

jQuery Fundamentals Training

Best Practices

Lesson 1, Activity 2: JavaScript Best Practices

Namespacing Variables

Avoid polluting the global namespace (the elements owned by window). Instead use self-executing anonymous functions or variables and functions owned by a single, global object.

Global Namespace Variables

Rather than declaring a number of separate global items, which can result in name collisions, use the same practice that jQuery uses -- create one global object that owns everything else.

```
var MyGlobal = {  
  someVariable: 5;  
  someFunction = function(a) { ... }  
}  
  
MyGlobals.someVariable = 10;  
MyGlobals.someFunction(3);
```

Self-Executing Anonymous Functions

We have seen these many times already. They contain variables that are global to any functions defined inside the self-executing functions scope, and, as such, are good for jQuery code to run when the document is ready. They are also good for adding plugins to jQuery or another external global variable.

Cache Frequently Used Values

Cache Length During Loops

In a for loop, don't access the length property of an array every time;

cache it beforehand.

```
var myLength = myArray.length;

for (var i = 0; i < myLength; i++) {
  // do stuff
}
```

Cache Retrieved Elements, Chains of Element Lookups, and jQuery Collections

Although `document.getElementById` is optimized, it is still inefficient to do it multiple times for the same element. Retrieve it once and store it in a variable. Even if the DOM changes around it, the reference is still valid.

Following chains of object property references can be expensive, particularly if done in loops.

As mentioned before, a query can be expensive, particularly if it is compiled, so caching the results will help speed up code if the collection won't change.

Append New Content Outside of a Loop

Touching the DOM comes at a cost; if you're adding a lot of elements to the DOM, do it all at once, not one at a time.

```
// This is bad
$.each(myArray, function(i, item) {
  var newListItem = '<li>' + item + '</li>';
  $('#myList').append(newListItem);
});

// Better: do this
// A document fragment is a node that is a container -
// when you append it somewhere, the fragment node
// disappears, but its contents are appended
```

```

var frag = document.createDocumentFragment();

$.each(myArray, function(i, item) {
    var newListItem = '<li>' + item + '</li>';
    frag.appendChild(newListItem);
});
$('#myList')[0].appendChild(frag);

// Or do this
var myHtml = '';

$.each(myArray, function(i, item) {
    html += '<li>' + item + '</li>';
});
$('#myList').html(myHtml);

```

Beware Anonymous Functions

Despite our use of them throughout the course, too many anonymous functions bound everywhere are a pain. They're difficult to debug, maintain, test, or reuse. Instead, use an object literal to organize and name handlers and callbacks that are at all complex or lengthy.

```

// Bad
$(document).ready(function() {
    $('#go').click(function(e) {
        $('#details').slideDown(function() {
            ...
        });
    });
});

$('#results').load('somepage.html #' + tag, function() {
    ...
});

// Better
var Globals = {
    url: 'somepage.html',

    onReady : function() {
        $('#go').click(Globals.showDetails);
        $('#results').load(Globals.url + ' #' + tag, Globals.afterLoad);
    },

```

```
showDetails : function(e) {  
    $('#details').slideDown(Globals.afterSlide);  
},  
  
afterSlide : function() { ... },  
  
afterLoad : function() { ... }  
};  
  
$(document).ready(Globals.onReady);
```

Also note the moving of the embedded url to a variable - it is tough to find things like that when they need to be edited if they are inline in code.

Lesson 1, Activity 4: jQuery Best Practices

Optimize Selectors

Selector optimization is less important than it used to be, as more browsers implement `document.querySelectorAll()` and the burden of selection shifts from jQuery to the browser. However, there are still some tips to keep in mind.

ID-Based Selectors

Beginning your selector with an ID is always best. An item preceding an id will actually de-optimize a query, since it will induce jQuery to use `getElementsByTagName` first, then search by attribute for the id, when the browsers already optimize `getElementById`.

```
//Not Optimized
$('ul#myId li');

// Better
$('#myId li');
```

Design Pages for jQuery Optimization, and Use Your Knowledge of the Page Structure

Adopting and enforcing rules on your HTML code will help create optimized selectors.

- Use classes to categorize elements.
- Define and use consistent structures for like elements. For instance, "flyout menus will be implemented as unordered lists, with a specific class for the top level" (and possibly for sublevels as well).
- Use table section elements, particularly `tbody`, so that you know that rows will be exactly two levels below the `table` tag.

A "flatter" DOM also helps improve selector performance, as the selector engine has fewer layers to traverse when looking for an element.

Avoid the Universal Selector

Selections that specify or imply that a match could be found anywhere can be very slow, particularly if the starting point is not very specific.

```
$('[name=city]');           // extremely expensive
$('input[name=city]');      // better
$('form input[name=city]'); // even better*
$('#custForm input[name=city]'); // best*
```

*Note that this assumes that the input is within the form, which is no longer required in HTML 5.

Use "Safe" Selectors

You should always take care when creating selectors to avoid selecting unwanted elements, particularly when working with nested lists, tables, divs, etc., or when you anticipate that a selector will find only a single tag.

Omitting Nested Elements

One approach is to use the child selector `>` in your selector:

```
$('#myList>li:odd')
```

Another way is to use the `children()` filter function:

```
$('#myList').children('li:odd')
```

Selecting a Single Element

With an id selector, you can assume that your query will find only a single element. But, with other selectors, that wouldn't necessarily be true. In this case, it doesn't affect your efficiency much to add a `:first` pseudo-selector to the query. In the example below, perhaps we only want to select one row, and are assuming that the `thead` section only contains one `tr` tag:

```
$('#myTable thead tr')          // would find multiple trs if they existed  
$('#myTable thead tr:first')  // better
```

Use Event Delegation

Event delegation allows you to bind an event handler to one container element (for example, an unordered list) instead of multiple contained elements (for example, list items). jQuery makes this easy with `$.fn.live` and `$.fn.delegate`. Where possible, you should use `$.fn.delegate` instead of `$.fn.live`, as it eliminates the need for an unnecessary selection, and its explicit context (vs. `$.fn.live`'s context of document) reduces overhead by approximately 80%.

In addition to performance benefits, event delegation also allows you to add new contained elements to your page without having to re-bind the event handlers for them as they're added.

```
// Bad (if there are lots of list items)  
$('li.trigger').click(handlerFn);  
  
// Best: event delegation  
$('#myList').delegate('li.trigger', 'click', handlerFn);
```


Note that As of jQuery 1.7, [.delegate\(\)](#) has been deprecated in favor of the [.on\(\)](#) method.

```
//for jQuery 1.4.3+, use delegate()
$(elements).delegate(selector, events, data, handler);

//for jQuery 1.7+, use on()
$(elements).on(events, selector, data, handler);
```

Detach Elements to Work With Them

The DOM is slow; you want to avoid manipulating it as much as possible. jQuery introduced `$.fn.detach` in version 1.4 to help address this issue, allowing you to remove an element from the DOM while you work with it.

```
var $table = $('#myTable');
var $parent = $table.parent();

$table.detach();
// add lots and lots of rows to table
$parent.append($table);
```

Use Stylesheets for Changing CSS on Many Elements

If you're changing the CSS of more than 20 elements using `$.fn.css`, consider adding a style tag to the page instead for a nearly 60% increase in speed.

```
// OK for up to 20 elements, slow after that
$('a.swedberg').css('color', '#abcdef');

//better if styling many elements
$('a.swedberg { color : #abcdef }')
.appendTo('head');
```

Note that the above is not an exact replacement -- while the original will result in an inline style, which would override any existing inline style, the alternative presented would not override an existing inline style.

Use `$.data` Instead of `$.fn.data`

Using `$.data` on a DOM element instead of calling `$.fn.data` on a jQuery selection can be up to 10 times faster. Be sure you understand the difference between a DOM element and a jQuery selection before doing this, though.

```
// Slower
$(elem).data(key,value);

// Faster
$.data(elem,key,value);
```

Also, remember that the need for `data` can be almost completely eliminated by the use of closures, as we have done in most of the demos and exercises.

Don't Act on Absent Elements

jQuery won't tell you if you're trying to run a whole lot of code on an empty selection; it will proceed as though nothing's wrong. It's up to you to verify that your selection contains some elements.

```
// Bad: this runs two functions, each of which
// finds there's nothing in the selection
$('#nosuchthing').addClass('defunct').slideUp();
```

```
// Better - test first
var $mySelection = $('#nosuchthing');
if ($mySelection.length) {
    $mySelection.addClass('defunct').slideUp();
}

// Best: add a plugin to coalesce functions into one
jQuery.fn.doOnce = function(func){
    if (this.length) func.apply(this);
    return this;
}

$('#nosuchthing').doOnce(function() {
    this.addClass('defunct').slideUp();
});
```

This guidance is especially applicable for jQuery UI widgets, which have a lot of overhead even when the selection doesn't contain elements.

Lesson 1, Activity 6: Code Organization

Key Concepts

Before we jump into code organization patterns, it's important to understand some concepts that are common to all good code organization patterns.

- Your code should be divided into units of functionality: modules, services, etc. Avoid the temptation to have all of your code in one huge `$(document).ready()` block. This concept, loosely, is known as encapsulation.
- Don't repeat yourself. Identify similarities among pieces of functionality, and use inheritance techniques to avoid repetitive code.
- Despite jQuery's DOM-centric nature, JavaScript applications are not all about the DOM. Remember that not all pieces of functionality need to, or should, have a DOM representation.
- Units of functionality should be *loosely coupled*: a unit of functionality should be able to exist on its own, and communication between units should be handled via a messaging system such as custom events or publish/subscribe. Stay away from direct communication between units of functionality whenever possible.

The concept of loose coupling can be especially troublesome to developers making their first foray into complex applications, so be mindful of this as you're getting started.

Encapsulation

The first step to code organization is separating pieces of your application into distinct pieces; sometimes, even just this effort is sufficient to dramatically improve the reusability and editability of the

code.

The Object Literal

An object literal is perhaps the simplest way to encapsulate related code. It doesn't offer any privacy for properties or methods, but it's useful for eliminating anonymous functions from your code, centralizing configuration options, and easing the path to reuse and refactoring.

Code Sample:

jqy-practices/Demos/object-literal.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="ISO-8859-1">
  <title>Using an Object Literal to Encapsulate Functionality</title>
</head>
<body>
<p>Nothing to show here.</p>
<script src="../../jqy-lib/jquery.js"></script>
<script>
var myFeature = {
  message : 'hello',
  config: { logger : function(x) { window.alert(x); } },

  showMsg : function() {
    this.config.logger(this.message);
  },

  init : function(config) {
    $.extend(this.config, config);
  },

  readConfig : function() {
    $.each(this.config, $.proxy(function(k, v) {
      this.config.logger(k + ': ' + v);
    }, this));
  }
};

console.log(myFeature.message); // logs 'hello'
myFeature.showMsg();           // alerts 'hello'
myFeature.message = 'goodbye'; // replace message
```

```
myFeature.init({                // replaces alert with console.log
  logger : console.log
});
myFeature.readConfig();          // logs config object
</script>
</body>
</html>
```

The `myFeature` function here behaves a little like a class with static methods. When the object literal is created, it has two properties and three methods. One of the properties, `message`, is a data value, the other, `config`, is itself an object containing configuration information.

Object configuration alteration is provided by the `init` method, which will copy any incoming configuration data into the `config` object. The utility method `readConfig` logs the state of the configuration object, and `showMsg` is the "useful" method, which simply writes the message to the current `logger`.

In our feature, there are no private elements. If we wanted any, we would wrap the feature definition in a self-executing anonymous function, and define the private elements within that function, but outside of the `myFeature` object literal. We would then need to store `myFeature` into a location outside the scope of the anonymous function, such as `window`.

The Module Pattern

The module pattern overcomes some of the limitations of the object literal, offering privacy for variables and functions while exposing a

public API if desired.

Lesson 1, Activity 8: **Don't Treat jQuery as a Black Box**

Use the source as your documentation; bookmark <http://bit.ly/jqsource> and refer to it often.