10 The Plug-in API

Whenever a task is to be performed two or more times, it is a good idea to apply the **DRY** principle—**Don't Repeat Yourself**. To facilitate this, jQuery provides several tools for developers that go beyond simple iteration and function creation.

One of these powerful tools is jQuery's **plug-in** architecture, which makes creating and reusing extensions to the jQuery library a simple task. In this chapter, we'll take a brief look at using the existing third-party plug-ins and then delve into the various ways of extending jQuery with plug-ins that we define ourselves.

Using a plug-in

Taking advantage of an existing jQuery plug-in is very straightforward. A plug-in is contained in a standard JavaScript file. There are many ways to obtain the file, but the most straightforward way is to browse the jQuery plug-in repository at `http://plugins.jquery.com/`. The latest releases of many popular plug-ins are available for download from this site.

To make the methods of a plug-in available to us, we just include it in the `<head>` of the document. We must ensure that it appears *after* the main jQuery source file, and *before* our custom JavaScript code.

```
<head>
  <meta http-equiv="Content-Type"
    content="text/html; charset=utf-8"/>
  <script src="jquery.js" type="text/javascript"></script>
  <script src="jquery.plug-in.js"
    type="text/javascript"></script>
  <script src="custom.js" type="text/javascript"></script>
  <title>Example</title>
</head>
```

After that, we're ready to use any of the methods made public by the plug-in. For example, using the Form plug-in, we can add a single line inside our custom file's `$(document).ready()` method to make a form submit via AJAX.

```
$(document).ready(function() {
  $('#myForm').ajaxForm();
});
```

Each plug-in is independently documented. To find out more about other plug-ins, we can explore the documentation linked from the jQuery plug-in repository, or read the explanatory comments found in the source code itself.

If we can't find the answers to all of our questions in the plug-in repository, the author's web site, and the comments within the plug-in, we can always turn to the jQuery discussion list. Many of the plug-in authors are frequent contributors to the list and are willing to help with any problems that new users might face. Instructions for subscribing to the discussion list can be found at `http://docs.jquery.com/Discussion`.

Appendix A, *Online Resources* lists even more resources for information about plug-ins and assistance in using them.

Developing a plug-in

As discussed earlier, plug-in development is a useful technique whenever we are going to perform a task more than once. Here we will itemize some of the components that can populate a plug-in file of our own design. Our plug-ins can use any combination of the following types of jQuery enhancements.

Object method

Add a new method to all jQuery objects created with the `$()` function.

```
jQuery.fn.methodName = methodDefinition;
```

Components

- `methodName`: A label for the new method
- `methodDefinition`: A function object to execute when `.methodName()` is called on a jQuery object instance

Description

When a function needs to act on one or more DOM elements, creating a new jQuery object method is usually appropriate. Object methods have access to the matched elements referenced by the jQuery object, and can inspect or manipulate them.

When we add a method to `jQuery.fn`, we are actually adding it to the **prototype** of the `jQuery` object. Because of JavaScript's native **prototypal inheritance**, our method will apply to every **instance** of the jQuery object. For more information about prototypal inheritance, see `https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide/Inheritance`.

The jQuery object can be retrieved from within the method implementation by referencing the `this` keyword. We can either call the built-in jQuery methods of this object, or we can extract the DOM nodes to work with them directly. As we saw in Chapter 8, *Miscellaneous Methods*, we can retrieve a referenced DOM node using array notation.

```
jQuery.fn.showAlert = function() {
  alert('You called the method on "' + this[0] + '".');
  return this;
}
```

However, we need to remember that a jQuery selector expression can always match zero, one, or multiple elements. We must allow for any of these scenarios when designing a plug-in method. The easiest way to accomplish this is to always call `.each()` on the method context. This enforces **implicit iteration**, which is important for maintaining consistency between plug-in and built-in methods. Within the function argument of the `.each()` call, `this` refers to each DOM element in turn.

```
jQuery.fn.showAlert = function() {
  this.each(function() {
    alert('You called the method on "' + this + '".');
  });
  return this;
}
```

Now we can apply our method to a jQuery object referencing multiple items.

```
$('.myClass').showAlert();
```

Our method produces a separate alert for each element that was matched by the preceding selector expression.

What is "this"?

It is very important to remember that the `this` keyword refers to different types of data in different situations. In the body of a plug-in method, `this` points to a jQuery object; in most callback functions such as `.each()` in our example, `this` points to a plain DOM element.

Note also that in these examples, we return the jQuery object itself (referenced by `this`) when we are done with our work. This enables the **chaining** behavior that jQuery users should be able to rely on. We must return a jQuery object from all plug-in methods, unless the method is clearly intended to retrieve a different piece of information and is documented as such.

A popular shorthand pattern for jQuery plug-ins is to combine the `.each()` iteration and the `return` statement as follows:

```
jQuery.fn.showAlert = function() {
  return this.each(function() {
    alert('You called the method on "' + this + '".');
  });
}
```

This has the same effect as the previous code block—enforcing implicit iteration and enabling chaining.

Global function

Make a new function available to scripts contained within the jQuery namespace.

```
jQuery.pluginName = fnDefinition;
jQuery.extend({
  pluginName: fnDefinition
});
jQuery.pluginName = {
  function1: fnDefinition1,
  function2: fnDefinition2
};
```

Components (first and second versions)

- `pluginName`: The name of the current plug-in
- `fnDefinition`: A function object to execute when `$.pluginName()` is called

Components (third version)

- `pluginName`: The name of the current plug-in
- `function1`: A label for the first function
- `fnDefinition1`: A function object to execute when `$.pluginName.function1()` is called
- `function2`: A label for the second function
- `fnDefinition2`: A function object to execute when `$.pluginName.function2()` is called

Description

What we call **global functions** here are technically methods of the `jQuery` function object. Practically speaking, though, they are functions within a jQuery namespace. By placing the function within the jQuery namespace, we reduce the chance of name conflicts with other functions and variables in scripts.

Plug-ins with a single function

The first and second usages above illustrate the creation of a global function when the plug-in needs only a single function. By using the plug-in name as the function name, we can ensure that our function definition will not be trod on by other plug-ins (as long as the others follow the same guideline!). The new function is assigned as a property of the `jQuery` function object:

```
jQuery.myPlugin = function() {
  alert('This is a test. This is only a test.');
};
```

Now in any code that uses this plug-in, we can write:

```
jQuery.myPlugin();
```

We can also use the `$` alias and write:

```
$.myPlugin();
```

This will work just like any other function call and the alert will be displayed.

Plug-ins with multiple functions

In the third usage, we see how to define global functions when more than one is needed by the same plug-in. We encapsulate all of the plug-ins within a single namespace named after our plug-in.

```
jQuery.myPlugin = {
  foo: function() {
    alert('This is a test. This is only a test.');
  },
  bar: function(param) {
    alert('This function was passed "' + param + '".');
  }
};
```

To invoke these functions, we address them as members of an object named after our plug-in, which is itself a property of the global jQuery function object.

```
  $.myPlugin.foo();
  $.myPlugin.bar('baz');
```

The functions are now properly protected from collisions with other functions and variables in the global namespace.

In general, it is wise to use this second usage from the start, even if it seems only one function will be needed, as it makes future expansion easier.

Example: A simple print function

In the various examples in the preceding reference chapters, we have had the need to output information to the screen to illustrate method behaviors. JavaScript's `alert()` function is often used for this type of demonstration, but does not allow for the frequent, timely messages we needed on occasion. A better alternative is the `console.log()` function available to Firefox and Safari, which allows printing messages to a separate log that does not interrupt the flow of interaction on the page. As this function is not available to Internet Explorer prior to version 8, we used a custom function to achieve this style of message logging.

The Firebug Lite script (described in Appendix B, *Development Tools*) provides a very robust cross-platform logging facility. The method we develop here is tailored specifically for the examples in the preceding chapters.

To print messages onto the screen, we are going to call the `$.print()` function. Implementing this function is simple as shown in the following code snippet:

```
  jQuery.print = function(message) {
    var $output = jQuery('#print-output');

    if ($output.length === 0) {
      $output = jQuery('<div id="print-output" />')
        .appendTo('body');
    }
    jQuery('<div class="print-output-line" />')
      .html(message)
      .appendTo($output);
  };
```

Our function first determines whether a container exists for our messages. If no element with the `print-output` ID already exists, we create one and append it to the `<body>` element. Then, we make a new container for our message, place the message inside it, and append it to the `print-output` container.

Note that we use the `jQuery` identifier rather than `$` throughout the script to make sure the plug-in is safe in situations where `$.noConflict()` has been called.

Selector expression

Add a new way to find DOM elements using a jQuery selector string.

```
jQuery.extend(jQuery.expr[selectorType], {
  selectorName: elementTest
});
```

Components

- `selectorType`: The prefix character for the selector string, which indicates which type of selector is being defined. In practice, the useful value for plug-ins is `':'`, which indicates a pseudo-class selector.
- `selectorName`: A string uniquely identifying this selector.
- `elementTest`: A callback function to test whether an element should be included in the result set. If the function evaluates to `true` for an element, that element will be included in the resulting set; otherwise, the element will be excluded.

Description

Plug-ins can add selector expressions that allow scripts to find specific sets of DOM elements using a compact syntax. Generally, the expressions that plug-ins add are new pseudo-classes, identified by a leading : character.

The pseudo-classes that are supported by jQuery have the general format `:selectorName(param)`. Only the `selectorName` portion of this format is required; `param` is available if the pseudo-class allows parameters to make it more specific.

The element test callback receives the following four arguments, which it can use to determine whether the element passes the test:

- `element`: The DOM element under consideration. This is needed for most selectors.
- `index`: The index of the DOM element within the result set. This is helpful for selectors such as `:eq()` and `:lt()`.
- `matches`: An array containing the result of the regular expression that was used to parse this selector. Typically, `matches[3]` is the only relevant item in the array. In a selector of the `:selectorName(param)` form, the `matches[3]` item contains `param`—the text within the parentheses.
- `set`: The entire set of DOM elements matched up to this point. This parameter is rarely needed.

For example, we can build a pseudo-class that tests the number of elements that are child nodes of an element and call this new selector expression `:num-children(n)`:

```
jQuery.extend(jQuery.expr[':'], {
  'num-children': function(element, index, matches, set) {
    var count = 0;
    for (var node = element.firstChild; node; node =
                                        node.nextSibling) {
      if ( node.nodeType === 1 ) {
        count++;
      }
    }
    return count == matches[3];
  }
});
```

Now we can select all `<ul>` elements with exactly two child DOM elements, and turn them red:

```
$(document).ready(function() {
  $('ul:num-children(2)').css('color', 'red');
});
```

Plug-in conventions

Before sharing our plug-in with the world at `http://plugins.jquery.com/`, we should check to ensure that the code conforms to the following conventions.

Use of the $ alias

jQuery plug-ins may not assume that the `$` alias is available. Instead, the full jQuery name must be written out each time.

In longer plug-ins, many developers find that the lack of the `$` shortcut makes code more difficult to read. To combat this, the shortcut can be locally defined for the scope of the plug-in by defining and executing a function. The syntax for defining and executing a function at once looks like this:

```
(function($) {
  // Code goes here
})(jQuery);
```

The wrapping function takes a single parameter to which we pass the global `jQuery` object. The parameter is named `$`. So within the function, we can use the `$` alias with no conflicts.

Naming conventions

Plug-in files should be named `jquery.myPlugin.js`, where `myPlugin` is the name of our plug-in. This allows jQuery plug-ins to be easily distinguished from other JavaScript files.

Global functions within `myPlugin` should be named `jQuery.myPlugin()`, or should be grouped as methods of the `jQuery.myPlugin` object. This convention helps to guard against conflicts with other plug-ins.

API standardization

Methods defined by our plug-in must abide by the contract established by the jQuery API. In order to provide a consistent experience for plug-in users, these methods must observe the following rules:

- Methods should support implicit iteration
- Methods should preserve chaining unless otherwise explicitly documented
- Arguments to methods should provide reasonable and configurable defaults
- Method definitions must terminate with a semicolon (`;`) character to avoid errors during code compression

In addition to following these conventions, the API for the plug-in should be well-documented.

[Further details and related techniques can be found online, or in Chapter 11 of the book *Learning jQuery 1.3*.]