Javascript Basics

DOM Interaction

Now that we've gone over most of the language fundamentals of Javascript, it's time to delve into the presentational aspects of it, which is why Javascript was developed in the first place. In this lesson, we're going to go over how to access elements in the DOM, and touch on events.

This lesson assumes you are familiar with HTML and how CSS selectors work. It also assumes that you know how to write Javascript functions and call functions on Javascript objects. If you're not comfortable with the Javascript portion of this session, check out the earlier session on functions.

I'll create an HTML page that has three elements in its body: a header element, and input element, and a script tag:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>DOM Interaction</title>
  </head>
  <body>
    <h1>Default Text</h1>
    <input type="text" />
    <script src="script.js"></script>
  </body>
</html>
```

I'm going to write a script that updates the text in the header based off what the user enters in the input.

First, I'll need to access the header element. There are a number of ways this can be done, but all of them involve methods on the document object. The document object is a property on the window object that interfaces with the web page's content. With it, we can access, create, and modify HTML elements.

To access the <h1> element's Javascript API, we're going to use the `querySelector` function. `querySelector` returns the first element that matches the specified selector text. In this case, to access the <h1> element, the selector is simply:

```
var header = document.querySelector('h1')
```

The selector string argument is usually identical to what you would enter if you were specifying a style in CSS. It's very similar to jQuery and Sizzle element selectors if you have experience with those.

For most projects, you'll be using a more specific query selector string to select an element. Otherwise, it could be harder to maintain your page. For example, you may have more than one header element on the page, and this query selector may select the wrong one.

To that end, we'll add an id to the <h1> element, and select based off the id instead.

```
<h1 id="header">Default Text</h1>
```

And:

```
var header = document.querySelector('#header')
```

When selecting an element based off id, you can also use the `getElementById` function. It returns the same object:

```
var header = document.querySelector('#header')
window.alert(header === document.getElementById('header'))
```

The object returned by the querySelector has lots of properties and functions, many of which overlap with the document object. You can get the attributes on the tag (`getAttribute()`), the immediate child elements (`.childNodes`), get HTML classes (`.classList`), query for inner elements (`.querySelector`), and more.

For now, we'll just change the text of the header to make sure the script gets called. To do that, we'll modify the `innerText` property:

```
var header = document.querySelector('#header')
header.innerText = 'Custom Header'
```

Now that we know that's working, let's wrap the code that sets the new text in a function:

```
var header = document.querySelector('#header')
function updateHeaderText (text) {
  header.innerText = text
}
updateHeaderText('Custom Header')
```

Now that the header can be updated via a function call, let's select the input element, and have the header update from that. As with the header element, it's generally good practice to make a specific reference to the input element in the HTML, so add an id to the element:

```
  <body>
    <h1 id="header">Default Text</h1>
    <input id="input" type="text" />
    <script src="script.js"></script>
  </body>
```

Now select the element:

```
var header = document.querySelector('#header')
var input = document.querySelector('#input')

function updateHeaderText (text) {
  header.innerText = text
}
updateHeaderText('Custom Header')
```

I personally like grouping any code that accesses elements in the DOM together. It makes maintenance easier.

Now that we have a reference to the input element, we need to respond to updates. Many elements in HTML emit events when a user does something, such as presses a key or clicks a button. The <input> element in particular has a myriad of events.

The simplest way to listen for and respond to an event on an element is to override a property on the element. There are lots of limitations to this approach, but it's more than enough for our purposes. Here's how to do it:

```
input.onchange = function (event) {
  console.log('Modified')
}
```

Whenever the input element changes (loses focus), the function will be called, and a console message will be displayed.

We're introducing an important concept in programming in Javascript: asynchrony. Javascript is (usually) single-threaded, which means it is not capable of running code in parallel. However, it can run sections of code at a later time, typically when responding to an event. An asynchronous function is handled once no other code is running; Javascript doesn't pause execution to handle events.

The function that is called takes an argument. This argument is a reference to the "Event" object. What is in the Event object varies depending on what event is being executed. For example, a mouse event will include information about the mouse position, and a keyboard event will include events about the key. The `target` property on the Event object is a reference to the HTML element that dispatched the event. We can use this property to get the input element's text:

```
input.onchange = function (event) {
  console.log(event.target.value)
}
```

Another option is to use the `this` keyword. In this case, the current scope is the input element:

```
input.onchange = function (event) {
  console.log(this.value)
}
```

We'll use `this` because it's a little shorter.

With the value of the updated input element, we can call the function that updates the header text:

```
var header = document.querySelector('#header')
var input = document.querySelector('#input')

function updateHeaderText (text) {
  header.innerText = text
}
updateHeaderText('Custom Header')

input.onchange = function (event) {
  updateHeaderText(this.value)
}
```

There are a lot of interesting complexities with Javascript events, such as how to handle events that occur on multiple elements at once. A divider element and an inner divider element might respond to a `mousemove` event in the same instant for example. We won't cover event bubbling, `event.preventDefault()`, or `event.stopPropagation()` in this session, but you may want to research those topics on your own.

That said, I do want to show another way to add event handlers to elements: the `addEventListener` function:

```
input.addEventListener('change', function (event) {
  updateHeaderText(this.value)
}, true)
```

Here's the main differences:

- You can add multiple event listeners to the same event with this method. That said, the only way to remove the event handler is to use `removeEventListener`.
- The name of the event is specified as a string, and does not begin with "on".
- There's an additional argument, "capture", which is a Boolean that specifies that the event should be handled before any events underneath it.

For those of you who haven't programmed in Javascript before this tutorial, I can understand that all of this might seem daunting. There's a lot of things to memorize here. Here's a quick recap of what we've gone over:

- `document.querySelector()` to access an element
- `element.innerText` to get/change the text of a presentation element
- `element.value` to get/change the value of an input element
- `element.onchange` to respond to updating an input element
- `element.addEventListener('change', function () {}, true)` same as above

There's one more thing I wanted to cover. `querySelector`, as well as any function on document that accesses elements, is returning the current state of the Document Object Model. If this script is called too soon, the HTML elements won't exist, and the document functions will return null. Because the script is defined in the HTML at the bottom of the body tag, we know the elements will exist once it is invoked, but if you don't know that will be the case, you'll want to put all the code that accesses DOM elements in a function that is invoked once the DOM content is ready or the window is loaded, depending on what you need.

Here's how you could do that:

```
function onLoad () {
  var header = document.querySelector('#header')
  var input = document.querySelector('#input')

  function updateHeaderText (text) {
    header.innerText = text
  }
  updateHeaderText('Custom Header')

  input.addEventListener('change', function (event) {
    updateHeaderText(this.value)
  }, true)
}

if (document.readyState === 'complete') {
```

```
  onLoad()
} else {
  window.addEventListener('DOMContentLoaded', onLoad, true)
}
```

The window object, like the document object, also has an `addEventListener` function. The `'DOMContentLoaded'` event fires when the HTML has been parsed, but before any external resources like stylesheets or images load. There's also the `load` event that fires once EVERYTHING loads. Depending on your use case you might want to use that instead.

The `document.readyState` property updates as the page loads, and has three possible values: `loading`, `interactive` (when the DOM content is ready but images/stylesheets are still loading) and `complete` (when everything has loaded).

I include the check because if for some reason the DOM content is loaded *before* the script executes, the `DOMContentLoaded` event will never fire.