

# jQuery Mobile

## Plugins

## Lesson 1, Activity 2: Plugins

**Why Plugins?**

Not every situation calls for the development of a plugin, certainly, but it makes lots of sense to use plugins when you are writing JavaScript code to perform any complex task. First, plugins are a way to work in "the jQuery way": you write some functionality and then you (or others, if using your plugin) apply that functionality to a DOM element or set of elements. A plugin named `myPlugin` might be called when the page initializes as such:

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

Second, plugins (good plugins) allow for chaining, so that you can apply the plugin in a chain just like any other jQuery methods:

```
$('#element').myPlugin().addClass('enabled')
```

Third, plugins are a great way to organize code: with an objected-oriented approach to plugin development - the approach we'll take here - you can combine stateful data, private methods, optional configuration parameters, and other features into one central hunk of code.

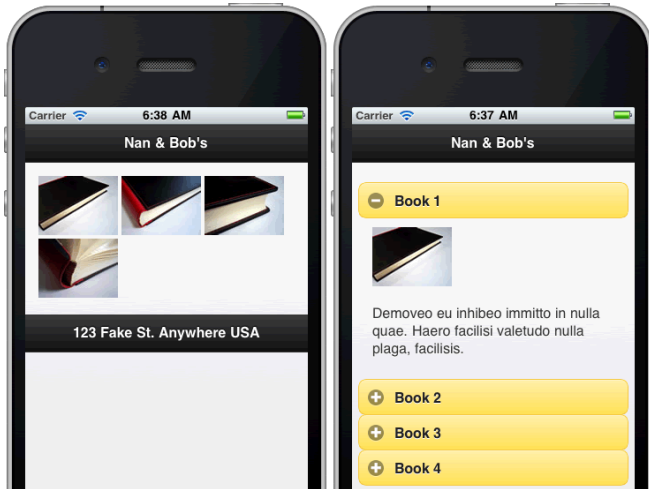
## Lesson 1, Activity 3: The jQuery UI Widget Factory

While there are a variety of approaches one can take when authoring plugins, we will here use the jQuery UI Widget Factory strategy [Recommended by jQuery Mobile](#). The Widget Factory "provides a flexible base for building complex, stateful plugins with a consistent API". See the [jQuery Development & Planning Wiki](#) for more info.

Let's look at how plugins are written by checking out an example. Suppose that some content for Nan & Bob's site comes from an external source - an XML syndication, say, or a database-backed CMS. Lots of the content happens to be images; the markup we inherit is a series of `img` tags wrapped in a `div`:

```
<div class="imgs">
  
  
  
  
</div>
```

It would be nice to present this inherited content in a manner both prettier and more useful for mobile visitors. We'll write a plugin to display these sets of images in a collapsible grid. The screenshot below shows (on the left) a demo page without applying the plugin, which we'll call "Collapsify", and the result (on the right) of applying the plugin. Note that the label for the collapsible header draws from each image's title attribute and that the text content inside the collapsible element draws from each image's `alt` attribute.



Open up [Plugins/Demos/index.html](#) to see the plugin in action. Note that the page links the JavaScript file `jquery.webucator.collapsify.js` - this is where we have written our plugin code.

On pageinit, we invoke the plugin:

```
$(document).on('pageinit',function(){
  $('div.imgs').collapsify({'datatheme':'e', 'openindex':0});
});
```

We are invoking the `collapsify` plugin on all divs of class `imgs` and passing a hash with two parameter name/value pairs: `datatheme` has value `e` and `openindex` has value `0`.

The interesting stuff is in the JavaScript code that defines the plugin itself - open up [Plugins/Demos/jquery.webucator.collapsify.js](#) in a file editor to check it out.

```
(function( $ ){
$.widget("webucator.collapsify", $.mobile.widget, {
  options: {
    datatheme: 'a',
    openindex: null
  },
  _create: function() {
    this.element.attr('data-role','collapsible-set').attr('data-theme',this.options.datatheme);
    this.images = this.element.children();
    var openindex = this.options.openindex;
    this.images.each(function(index) {
      $(this).wrap('<div data-role="collapsible" + (openindex == index ? ' data-collapsed="false" : '' + ' />').before('<h2>' + $(this).attr('title') + '</h2>').after('<p>' + $(this).attr('alt') + '</p>');
    });
    this.element.trigger('create');
  },
  _setOption: function(key, value) {
    this.options[key] = value;
    this._super( "_setOption", key, value );
  },
});
})( jQuery );
```

We wrap the code in `function ( $ )` - this adds our plugin to the jQuery object.

Next, we define our plugin. `webucator.collapsify` defines both a namespace (`webucator`, to avoid collisions with others' plugins) and the name (`collapsify`) we give to our plugin. The next parameter, `$.mobile.widget`, states that we are subclassing `widget`. The following opening brace (`{`) is the start of an object literal that defines the contents of our plugin.

The `options` field is a hash defining parameters to which we give default values; the user can override these values by supplying a hash when calling the plugin. Method `_setOptions` - a private method, because of the underscore at the start of its name - captures the option values, if any, supplied by the user. Note that we are overriding (and calling) the parent version of this method; often, we would want to add some logic in this method - validating some parameters, say, to conform to logical values.

Method `_create` is the constructor for our plugin. The first line adds two attributes to the `div` upon which the widget is called: we add `data-role="collapsible-set"` to make this a collapsible set, and we add the `data-theme` attribute to use the value from the `datatheme` option specified when initializing the plugin.

`this.element` refers to the object upon which the plugin is acting. Note that, in the context of the plugin, this refers to the object upon which the plugin is operating - not the DOM element. Thus we write code like `this.element.css('padding')`, rather than `$(this).element.css('padding')`. (That changes when we use a jQuery iterator to loop over the set of images: inside the `each` loop, as we'll see below, `$(this)` refers to each element of the loop.)

`openindex` is the option value (defaulting to `null`) of the item to be open on page load, indexed (like JavaScript arrays) starting at zero, not one.

Next, we loop over the images found in the `div` upon which our plugin is acting. Here we'll make use of the following jQuery functions:

- `.wrap()` wraps an HTML structure around the inner HTML of each element in the set of matched elements.
- `.before()` inserts an HTML structure or DOM element before each element in the set of matched elements.
- `.after()` inserts an HTML structure or DOM element before after element in the set of matched elements.

We wrap each image in a `div`, with attribute `data-role="collapsible"`, adding attribute `data-collapsed="false"` if we happen to be on the element specified to be open. We also add an `<h2>` and a `<p>`, the contents for which we get from the `title` and `alt` attributes of the image, respectively.

Last, we use `.trigger('create')` to refresh the DOM, to ensure that our changes show for the user, since the page's DOM will have already loaded.

Though it may look fairly complex, our example here is relatively simple: we haven't captured any state and manipulated the DOM just a little bit. Many plugins are extensive - handling state, capturing events, etc. - but this simple example shows the power of plugins.

## Lesson 1, Activity 5: The Listify Plugin

Duration: 20 to 30 minutes.

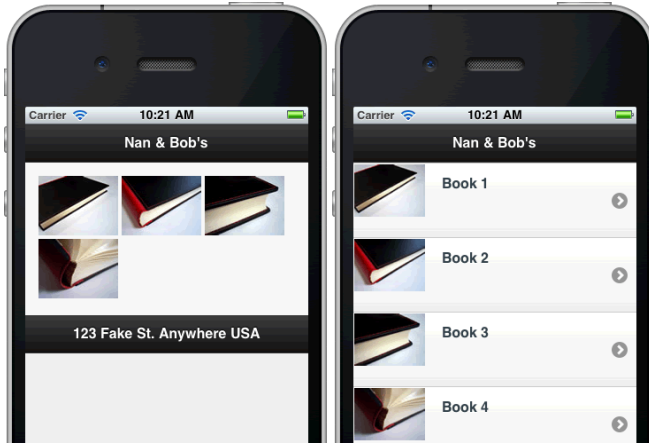
In this exercise, you will write a plugin to convert a set of images, wrapped in a div, into a jQuery Mobile listview. The markup we aim to act upon and modify is shown below, similar to that from the "Collapsify" example above:

```
<div class="listme" data-theme="a">




</div>
```

The screenshot below shows the above markup without the effect of the plugin, on the left; the right screenshot shows the page after the plugin is applied.



1. Open [Plugins/Exercises/listify/jquery.webucator.listify.js](#) - there's no need to edit the [index.html](#) file.
2. Where indicated, get the element's data-theme value - that is, the element being acted upon by the plugin (to which we will refer as "the element" going forward).
3. Assign the collection of images contained in the element to `this.images`.
4. Create a DOM element, `mylistview`, that is an unordered list with appropriate `data-role` and `data-theme` attributes to display as a jQuery Mobile listview.
5. Use the jQuery method `after` to add the new unordered list after the element.
6. Inside the each loop, append to `mylistview` a list item containing a link containing an image and some text; set `alt` and other attributes appropriately.
7. Remove the element being acted upon - we've replaced the `div` with the newly-created unordered list.
8. Unlike the "Collapsify" example above - in which we explicitly invoked our plugin from the [index.html](#) file - let's make this plugin run itself: where indicated at the bottom of the file, add a call to invoke the plugin inside the loop, thus applying `listify` to any `div` with class `listme`.
9. Test your work in a mobile browser.

**Solution:**

[Plugins/Solutions/listify/jquery.webucator.listify.js](#)

```
(function( $ ){
$.widget("webucator.listify", $.mobile.widget, {
  _create: function() {
    this.datatheme = this.element.attr('data-theme');
    this.images = this.element.children();

    mylistview = $('<ul data-role="listview" data-theme="'+ this.datatheme + '">');
    this.element.after(mylistview);
    this.images.each(function(index) {
      $(mylistview).append('<li><a href="'+ $(this).attr('data-link') + '">'+ $(this).attr('title') + '</a></li>');
    });
    this.element.remove();
    mylistview.trigger('create');
  },
  _setOption: function(key, value) {
    this.options[key] = value;
    this._super( "_setOption", key, value );
  },
});

$( ":jqmData(role='page')" ).live( "pagecreate", function() {
  $( "div.listme", this ).each(function() {
    $(this).listify();
  });
});

})( jQuery );
```

We get the element's theme with `this.element.attr('data-theme')`, saving it as `this.datatheme` for later use.

On the next line, `this.images = this.element.children()` assigns the set of images contained in the element to `this.images`.

We next create the unordered list that become the container for our listview, with appropriate `data-role` and (using the value we found previously) `data-theme` attributes.

The jQuery `after` method adds our new listview to the DOM, following the element being acted upon.

We use `each` to loop over the collection of images, appending a new list item/link/image for each

We then remove the original `div` element and use `trigger('create')` to refresh the DOM.

Lastly, we call `$(this).listify()` for each `div` of class `listme` on a page, binding this call (with the `live` method) to the `pagecreate` event.

Lesson 1, Activity 7: **Using Others' Plugins**

Much easier than writing your own code is to grab someone else's. Though still a relatively new framework, more and more third-party jQuery Mobile plugins are available - usually for free - to enhance the user experience on your sites. The [Google Maps plugin](#), for example, makes it easy to add maps to your pages, while the [Datebox plugin](#) offers an easy way to add a date picker. You can find more examples on the [jQuery Mobile website](#).

In the next exercise, you will download and implement the ActionSheet plugin, an easy way to add a nice modal dialog:



## Lesson 1, Activity 8: Implementing the ActionSheet Plugin

Duration: 10 to 15 minutes.

In this exercise, you will download the ActionSheet plugin and use it to produce a simple modal dialog.

1. Visit Stefan Gebhardt's [ActionSheet page on GitHub](#) to download the plugin's JavaScript and CSS files.
2. Open [Plugins/Exercises/actionsheet/index.html](#) in a file editor.
3. Move the downloaded files to [Plugins/Exercises/actionsheet/](#) and link them from the [index.html](#) page.
4. Use the [plugin documentation](#) to add some sample content. Note that you won't need to write any JavaScript - the plugin relies solely on data attributes in the markup.
5. Test your work in a mobile browser.

**Solution:**Plugins/Solutions/actionsheet/index.html

```
<a data-icon="plus" data-role="actionsheet">Open Dialog</a>
<div>
  <a data-role="button" href="#page1">Page 1</a>
  <a data-role="button" href="#page2">Page 2</a>
  <br/>
  <a data-role="button" data-rel="close" href="#">Cancel</a>
</div>
```

After downloading and linking the JavaScript and CSS files, using the plugin is quite simple. We add a button - a standard a tag - with attribute data-role="actionsheet". This becomes the toggle that opens the dialog.

The content for the opened dialog comes from the HTML element immediately following the data-role="actionsheet" toggle. Alternately, as the documentation states, we could have connected the toggle and dialog content with an attribute data-sheet on the toggle and an id on the dialog content. We add an optional "cancel" button (with attribute data-rel="close") in the dialog content.

Check out the JavaScript code for the plugin, too: open up [Plugins/Solutions/actionsheet/jquery.mobile.actionsheet.js](#). You'll note that the plugin adopts the UI Widget Factory paradigm. Also note that, at the bottom, the plugin calls itself:

```
$( ":jqmData(role='page')" ).live( "pagecreate", function() {
  $( ":jqmData(role='actionsheet')", this ).each(function() {
    $(this).actionsheet();
  });
});
```

This code says: "When the page has finished loading, check each page to find all elements with attribute data-role="actionsheet" and call the actionsheet() plugin upon them."