

jQuery Fundamentals Training

Plugins

Lesson 1, Activity 2: A jQuery plugin is simply a new method that we use to extend jQuery's prototype object. By extending the prototype object you enable all jQuery objects to inherit any methods that you add. Once the plugin is established, whenever you call `$()` you're creating a new jQuery object, with your plugin now method included along with all of jQuery's existing methods.

The idea of a plugin is to do something with a collection of elements. You could consider each method that comes with the jQuery core a plugin, like `fadeOut` or `addClass`.

You can make your own plugins and use them privately in your code or you can release them into the wild. There are thousands of jQuery plugins available online. The barrier to creating a plugin of your own is so low that you'll want to do it straight away!

How to Create a Basic Plugin

The typical approach for creating a plugin is as follows:

```
(function($) {  
  $.fn.myNewPlugin = function() {  
    return this.each(function() {  
      // do something  
    });  
  };  
})(jQuery);
```

The outer wrapper is an immediately-invoked function, inside which we create the plugin:

```
(function($) {  
  //...
```

```
}(jQuery));
```

This has the effect of creating a "private" scope that allows us to extend jQuery using the dollar symbol without having to risk the possibility that the dollar has been overwritten by another library. Although it is not required that you establish the plugin this way, this is the conventional approach.

The most important aspect of creating a plugin is to assign the method to the base jQuery object, which is what this part accomplishes:

```
$.fn.myNewPlugin = function() { ... }
```

`$.fn` is the base object containing all the jQuery collection methods. Any methods we add to this base object will be available to anything retrieved with `$(selector)`.

The `this` keyword within the new plugin refers to the jQuery object on which the plugin is being called.

Your typical jQuery object will contain references to any number of DOM elements, which is why jQuery objects are often referred to as collections.

To do something with a collection we need to loop through it, which is most easily achieved using jQuery's `each()` method:

```
$.fn.myNewPlugin = function() {
  return this.each(function() {
    // perform operations on individual elements
    // which will be available as "this"
  });
};
```

jQuery's `each()` method, like most other jQuery methods, returns a jQuery object, thus enabling what we've all come to know and love as "chaining", for example: `$(...).css().attr(...)`. We wouldn't want to break this convention, so we return the `this` object. Within this loop you can do whatever you want with each element. Here's an example of a small plugin using some of the techniques we've discussed, intended to show a hyperlink's url as part of its displayed text:

```
(function($){
$.fn.showLinkLocations0 = function() {
return this.each(function(){
$(this).append(' (' + $(this).attr('href') + ')');
});
};
}(jQuery));
```

This code appends text to every element it finds, showing that elements' href attribute in parentheses. The problem with this approach is that it will do that indiscriminately, regardless of whether an element is actually an `a` tag or not. Since we have no control over what selector is used, this will cause a problem. Consider this version of the plugin:

```
(function($){
$.fn.showLinkLocations1 = function() {
return this.filter('a').each(function(){
$(this).append(' (' + $(this).attr('href') + ')');
}).end();
};
}(jQuery));
```

Now we are filtering the elements, and only operating on `a` tags. Note

the call to `end()` at the end of the chain. `filter()` will produce a subset of the original set, but whoever uses the function would probably rather have it return the original set.

Testing Our Plugin

OK, great, so we're done. Open up jqy-plugins/Demos/showLinkLocation.html and let's see what different versions of the demo look like...

Version 1 of our Plugin

From the screenshot below, you can see how the plugin works when invoking the `showLinkLocations1()` function. Note that not every link is affected. We select all elements with css class `".special"`, and links with this class do get changed. But, links within the `<div class="special">` do not. `filter` finds a subset of the current set of elements, but does not select descendants of the current set. We would probably like a tags that are contained within any selected elements to be changed as well.

[Webucator \(http://www.webucator.com/\)](http://www.webucator.com/) [Home \(index.html\)](#) [More \(more.html\)](#)

This is special, too!

[Webucator](#) [Home](#) [More](#)

This is not special!

Version 2 of our Plugin

```
(function($){
$.fn.showLinkLocations2 = function() {
return this.find('a').each(function(){
$(this).append(' (' + $(this).attr('href') + ')');
```

```

    }).end();
  };
}(jQuery));

```

This version uses `find` instead of `filter`, and finds all descendant `a` tags (notice the screenshot below). But, now the problem is reversed -- `find` will produce only descendant elements within the original selection, and omit the `a` tags that directly selected by the query.

[Webucator](#) [Home](#) [More](#)

This is special, too!

[Webucator \(http://www.webucator.com/\)](http://www.webucator.com/) [Home \(index.html\)](#) [More \(more.html\)](#)

This is not special!

Final Version of our Plugin

So, we need to use both approaches, as shown below.

Code Sample:

<jqy-plugins/Demos/showLinkLocation.html>

```
.redText { color: red; }
```

[Webucator](#)
[Home](#)
[More](#)

This is special, too!

[Webucator](#)
[Home](#)

[More](#)

[Webucator](#)

[Home](#)

[More](#)

This is not special!

This plugin first finds all the descendants of a selection, then backs up and uses filter, and then backs up again to return the original selection (which is confirmed by the font size change in the one `p.special` element).

[Webucator \(http://www.webucator.com/\)](http://www.webucator.com/) [Home \(index.html\)](#) [More \(more.html\)](#)

This is special, too!

[Webucator \(http://www.webucator.com/\)](http://www.webucator.com/) [Home \(index.html\)](#) [More \(more.html\)](#)

This is not special!

Here's another example of a plugin. This one doesn't require us to loop through every element with the `each()` method. Instead, we're simply going to delegate to other jQuery methods directly:

```
(function($){
$.fn.fadeInAndAddClass = function(duration, className) {
return this.fadeIn(duration, function(){
$(this).addClass(className);
});
};
})(jQuery);

// Usage example:
$('a').fadeInAndAddClass(400, 'finishedFading');
```



Lesson 1, Activity 4: **Finding and Evaluating Plugins**

Plugins extend the basic jQuery functionality, and one of the most celebrated aspects of the library is its extensive plugin ecosystem. From table sorting to form validation to autocompletion. If there's a need for it, chances are good that someone has written a plugin for it.

The quality of jQuery plugins varies widely. Many plugins are extensively tested and well-maintained, but others are hastily created and then ignored. More than a few fail to follow best practices.

Google is your best initial resource for locating plugins, though the jQuery team is working on an improved plugin repository. Once you've identified some options via a Google search, you may want to consult the jQuery mailing list or the #jquery IRC channel to get input from others.

When looking for a plugin to fill a need, do your homework. Ensure that the plugin is well-documented, and look for the author to provide lots of examples of its use. Be wary of plugins that do far more than you need; they can end up adding substantial overhead to your page. For more tips on spotting a subpar plugin, read *Signs of a poorly written jQuery plugin* by Remy Sharp.

Once you choose a plugin, you'll need to add it to your page. Download the plugin, unzip it if necessary, place it in your application's directory structure, then include the plugin in your page using a script tag (after you include jQuery).

The Mike Alsup jQuery Plugin Development Pattern

For more on plugin development, read Mike Alsup's essential post, *A Plugin Development Pattern*

(<http://www.learningjquery.com/2007/10/a-plugin-development-pattern>).

In it, he creates a plugin called `$.fn.hilight`, which provides support for the jQuery *Metadata Plugin* if it's present, and provides a centralized method for setting global and instance options for the plugin.

The metadata plugin extracts options from data contained in the first element in a collection, such as:

```
<p class="special { foreground: white; background: blue; }">Test</p>
```

The sample plugin below highlights elements by setting foreground and background colors, and then wrapping the HTML in a `` tag.

Code Sample:

jqy-plugins/Demos/Alsup-jQuery.js

```
(function($) {

    // define plugin and add to $.fn
    $.fn.hilight = function(options) {

        // demonstrates use of private function
        debug(this);

        // build main options before element iteration
        // extending built-in defaults with passed in options
        var opts = $.extend({}, $.fn.hilight.defaults, options);

        // iterate and reformat each matched element
        return this.each(function() {
            $this = $(this);

            // build element specific options from metadata
            // extending current options with metadata options
            // leave this out if not supporting metadata
            var o = $.meta ? $.extend({}, opts, $this.data()) : opts;
```

```

// update element styles
$this.css({
  backgroundColor: o.background,
  color: o.foreground
});

var markup = $this.html();

// call our format function
markup = $.fn.hilight.format(markup);
$this.html(markup);
});

// private function for debugging
function debug($obj) {
  if (window.console && window.console.log)
    window.console.log('hilight selection count: ' + $obj.size());
};

// define and expose format function by adding it to $.fn.hilight
$.fn.hilight.format = function(txt) {
  return '<strong>' + txt + '</strong>';
};

// built-in defaults, also added to $.fn.hilight
$.fn.hilight.defaults = {
  foreground: 'red',
  background: 'yellow'
};

})(jQuery);

```

The line

```
var opts = $.extend({}, $.fn.hilight.defaults, options);
```

creates an object starting with the contents of `defaults`, and overriding with any values in `options`. We do this before we start processing any elements in the collection.

Then, for each individual element, we extend the options again with

the data retrieved from the tag metadata, if any exists.

```
var o = $.meta ? $.extend({}, opts, $this.data()) : opts;
```

The `markup` function has been exposed by adding it to `$.fn.highlight`, meaning that you can replace it. The HTML page that uses this does that before the second application of the plugin.

The `debug` function has not been exposed, so it is only available within the plugin's code.

Lesson 1, Activity 5: Creating a Plugin Using the Alsup Pattern

Duration: 5 to 10 minutes.

1. The exercise file [jqy-plugins/Exercises/js/stripe.js](#) contains a table-stripping function `Globals.stripe` that will only stripe one level of table rows -- it will not descend into child tables. The file [table.html](#) contains a table with nested tables, and invokes the stripping function separately for the outer and inner tables.
2. Your job is to turn the stripe function into a plugin using the Alsup pattern (don't bother with the metadata support, though).
3. The odd and even colors should now be options.
4. Change the call to `css` after the second stripping operation to be chained on after the stripping, since our plugin should allow this.
5. Right now, the plugin stripes all rows, regardless of what type of table section they are in. As a challenge, see if you can modify the plugin to have a default option of stripping only the `tbody`, which can be overridden by the `options` object passed to the plugin to stripe all rows instead. Hint: the `children` function can take a query string, which can be used to filter the set of tags it returns.

Solution:

[jqy-plugins/Solutions/js/stripe.js](#)

```
(function($) {

$.fn.stripe = function(options) {
  // override defaults with options
  var opts = $.extend({}, $.fn.stripe.defaults, options);

  // start with empty set
  var $tables = $()
  // add any tables that are direct entries in current collection
  .add($this.filter('table').get())
  // add ones that are descendants of entries in current collection
  .add($this.find('table').get());
  // now find any tables that are descendants of the tables we found
  var $omit = $tables.find('table');
  // and omit them from the collection
```

```

$tables = $tables.not($omit);

var oddColor = opts.oddColor ? opts.oddColor : null;
var evenColor = opts.evenColor ? opts.evenColor : null;

$tables.each(function() {
  // this finds only grandchildren (table -> table section -> tr)
  var $sections = $(this).children();
  if (oddColor) $sections.children('tr:odd')
    .css('backgroundColor', oddColor);
  if (evenColor) $sections.children('tr:even')
    .css('backgroundColor', evenColor);
});
return this;
};

$.fn.stripe.defaults = {
  evenColor: '#ddddff'
}

}(jQuery));

```

The function has been stored in `$.fn` now, instead of our `Globals` object. Similarly, near the end, we store our default color in `$.fn.stripe`.

The first parameter has been removed, since that will now be the query upon which our plugin acts. Because of that, within the code, the `$(query)` has been replaced with `this`.

Before we start finding tables, we override the defaults with any incoming options.

To properly behave as a jQuery collection method, we return `this` at the end.

Challenge Solution:

<jqy-plugins/Solutions/js/stripe-challenge.js>

```

(function($) {
$.fn.stripe = function(options) {
  var $this = $(this);

  // override defaults with options
  var opts = $.extend({}, $.fn.stripe.defaults, options);

  // start with empty set
  var $tables = $()
  // add any tables that are direct entries in current collection
  .add($this.filter('table').get())
  // add ones that are descendants of entries in current collection
  .add($this.find('table').get());
  // now find any tables that are descendants of the tables we found
  var $omit = $tables.find('table');
  // and omit them from the collection
  $tables = $tables.not($omit);

  var oddColor = opts.oddColor ? opts.oddColor : null;
  var evenColor = opts.evenColor ? opts.evenColor : null;

  $tables.each(function() {
    // this finds only grandchildren (table -> table section -> tr)
    if (opts.tbodyOnly) var $sections = $(this).children('tbody');
    else var $sections = $(this).children();
    if (oddColor) $sections.children('tr:odd')
      .css('backgroundColor', oddColor);
    if (evenColor) $sections.children('tr:even')
      .css('backgroundColor', evenColor);
  });
  return this;
};

$.fn.stripe.defaults = {
  evenColor: '#ddddff',
  tbodyOnly: true
}
}(jQuery));

```

We have added a `tbodyOnly` property to the defaults object. The code to find the tables' children has been split with a conditional to either filter with `'tbody'` or not.

Lesson 1, Activity 7: Writing Stateful Plugins with the jQuery UI Widget Factory

Note: Rebecca Murphey originally based this section, with permission, on the blog post *Building Stateful jQuery Plugins* by Scott Gonzalez. It has been modified further.

While most existing jQuery plugins are stateless, that is, we call them on an element and that is the extent of our interaction with the plugin, there's a large set of functionality that doesn't fit into the basic plugin pattern.

In order to fill this gap, jQuery UI (<http://jqueryui.com/>) has implemented a more advanced plugin system. Their system manages state, allows multiple functions to be exposed within a single plugin, and provides various extension points. This system is called the *widget factory* and is exposed as `jQuery.widget`.

Although the widget factory it is created by jQueryUI, it can be used independently of the rest of the framework. jQueryUI's web site provides a means to build a custom JavaScript file with only selected elements from their toolkit. We have built a file with just the widget factory for your use.

A Simple, Stateful Plugin Using the jQuery UI Widget Factory

To demonstrate the capabilities of the widget factory, we'll build a simple progress bar plugin.

To start, we'll create a progress bar that just lets us set the progress once.

As we can see below, this is done by calling `jQuery.widget` with two parameters: the name of the plugin to create and an object literal containing functions to support our plugin. When our plugin gets called, it will create a new plugin instance -- all functions will be executed within the context of that instance. This is different from a standard jQuery plugin in two important ways:

- the context is an object, not a DOM element
- the context is always a single object, never a collection

To create a widget, invoke `$.widget` with two parameters: a widget name string, and an object containing an `options` property, a `_create` method, and other methods you will need.

```
$.widget (
  namespace.widgetName,
  {
    options: optionsObjectWithDefaultValues,
    _create: elementToWidgetInitializationFunction,
    optionsSetterGetterMethod(s)
  ,
    _privateMethod(s)
  }
);
```

Widget Naming Rules

The name of the plugin must contain a namespace, which must be exactly one level deep, that is, we can't use a namespace like `nmk.foo`.

Setting Default Options for a Widget

The second parameter to `$.widget` can contain an `options` property, which is an object containing optional properties as its fields. While technically this property is not required, there is not much you

can do without any options.

The object you provide will be used as the defaults, but those values can be overridden and/or changed later.

```
$.widget('nmk.progressbar', {
  // default options
  options: {
    value: 0
  },
  // rest of widget code
});
```

Initializing the Widget

The next element, the `_create` method, is required. This is the method that the widget factory will invoke to set up the widget. The `this` reference will refer to the current object (the one you are passing in to `$.widget`).

```
_create: function() {
  var progress = this.options.value + '%';
  this.element
    .addClass( 'progressbar' )
    .text( progress ).css({
      width: this.options.width * progress / 100 + 'px',
      backgroundColor: this.options.color,
      color: this.options.txtColor
    })
};
```

The `_create` method you supply will be used to turn the selected elements into instances of your widget. It would generally use the `options` as part of the initialization.

Adding Methods to a Widget

Since your widget is a jQuery object, if you were to add directly-callable methods to it, those methods would exist for all other jQuery objects as well. So, instead the UI factory reuses the widget name as a method, through which you can invoke methods in object you passed to

`$.widget`. The first parameter is the name of the function you wish to invoke, the subsequent parameters are passed on to that function.

If you wish to have the conventional getter/setter behavior of jQuery methods, it is up to you to write that into your methods. The default return value is the jQuery object representing the widget, but you can replace that for a getter method.

Methods whose names begin with an underscore are considered private -- they will not be exposed.

```
// create a public method
value: function(value) {
// no value passed, act as a getter
if (value === undefined) {
return this.options.value;
// value passed, act as a setter
} else {
this.options.value = this._constrain(value);
this._update();
}
},

// create private methods
_constrain: function(value) {
if (value > 100) {
value = 100;
}
if (value < 0) {
value = 0;
}
return value;
},

_update() {
var width = this.options.width * value / 100;
```

```

var progress = this.options.value + '%';
this.element.text(progress)
.css({'width': width + 'px' });
}

```

Working with Widget Options

In the above example, we wrote a method to set and get the `value` option. An alternative approach is to use one of the methods that is automatically available to your plugin: the `option` method. This method allows you to get and set options after initialization. This method works like jQuery's `css` and `attr` methods, except that you invoke the same way as other public methods, by passing "option" to your plugin method. Then, for the subsequent parameters, you can pass just a name to use it as a getter, a name and value to use it as a single setter, or a hash of name/value pairs to set multiple values. When used as a getter, the plugin will return the current value of the option that corresponds to the name that was passed in. When used as a setter, the plugin's `_setOption` method will be called for each option that is being set. We can write a `_setOption` method in our plugin to react to option changes.

Responding When an Option is Set

```

// _setOption method
_setOption: function(key, value) {
  if (key === 'value') {
    this.options.value = this._constrain(value);
    this._update();
  } else {
    this.options[key] = value;
  }
  this.element
    .css({
      backgroundColor: this.options.color,
      color: this.options.txtColor
    })
}

```

```
});  
}
```

Setting Options for a Plugin Instance

If you have written a `_setOptions` method, you invoke as you would any public method for your plugin, by using the plugin name as a method, and passing a string "option" as the first parameter. Then, either two parameters, an option name and a value, or an object hash containing *name: value* pairs.

```
$progBar3.progressbar(  
'option', { width: 100, color: '#ffccaa', txtColor: '#000000' }  
);
```

Adding a Widget Instance to a Page

By calling `$.widget`, you have added the short (no-namespace) version of the name as a method to `$.fn`. To add instances of your widget to a page, make a jQuery selection, then call that method, passing an options object if you wish to override any of the default option values.

```
var $progBar1 = $('#loadingIndicator').progressbar();  
  
var $progBar2 = $('')  
.appendTo('body')  
.progressbar( { width: '500', color: '#ffaacc', txtColor: '#0000aa' } );
```

Calling Methods on a Plugin Instance

To call one of your widget's public methods, you again use the widget name as a method, but now pass at least one parameter. The first parameter must be the desired method name as a string, and any

subsequent parameters will be passed as the parameters to that method.

```
// setter
$progBar1.progressbar('value', 50);

// getter
var current = $progBar1.progressbar('value');
```

Code Sample:

jqy-plugins/Demos/widget-factory-plugin-option.html

Creating & Using jQuery-UI Plugins - Setting Options

Progress Bar 1 (using defaults)

Progress Bar 2 (using custom values)

In this case we've used the namespace `nmk`. There is a limitation that namespaces be exactly one level deep, that is, we can't use a namespace like `nmk.foo`. We can also see that the widget factory has provided two properties for us. `this.element` is a jQuery object containing exactly one element. If our plugin is called on a jQuery object containing multiple elements, a separate plugin instance will be created for each element, and each instance will have its own `this.element`. The second property,

`this.options`, is a hash containing key/value pairs for all of our plugin's options. These options can be passed to our plugin as shown here.

Note: In our example we use the `nmk` namespace. The `ui` namespace is reserved for official jQuery UI plugins. When building your own plugins, you should create your own namespace. This makes it clear where the plugin came from and whether it is part of a larger collection.

When we call `jQuery.widget` it extends jQuery by adding a method to `$.fn` (the same way we'd create a standard plugin). The name of the function it adds is based on the name you pass to `$.widget`, without the namespace; in our case it will create `$.fn.progressbar`. The options passed to our plugin get set in `this.options` inside of our plugin instance. As shown, we can specify default values for any of our options. When designing your API, you should figure out the most common use case for your plugin so that you can set appropriate default values and make all options truly optional.

Adding Callbacks to a Plugin

One of the easiest ways to make your plugin extensible is to add callbacks so users can react when the state of your plugin changes. These can be either options that store functions or custom event bindings.

We can see below how to add a callback to our progress bar to signify when the progress has reached the halfway point.

Providing Callbacks for User Extension

```
_update: function() {
var width = this.options.width * this.options.value / 100;
var progress = this.options.value + '%';
this.element.text(progress)
.css({'width': width + 'px' });

// exercise the callback
if (this.options.value == 50) {
if (this.options.halfway) this.options.halfway(this);
}
}

// the callback function
var $progBar1 = $('#loadingIndicator')
.progressbar({
halfway: function() {
$('Halfway!').insertAfter($progBar1);
}
});
```

Callback functions are essentially just additional options, so you can get and set them just like any other option.

Adding Custom Event Callbacks

A more loosely-coupled form of callback is a custom event, which can be bound and triggered using jQuery's event-management capabilities. The `_trigger` method in jQueryUI widgets takes three parameters: the name of the event, a native event object that initiated the callback, and a hash of data relevant to the event. The event name is the only required parameter, but the others can be very useful for users who want to implement custom functionality for a plugin. For example, if we were building a draggable plugin, we could pass the native `mousemove`

event when triggering a drag callback; this would allow users to react to the drag based on the x/y coordinates provided by the event object. Whenever a callback is executed, a corresponding event is triggered as well.

The event type is determined by concatenating the plugin name and the event name. The event handler receives two parameters: an event object and a hash of data relevant to the event, as we'll see below.

Binding to widget events

```
_update: function() {
  var width = this.options.width * this.options.value / 100;
  var progress = this.options.value + '%';
  this.element.text(progress)
  .css({'width': width + 'px' });
  if (this.options.value == 100) {
    this._trigger('complete', null, { value: 100 });
  }
}
```

If your plugin has functionality that you want to allow the user to prevent, the best way to support this is by creating cancelable callbacks. Users can cancel a callback, or its associated event, the same way they cancel any native event: by calling `event.preventDefault()` or using `return false`. If the user cancels the callback, the `_trigger` method will return `false` so you can implement the appropriate functionality within your plugin.

The Widget Factory: Under the Hood

When you call `jQuery.widget`, it creates a constructor function for your plugin and sets the object literal that you pass in as the prototype for your plugin instances. All of the functionality that automatically gets

added to your plugin comes from a base widget prototype, which is defined as `jQuery.Widget.prototype`. When a plugin instance is created, it is stored on the original DOM element using `jQuery.data`, with the plugin name as the key.

Because the plugin instance is directly linked to the DOM element, you can choose to access the plugin instance directly instead of going through the exposed plugin method. This will allow you to call methods directly on the plugin instance instead of passing method names as strings and will also give you direct access to the plugin's properties.

```
var bar1 = $progBar1.data('progressbar');

// succeeds in changing value, but the text color was set via _create
// and not looked at again
bar1.options.txtColor = 'black';

// but _setOption causes element css to be modified
bar1._setOption( 'width', 600 );
```

One of the biggest benefits of having a constructor and prototype for a plugin is the ease of extending the plugin. By adding or modifying methods on the plugin's prototype, we can modify the behavior of all instances of our plugin. For example, if we wanted to add a method to our progress bar to reset the progress to 0% we could add this method to the prototype and it would instantly be available to be called on any plugin instance.

```
$.nmk.progressbar.prototype.setWidth = function(width) {
  this.options.width = width;
};
var bar2 = $progBar2.data('progressbar');
bar2.setWidth( 100 );
```

Cleaning Up

In some cases, it will make sense to allow users to apply and then later unapply your plugin. You can accomplish this via the `destroy` method. Within the `destroy` method, you should undo anything your plugin may have done during initialization or later use. The `destroy` method is automatically called if the element that your plugin instance is tied to is removed from the DOM, so this can be used for garbage collection as well. The default `destroy` method removes the link between the DOM element and the plugin instance, so it's important to call the base function from your plugin's `destroy` method.

Adding a Destroy Method to a Widget

```
destroy: function() {
  this.element
    .removeClass('progressbar')
    .text('')
    .css({
      width: '',
      backgroundColor: '',
      color: ''
    });

  // call the base destroy function
  $.Widget.prototype.destroy.call(this);
}
```

Conclusion

The widget factory is only one way of creating stateful plugins. There are a few different models that can be used and each have their own advantages and disadvantages. The widget factory solves lots of common problems for you and can greatly improve productivity, it also

greatly improves code reuse, making it a great fit for jQuery UI as well as many other stateful plugins.

Lesson 1, Activity 8: **Make a Table Sortable (Optional)**

Duration: 5 to 10 minutes.

For this exercise, your task is to identify, download, and implement a table sorting plugin on the index.html page (located at [jqy-plugins/Exercises](#)). When you're done, all columns in the table on the page should be sortable.