting to use).

You also want to make your plug-in as easy to use as possible, because you want to increase your plug-in's chances of success. This means you should also go ahead and define some default options so the user doesn't have to pass in the options if they don't want to. Because I'm writing this from the U.S., and that also happens to be the most common number formatting in the world, I'll default to the "us" locale, and I'll default to the "#,###.00" format, for no other reason other than it can be used for currency with this default.

Listing 3. Allowing options in your plug-in

```
jQuery.fn.format = function(options) {  
  
    // the jQuery.extend function takes an unlimited number of arguments, and each  
    // successive argument can overwrite the values of the previous ones.  
    // This setup is beneficial for defining default values, because you define  
    // them first, and then use the options passed into the method as the  
    // second argument. This allows the user to override any default values with their  
    // own in an easy-to-use setup.  
    var options = jQuery.extend( {  
  
        format: "#,###.00",  
        locale: "us"  
  
    }, options);
```

The final step to creating your plug-in framework is to work properly with the selected elements that were passed into the method. If you'll recall from the examples above, the selected elements could be 1 page element, or it could be many page elements. You have to deal with them all equally well. Also, recall from the jQuery plug-in commandments, the "this" object is a reference to the jQuery object. So, you have a reference to the jQuery-selected elements that were passed into the method, and now you just need to iterate through them. Also, looking back at the Commandments, each plug-in method should return the jQuery object. You know, of course, the jQuery object is "this", so you can simply return this in your method and be all good. Let's take a look at how you can accomplish both in a code snippet, iterating through each selected element and return the jQuery object.

Listing 4. Working with the jQuery object

```
jQuery.fn.format = function(options) {  
  
    var options = jQuery.extend( {  
  
        format: "#,###.00",  
        locale: "us"  
  
    }, options);  
  
    // this code snippet will loop through the selected elements and return the jQuery object  
    // when complete  
    return this.each(function(){  
        // inside each iteration, you can reference the current element by using the standard  
        // jQuery(this) notation  
        // the rest of the plug-in code goes here
```

```
});
```

Because the actual plug-in code itself isn't the point of this article, I won't spend any time on it, but you can see it all in the plug-in code attached to this article (see [Download](#)). I'd also like to show you an example of how you'd set up your plug-in architecture if you decided to write a function, as opposed to a method.

Listing 5. Example plug-in using a function

```
jQuery.exampleFunction = function(options) {  
    var options = jQuery.extend( {  
        // your defaults  
    }, options);  
    jQuery(".exampleSelector").each(function(){  
    });  
});
```

Fine tuning the plug-in

Most beginner plug-in articles you find around the Web will stop there, letting you take the basic plug-in format and run with it. However, this basic framework is just that, basic. There are other important things that you must consider when writing your own plug-in, things that will give your plug-in the polish it needs to be more than a beginner plug-in. With these two added steps, you can turn your beginner plug-in into an intermediate-level plug-in.

Fine tune #1 - Make internal methods private

In any object-oriented programming language, you will find it handy to create external functions that can run repetitious code. In the NumberFormatter plug-in that I've created, there is an example of this code — the code that decides which locale got passed into the function and which characters to use as the decimal points and the group separator. This code is needed in the both the `format()` method and the `parse()` method, and any beginning programmer can tell you this belongs in its own method. However, one problem arises from this decision because you are working with a jQuery plug-in: if you simply make it its own function, defined in JavaScript, then the method can be called by anyone who uses the script, for any purpose. This isn't what the function is intended for, and I would like to prevent outsiders from calling it, because it is intended only for my internal work. So, let's see how you can make this function a private one.

The solution for this private method problem is something called a *Closure*, and it

essentially closes the entire plug-in code from outside calls, except those you attach to the jQuery object (which is public). Using this design, you can put any code you want within your plug-in and not worry about it being called by outside scripts. By attaching your plug-in methods to the jQuery object, you are essentially making them public methods, with all the rest of the functions/classes becoming private. Listing 6 gives you a look at the code required to do this.

Listing 6. Make a function private

```
// this code creates the Closure structure
(function(jQuery) {

    // this function is "private"
    function formatCodes(locale) {
        // plug-in specific code here
    }; // don't forget the semi-colon

    // this method is "public" because it's attached to the jQuery object
    jQuery.fn.format = function(options) {

        var options = jQuery.extend( {

            format: "#,###.00",
            locale: "us"

        },options);

        return this.each(function(){
            var text = new String(jQuery(this).text());
            if (jQuery(this).is(":input"))
                text = new String(jQuery(this).val());

            // you can call the private function like any other function
            var formatData = formatCodes(options.locale.toLowerCase());

            // plug-in-specific code here
        });
    }; // don't forget the semi-colon to close the method

// this code ends the Closure structure
})(jQuery);
```

Fine tune #2 - Make your plug-in's defaults overridable

The last step to fine-tuning this plug-in is to give it the ability to have its defaults overridden. After all, if a developer in Germany downloaded this plug-in, and knew that all of his Web application's users would want to see the German locale, you should give him the ability to change the default locale in one line of code, rather than requiring him to write it in every method call. This is particularly handy in your plug-in because it is highly unlikely that a Web application will present numbers in different international formats to its users. Chances are, if you're on a Web page, all the numbers are going to be formatted using the same locale.

This step will require you to modify your code somewhat, so what you've seen so far has been updated to reflect this latest polishing step.

Listing 7. Overridable defaults

```

jQuery.fn.format = function(options) {
  // Change how you load your options in to take advantage of your overridable defaults
  // You change how your extend() function works, because the defaults
  // are globally defined, rather than within the method. If you didn't use the
  // {} as the first argument, you'd copy the options passed in over the defaults, which is
  // undesirable. This {} creates a new temporary object to store the options
  // You can simply call the defaults as an object within your plug-in
  var options = jQuery.extend({}, jQuery.fn.format.defaults, options);

  return this.each(function(){

    // rest of the plug-in code here

  // define the defaults here as an object in the plug-in
  jQuery.fn.format.defaults = {
    format: "#,###.00",
    locale: "us"
  }; // don't forget the semi-colon

```

That is the last step to creating your plug-in! What you should have now at this point is a polished plug-in that's ready for the final steps of testing. Listing 8 shows the completed plug-in that you've put together in this article so you can see how all the pieces fit together. Also, included is the `parse()` function, which I haven't talked about up until this point, but which is included in the plug-in. (I've excluded the grunt work of the plug-in, the parts that deal with formatting, because they are outside the scope of the article. They are included in the examples, and are of course in the plug-in itself).

Listing 8. The NumberFormatter plug-in

```

(function(jQuery) {

  function FormatData(valid, dec, group, neg) {
    this.valid = valid;
    this.dec = dec;
    this.group = group;
    this.neg = neg;
  };

  function formatCodes(locale) {
    // format logic goes here
    return new FormatData(valid, dec, group, neg);
  };

  jQuery.fn.parse = function(options) {

    var options = jQuery.extend({}, jQuery.fn.parse.defaults, options);

    var formatData = formatCodes(options.locale.toLowerCase());

    var valid = formatData.valid;
    var dec = formatData.dec;
    var group = formatData.group;
    var neg = formatData.neg;

    var array = [];
    this.each(function(){

```

```
    var text = new String(jQuery(this).text());
    if (jQuery(this).is(":input"))
        text = new String(jQuery(this).val());

    // now we need to convert it into a number
    var number = new Number(text.replace(group, '').replace(dec, ".").replace(neg, "-"));
    array.push(number);
  });

  return array;
};

jQuery.fn.format = function(options) {

  var options = jQuery.extend({}, jQuery.fn.format.defaults, options);

  var formatData = formatCodes(options.locale.toLowerCase());

  var valid = formatData.valid;
  var dec = formatData.dec;
  var group = formatData.group;
  var neg = formatData.neg;

  return this.each(function(){
    var text = new String(jQuery(this).text());
    if (jQuery(this).is(":input"))
      text = new String(jQuery(this).val());

    // formatting logic goes here

    if (jQuery(this).is(":input"))
      jQuery(this).val(returnString);
    else
      jQuery(this).text(returnString);
  });
};

jQuery.fn.parse.defaults = {
  locale: "us"
};

jQuery.fn.format.defaults = {
  format: "#,###.00",
  locale: "us"
};

})(jQuery);
```

Testing the plug-in

The final step in creating my plug-in is to test it—thoroughly. Nothing angers a plug-in user more than seeing a bug in your plug-in. Rather than fix it, they will quickly dump your plug-in and move on. Getting a few of these types of users as well as some bad reviews can quickly sink your plug-in. Besides, it's a good reciprocal behavior—you want the plug-ins you use in your code to be well tested, so you should return the favor in your own plug-in code.

I created a quick test structure to test my plug-in (no need for a unit testing library), which creates dozens of spans with various numbers and the correct format right

after the number. The JavaScript test calls `format` on the numbers and then compares the formatted number with the desired result, showing them as red if it fails. With this quick test, I can set up dozens of different test cases, testing every possible format, which I've done. I've attached my test page to the examples [download](#) so you can see one possible solution to testing your plug-in, utilizing jQuery to do the testing for you.

Looking at the finished plug-in

Let's take a look at the new `NumberFormatter` in action. I've created a simple Web application that you can look at to see how the `NumberFormatter` plug-in can fit in your applications.

Figure 1. `NumberFormatter` in action

Example 1 - My Tax Plan

How much money do you make?	<input type="text" value="65.200,00"/>
How much is your house worth?	<input type="text" value="212.500"/>
How many kids do you have under 18?	<input type="text" value="3"/>

Your tax would be \$8.719

The Web application is simple and straightforward enough. When the user leaves the textfield, after typing in their salary, house, and child information, the `NumberFormatter` plug-in will format their answers appropriately. The plug-in allows the Web app to consistently format numbers to the user. Also, note that this Web application is formatted for a user in Germany, as the decimal and group characters are different than it would be for a user in the U.S. (this lets me show how to change defaults).

Listing 9. `NumberFormatter` in action

```
$(document).ready(function() {  
    // use the AlphaNumeric plug-in to limit the input  
    $(".decimal").decimal();  
    $(".numeric").numeric();  
});
```

```

// you want to change the defaults to use the German locale
// this will change the default for every method call on the
// entire page, so you won't have to pass in the "locale"
// argument to any function
$.fn.format.defaults.locale = "de";
$.fn.parse.defaults.locale = "de";

// when the salary field loses focus, format it properly
$("#salary").blur(function(){
    $(this).format({format:"#,###.00"});
});

// when the house field loses focus, format it properly
$("#houseWorth").blur(function(){
    $(this).format({format:"#,###"});
});

// when the kids field loses focus, format it properly
$("#kids").blur(function(){
    $(this).format({format:"#"});
});

// calculate the tax
$("#calculate").click(function(){
    // parse all the numbers from the fields
    var salary = $("#salary").parse();
    var house = $("#houseWorth").parse();
    var kids = $("#kids").parse();
    // make some imaginary tax formula
    var tax = Math.max(0,(0.22*salary) + (0.03*house) - (4000*kids));
    // place the result in the tax field, and then format the resulting number
    // you need one intermediate step though, and that's the "formatNumber" function
    // because all numbers in JavaScript use a US locale when made into a String
    // you need to convert this Number into a German locale String before
    // calling format on it.
    // So, the steps are:
    // 1) the tax is a Number that looks like 9200.54 (US locale)
    // 2) formatNumber converts this to a String of 9200,54 (German locale)
    // 3) put this String in the #tax field
    // 4) Call format() on this field
    $("#tax").text($.formatNumber(tax)).format({format:"#,###"});
});
});

```

Some remaining things to point out about the NumberFormatter plug-in before I close things up. First of all, this is the first 1.0.0 release of the plug-in, so I'm hoping to expand it in the future to include more formatting features that are included in the Java DecimalFormat. This includes support for currency, scientific notation, and percentages. It will also include separate formatting rules for positive and negative numbers, beyond the simple "-" for a negative (for example, using (5,000) for negative numbers, which is done in accounting). Finally, a good formatter will allow any character in the format, it just ignores any that aren't part of the reserved characters. These are all features I hope to add in the near term to make this a robust plug-in.

Getting the locale of a user

One last question that's not specific to jQuery plug-ins, but one that might arise from

using this plug-in—how do you get the user's locale? This is a good question, because there's currently no way to get this information using JavaScript. You'll need to create a JavaScript Bridge to do it. What's a JavaScript Bridge? I just mean you can set up a simple design pattern to get values into the JavaScript code from the server-side code. Listing 10 shows you could use Java; how you could do this on a JSP page.

Listing 10. Getting a user's locale

```
<%
// the request object is built into JSPs
// unfortunately, it's not any easier
// tested on FF, IE, Safari, Chrome
String locale = "us"; // or your default locale
String accLang = request.getHeader("Accept-Language");
if (accLang.length() > 5)
{
    accLang = accLang.substring(0,5);
    locale = accLang.substring(accLang.indexOf("-")+1);
}
%>

$(document).ready(function() {
// take advantage of the ability to override defaults by using the JavaScript
// Bridge here. Then your page can use the format() and parse() functions
// elsewhere in the page without modifying them for a user's locale.
$.fn.format.defaults.locale = "<%=locale%>";
$.fn.parse.defaults.locale = "<%=locale%>";
});
```

Sharing the plug-in

Finally, you are done with writing and testing the plug-in. The last step is to share it with the world and get it on the jQuery Web site's plug-in repository.

- Go the plug-in page on the jQuery Web site and in the left navigation, click **Login/Register** and then **Create New Account**. Log in if you already have an account or create a new account.
- After authenticating, you will be presented with options in the left navigation. Among these is "Add plug-in."
- Navigate through the plug-in creation page. Because you only tested this plug-in in jQuery 1.2, you will only include that as a compatible version. Take some time to write a good title for the plug-in as well as a good description. After all, this your time to sell your plug-in to other users. There are hundreds of plug-ins, and you need a way to make yours stand out from the crowd. Don't be shy about all the great things it can do.

- The plug-in will ask you to supply home pages for the plug-in. If you just created the plug-in, chances are you don't have a home page for it. Luckily, Google happily provides hosting to open source projects, if you don't have your own server set-up to host the plug-in. I chose to put this plug-in in Google Code. To set up your own Google Code project, simply visit code.google.com and follow the registration process.
- Press **Submit** and your plug-in is created! Congratulations, your plug-in is now a part of the jQuery plug-in community, and you are officially a contributor to an open source project. Reward yourself by giving your plug-in a 5-star rating, because you earned it!

Conclusion

Throughout this article, I've focused on how to create a plug-in for the jQuery JavaScript framework. I started at the very beginning, coming up with the original idea and putting together some of the specs that would be used in the plug-in. Then I read the jQuery plug-in commandments, rules that are put in place to ensure that plug-ins are consistent across authors. I also reminded you throughout the article about these rules, because I've seen many of these rules get broken in plug-ins, especially the rule that states you should only use "jQuery" instead of "\$" in your plug-in. (Of course, I followed this rule).

After introducing you to the background of the plug-in, and the rules of writing a plug-in, you learned the basic plug-in framework and the difference between writing a method and writing a function in your plug-in. A method should be used whenever you want to take the selected elements as the arguments and take action upon them. The burden of supplying the page elements rests with the person calling the method. On the other hand, a function should be used when you aren't interested in selected elements because you already know the elements on the page on which you will take action. The burden of supplying the page elements rests with the plug-in developer writing the function. Both forms of the plug-in are valid, and just differ depending on the plug-in's need. Finally, you looked at the basic way to set up default options and how to allow users to supply their own options.

The next step of the article added some polish to the plug-in to make it more advanced. This step added a Closure around the entire plug-in, effectively allowing you to create private functions and public functions. I made the functions I call internally in the plug-in private, so that no one outside of the plug-in can call them. You also saw how to expose the defaults to plug-in users, letting the users define their own defaults in one spot, easing coding for them.

Finally, you used the plug-in in an example Web application to show you how it behaves. The last part of the article was what you put all this hard work in for—you

got to check the plug-in into the jQuery plug-in Community site, and be a part of the JavaScript library many have grown to depend on.

Downloads

Description	Name	Size	Download method
Zip file with Example	examples.zip	24KB	HTTP

[Information about download methods](#)

Resources

Learn

- Read the complete [jQuery API page](#) to see all of the available functions in the library.
- The [NumberFormatter Google Code Home Page](#) lets you keep up to date with this project and download it for yourself.
- The [NumberFormatter jQuery page](#) exists in the plug-in community, for everyone to use.
- Visit the [jQuery plug-in](#) page to see all the possible plug-ins you can use in your projects.
- Read [jQuery's Official Plug-in Guide](#), which offers some advanced tips (but skimps on the beginning steps) to writing a plug-in.
- Get a thorough and complete background on CSS, JavaScript, and any other Web language at [W3Schools](#).
- Read the first three articles in my jQuery series, that deal with an introduction to the library. "[Working with jQuery, Part 1](#)," "[Working with jQuery, Part 2](#)," "[Working with jQuery, Part 3](#)."
- Host your own code on the [Google Code Project](#).

Get products and technologies

- Download the [1.3.2 Minimized jQuery](#), which was the latest stable version at the time of this article, and drop it in your own code.

About the author

Michael Abernethy

In his 10 years in the technology field, Michael Abernethy has worked with a wide variety of technologies and clients. He currently works as the product development manager for Optimal Auctions, an auction software company. His focus is on Rich Internet Applications and making them both more complex and simpler at the same time. When he's not working at his computer, he can be found on the beach in Mexico with a good book.