jQuery

jQuery is a popular library for manipulating the DOM and executing Ajax requests. There's nothing you can do with jQuery that you can't do with the DOM API (jQuery is, after all, built on the DOM API itself), but it offers three main advantages:

- jQuery protects you from idiosyncrasies in the way different browsers implement the DOM API (especially older browsers).
- jQuery provides a simpler Ajax API (which is welcome because Ajax is used heavily in today's websites).
- jQuery provides many powerful and compact enhancements on the built-in DOM API.

There is a growing community of web developers who feel that jQuery is no longer necessary with the improvements in the DOM API and browser quality. This community touts the performance and purity of "vanilla JavaScript." It's true that the first point (browser idiosyncrasies) is becoming less true over time, but it has not vanished altogether. I feel that jQuery remains relevant, and provides many features that would be extremely time-consuming to reimplement in the DOM API. Whether or not you choose to use jQuery, its ubiquity makes it hard to avoid entirely, and the wise web developer will know the basics.

The Almighty Dollar (Sign)

jQuery was one of the first libraries to take advantage of JavaScript's inclusion of the dollar sign as an identifier. Perhaps it was hubris when the decision was first made, but given jQuery's ubiquity today, the decision seems prescient. When you include

jQuery in your project, you can either use the variable `jQuery` or the much shorter `$`.[1] In this text, we'll use the `$` alias.

Including jQuery

The easiest way to include jQuery is by using a CDN:

```
<script src="//code.jquery.com/jquery-2.1.4.min.js"></script>
```

jQuery 2.x dropped support for Internet Explorer 6, 7, and 8. If you need to support those browsers, you will have to use jQuery 1.x. jQuery 2.x is considerably smaller and simpler because it doesn't need to support these aging browsers.

Waiting for the DOM to Load

The way a browser reads, interprets, and renders an HTML file is complicated, and many an unwary web developer has been caught off-guard by attempting to programmatically access DOM elements before the browser has had a chance to load them.

jQuery allows you to put your code in a callback that will only be invoked once the browser has fully loaded the page and the DOM has been constructed:

```
$(document).ready(function() {
    // code here is run after all HTML has been
    // loaded and the DOM is constructed
});
```

It is safe to use this technique multiple times, allowing you to place jQuery code in different places and still have it safely wait for the DOM to load. There is also a shortcut version that is equivalent:

```
$(function() {
    // code here is run after all HTML has been
    // loaded and the DOM is constructed
});
```

When using jQuery, putting all of your code in such a block is a ubiquitous practice.

1 There is a way to prevent jQuery from using `$` if it conflicts with another library; see jQuery.noConflict.

jQuery-Wrapped DOM Elements

The primary technique for manipulating the DOM with jQuery is using *jQuery-wrapped DOM elements*. Any DOM manipulation you do with jQuery starts with creating a jQuery object that's "wrapped around" a set of DOM elements (keep in mind the set could be empty or have only one element in it).

The jQuery function (`$` or `jQuery`) creates a jQuery-wrapped set of DOM elements (which we will simply be referring to as a "jQuery object" from here on out—just remember that a jQuery object holds a set of DOM elements!). The jQuery function is primarily called in one of two ways: with a CSS selector or with HTML.

Calling jQuery with a CSS selector returns a jQuery object matching that selector (similar to what's returned from `document.querySelectorAll`). For example, to get a jQuery object that matches all `<p>` tags, simply do:

```
const $paras = $('p');
$paras.length;              // number of paragraph tags matched
typeof $paras;              // "object"
$paras instanceof $;        // true
$paras instanceof jQuery;   // true
```

Calling jQuery with HTML, on the other hand, creates new DOM elements based on the HTML you provide (similar to what happens when you set an element's `innerHTML` property):

```
const $newPara = $('<p>Newly created paragraph...</p>');
```

You'll notice that, in both of these examples, the variable we assign the jQuery object to starts with a dollar sign. This is not necessary, but it is a good convention to follow. It allows you to quickly recognize variables that are jQuery objects.

Manipulating Elements

Now that we have some jQuery objects, what can we do with them? jQuery makes it very easy to add and remove content. The best way to work through these examples is to load our sample HTML in a browser, and execute these examples on the console. Be prepared to reload the file; we'll be wantonly removing, adding, and modifying content.

jQuery provides `text` and `html` methods that are rough equivalents of assigning to a DOM element's `textContent` and `innerHTML` properties. For example, to replace every paragraph with the same text:

```
$('p').text('ALL PARAGRAPHS REPLACED');
```

Likewise, we can use `html` to use HTML content:

```
$('p').html('<i>ALL</i> PARAGRAPHS REPLACED');
```

This brings us to an important point about jQuery: jQuery makes it very easy to operate on many elements at once. With the DOM API, `document.querySelectorAll()` will return multiple elements, but it's up to us to iterate through them and perform whatever operations we want to. jQuery handles all the iteration for you, and by default assumes that you want to perform actions on every element in the jQuery object. What if you only want to modify the third paragraph? jQuery provides a method `eq` that returns a new jQuery object containing a single element:

```
$('p')          // matches all paragraphs
    .eq(2)      // third paragraph (zero-based indices)
    .html('<i>THIRD</i> PARAGRAPH REPLACED');
```

To remove elements, simply call `remove` on a jQuery object. To remove all paragraphs:

```
$('p').remove();
```

This demonstrates another important paradigm in jQuery development: *chaining*. All jQuery methods return a jQuery object, which allows you to *chain* calls as we just did. Chaining allows for very powerful and compact manipulation of multiple elements.

jQuery provides many methods for adding new content. One of these methods is `append`, which simply appends the provided content to every element in the jQuery object. For example, if we wanted to add a footnote to every paragraph, we can do so very easily:

```
$('p')
    .append('<sup>*</sup>');
```

`append` adds a child to the matched elements; we can also insert siblings with `before` or `after`. Here is an example that adds `<hr>` elements before and after every paragraph:

```
$('p')
    .after('<hr>')
    .before('<hr>');
```

These insertion methods also include counterparts `appendTo`, `insertBefore`, and `insertAfter` that reverse the order of the insertion, which can be helpful in certain situations. For example:

```
$('<sup>*</sup>').appendTo('p');  // equivalent to $('p').append('<sup>*</sup>')
$('<hr>').insertBefore('p');      // equivalent to $('p').before('<hr>')
$('<hr>').insertAfter('p');       // equivalent to $('p').after('<hr>');
```

jQuery also makes it very easy to modify the styling of an element. You can add classes with `addClass`, remove classes with `removeClass`, or *toggle* classes with `toggleClass` (which will add the class if the element doesn't have it, and remove the

class if it does). You can also manipulate style directly with the `css` method. We'll also introduce the `:even` and `:odd` selectors, which allow you to select every other element. For example, if we wanted to make every other paragraph red, we could do the following:

```
$('p:odd').css('color', 'red');
```

Inside a jQuery chain, it's sometimes desirable to select a subset of the matched elements. We've already seen `eq`, which allows us to reduce the jQuery object to a single element, but we can also use `filter`, `not`, and `find` to modify the selected elements. `filter` reduces the set to elements that match the specified selector. For example, we can use `filter` in a chain to make every other paragraph red *after* we've modified each paragraph:

```
$('p')
    .after('<hr>')
    .append('<sup>*</sup>')
    .filter(':odd')
    .css('color', 'red');
```

`not` is essentially the inverse of `filter`. For example, if we want to add an `<hr>` after every paragraph, and then indent any paragraph that doesn't have the class `highlight`:

```
$('p')
    .after('<hr>')
    .not('.highlight')
    .css('margin-left', '20px');
```

Finally, `find` returns the set of *descendant* elements that match the given query (as opposed to `filter`, which filters the existing set). For example, if we wanted to add an `<hr>` before every paragraph and then increase the font size of elements with the class `code` (which, in our example, are descendants of the paragraphs):

```
$('p')
    .before('<hr>')
    .find('.code')
    .css('font-size', '30px');
```

Unwrapping jQuery Objects

If we need to "unwrap" a jQuery object (get access to the underlying DOM elements), we can do so with the `get` method. To get the second paragraph DOM element:

```
const para2 = $('p').get(1);      // second <p> (zero-based indices)
```

To get an array containing all paragraph DOM elements:

```
const paras = $('p').get();       // array of all <p> elements
```

Ajax

jQuery provides convenience methods that make Ajax calls easier. jQuery exposes an `ajax` method that allows sophisticated control over an Ajax call. It also provides convenience methods, `get` and `post`, that perform the most common types of Ajax calls. While these methods support callbacks, they also return promises, which is the recommended way to handle the server response. For example, we can use `get` to rewrite our `refreshServerInfo` example from before:

```
function refreshServerInfo() {
    const $serverInfo = $('.serverInfo');
    $.get('http://localhost:7070').then(
        // successful return
        function(data) {
            Object.keys(data).forEach(p => {
                $(`[data-replace="${p}"]`).text(data[p]);
            });
        },
        function(jqXHR, textStatus, err) {
            console.error(err);
            $serverInfo.addClass('error')
                .html('Error connecting to server.');
        }
    );
}
```

As you can see, we have significantly simplified our Ajax code by using jQuery.

Conclusion

The future of jQuery is unclear. Will improvements to JavaScript and browser APIs render jQuery obsolete? Will the "vanilla JavaScript" purists be vindicated? Only time will tell. I feel that jQuery remains useful and relevant, and will so for the foreseeable future. Certainly the usage of jQuery remains quite high, and any aspiring developer would do well to at least understand the basics.

If you want to learn more about jQuery, I recommend *jQuery: Novice to Ninja* by Earle Castledine and Craig Sharkie. The online jQuery documentation is also quite good.