Lesson Weekend

Workshops (/workshops) / Intro to Programming Workshop (/workshops/intro-to-programming-workshop) / Interactivity with JavaScript & jQuery

Text

JavaScript and the web

We don't spend a lot of time dealing with alert boxes on the web. Ideally we want as much of our functionality to happen on the page as possible. To do that we need to have our JavaScript logic - the same logic that did our addition, multiplication and concatenation - see the information our HTML is dealing with.

To make this communication between HTML and JavaScript possible, we're going to use a JavaScript **library**. A library in JavaScript is just like a CSS library like Bootstrap. It's a collection of already existing code that can help speed up our process.

We'll use a library called **jQuery**. The job that jQuery does is to look at our logic and look at our HTML and let them interact. Let's see how this works.

First we need to alter our HTML head again to include the library. We'll use a URL to reference it, just like we did when we used a URL to reference our image and our Bootstrap styling.

index.html

```
...
<head>
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
  <link rel="stylesheet" href="css/styles.css">
  <script src="https://code.jquery.com/jquery-2.2.4.min.js"></script>
  <script src="js/scripts.js"></script>
  <title>Epicodus Workshop Project</title>
</head>
...
```

Note that we've listed this new `<script>` *above* the script we're writing ourselves. When these JavaScript files load, they do so from the top down. Since the code we're about to write depends on the jQuery library, we need to make sure jQuery loads first.

Let's change the code in our *scripts.js* file to match the following code.

js/scripts.js

```
$(document).ready(function() {
  alert("jQuery is working");
})
```

There's a lot happening here that we haven't seen before, so let's explain.

- `$(document).ready(function() {` on the first line is pure jQuery. Its job is to wait until the document is finished loading, or `ready()`. Remember how we learned that JavaScript code executes as soon as the script is loaded? This is how we tell JavaScript to wait for the rest of the code to load before doing anything. Within the `ready()` function, that new `function() {` opens a new function, where we get to define what happens.
- `alert("jQuery is working");` is what we want to happen within this new function, once the page loads.
- `})` on the last line shows where the things we want to happen when the page loads end. Just like a CSS style, this curly-brace/parens combo closes the `ready()` function and the area where we define those actions. An area of actions within a function is called that function's **scope**. The scope of our jQuery code contains only the single alert function.

Now let's get this code to talk to our page. Remember how we used **classes** and **ids** to target different parts of our HTML for our CSS to apply styles? We use those same classes and ids to target parts of our HTML for jQuery to interact with. Let's

change our function to the following:

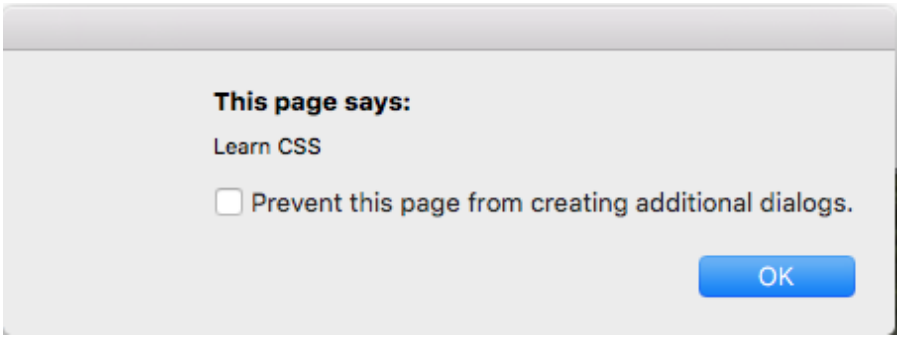
js/scripts.js

```
$(document).ready(function() {
  alert($("#css-item").text());
})
```

Look complicated? Let's look at only what's inside the parens of the `alert()` function.

```
$("#css-item").text()
```

When we started using jQuery and `$(document)`, we were telling jQuery to wait for the entire webpage to be ready (that's what `document` refers to). This time we only want to target a specific part of the page. We say `$("#css-item")`. Just like in our CSS, we use the `#` to indicate an `id`. Then, once we've targeted the part of the HTML we want to interact with, we can do something with it. In this case we're calling a new jQuery method called `text()`. The job of this function is to report the text contents of an element. So in simple terms, when we load our page, the text of the element with the id `css-item` is populated into our `alert()`. Pretty neat!



Try to add different ids to other elements within the HTML, to populate them into the alert box instead. Try adding an id to our `<p>` element, or a header element.

Adding to our list.

Let's create one last, bigger piece of functionality today. Let's create a form that allows us to add new items to our list of programming goals. We'll add this form to our *index.html* file, just above the `<ul>` that contains our list of goals.

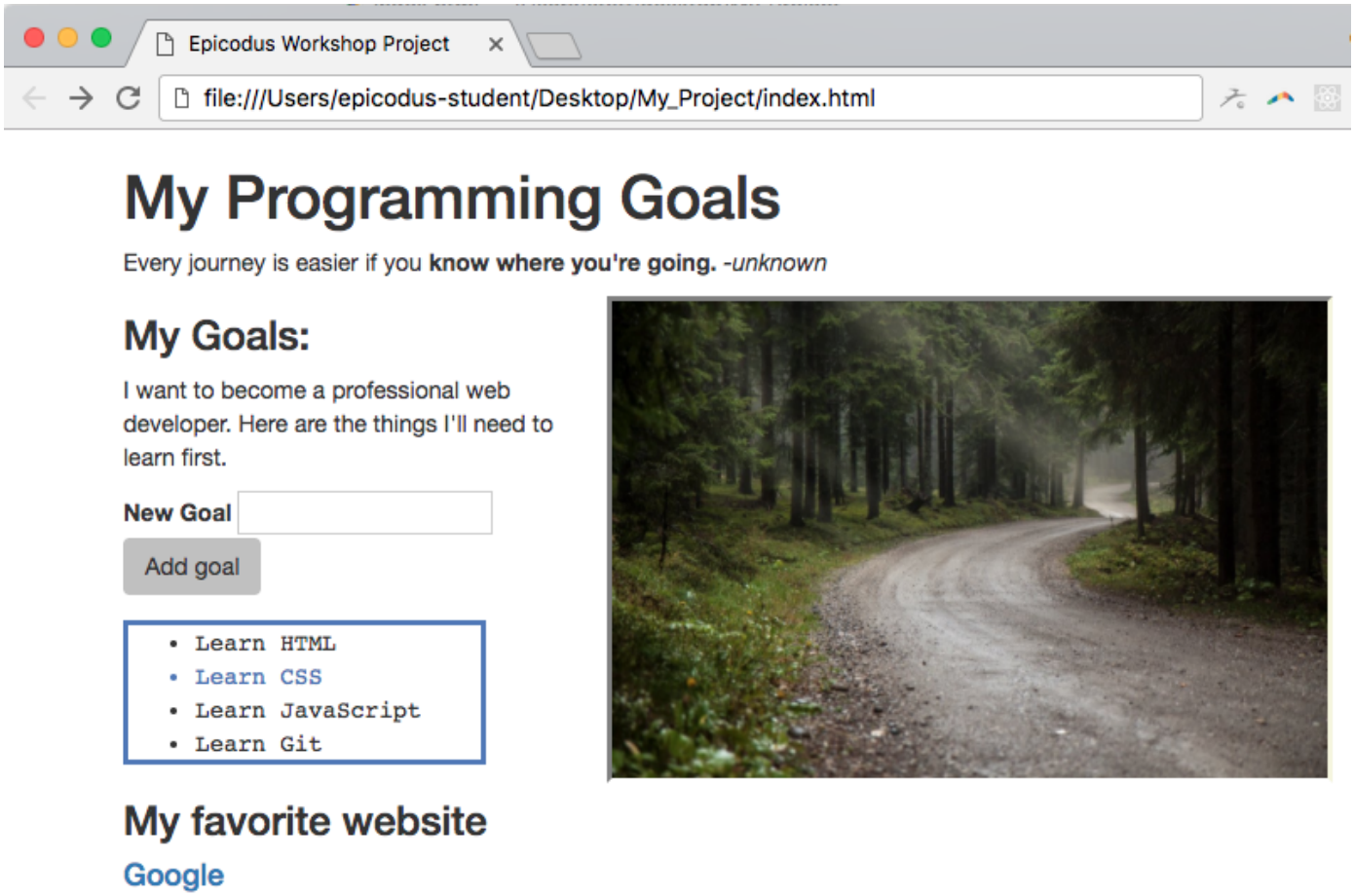
index.html

```
...
  <div class="form-group">
    <label>New Goal</label>
    <input type="text" id="new-goal">
    <button class="btn" type="button" id="goal-button">Add goal</butto
n>
  </div>
...
```

Notice how these HTML elements are indented to keep them organized? Good coding includes using good organization and conventions to help make these items easier to read.

Our new form contains a label, a text input field for the user, and a button to click when the form has been filled out. Note the `id` given to the text input. That's about to become important.

If we reload our page now, we see our form, and we can enter a new goal into it.



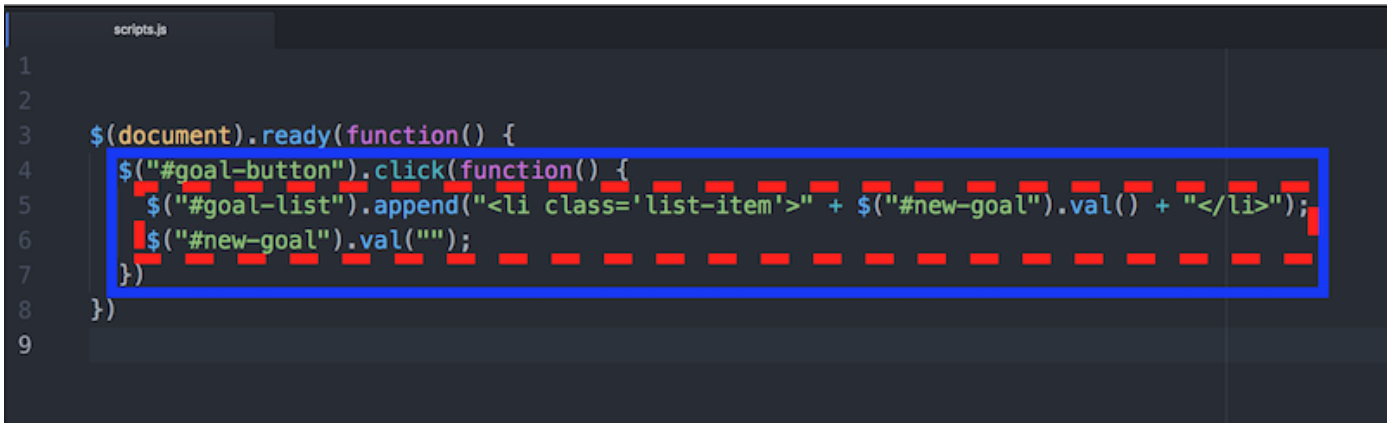
When we click the button however, nothing happens. Let's use jQuery in our *scripts.js* file to change that.

In our *scripts.js* file, let's replace our alert with a new kind of function. Here's what we want our file to look like.

js/scripts.js

```
$(document).ready(function() {
  $("#goal-button").click(function() {
    $("#goal-list").append("<li class='list-item'>" + $("#new-goal").v
al() + "</li>");
    $("#new-goal").val("");
  })
})
```

Let's look at these new lines together, and we'll see that it's not all that complicated.



- First off, notice that the majority of code (everything in the solid blue rectangle above) is nested under the `$(document).ready(function() { ...` line. As discussed earlier, code that resides within this must wait until the webpage is fully loaded before it runs.
- Secondly, notice the `$("#goal-button").click(function() {` line. Similar to the way `$(document).ready(function() { ...` forces all code within *its* block to wait until the page is loaded to run, this line instructs code nested within *it* to wait until a specific item is clicked to run. In our case, it's waiting for our button to be clicked. We know this because the line reads `$("#goal-button").click(function() {`, and our button has an `id` of `goal-button`. This is called a **callback function** because it is called-back-to when the event it listens for takes place. So, **code within the red dotted rectangle only runs when our `goal-button` is clicked.**
- When this button is clicked, the code nested within the `$("#goal-button").click(function() {` will run. Let's look at each of these two lines individually:

- `$("#goal-list").append("<li class='list-item'>" + $("#new-goal").val() + "</li>");` is composed of two primary actions:
    - `$("#goal-list").append()` tells jQuery we have something we want to **append** to the `#goal-list` element. To append is simply to add to the end.
    - `"<li class='list-item'>" + $("#new-goal").val() + "</li>"` See that we're using concatenation here? We've used our knowledge of HTML to add `<li>` tags to create a new list item for our HTML list.
    - What about in the middle of that expression? `$("#new-goal").val()`? This function, `val()` refers to the **value** of the `#new-goal` element. Happily, the value is... whatever the user has typed into our form field!
- `$("#new-goal").val("");` The last line resets the input form to blank. Blank is signaled by the `""`. An empty string.

To review, as soon as we visit our page, the following events occur in this order:

1. The `$(document).ready(function(){...` line forces all code within it to wait until the page completes loading.
2. Once the page is finished loading, the `$("#goal-button").click(function(){....` line begins actively listening for a user to `click` on the button with an `id` of `goal-button`.
3. When (and *only* when) a user clicks the button is the `$("#goal-button").click(function(){....` click listener activated:

index.html

```
<div class="form-group">
  <label>New Goal</label>
  <input type="text" id="new-goal">
  <button class="btn" type="button" id="goal-button">Add goal</button>
</div>

<ul id="goal-list">
  <li class="list-item">Learn HTML</li>
  <li class="list-item" id="css-item">Learn CSS</li>
  <li class="list-item">Learn JavaScript</li>
  <li class="list-item">Learn Git</li>
</ul>
```

scripts.js

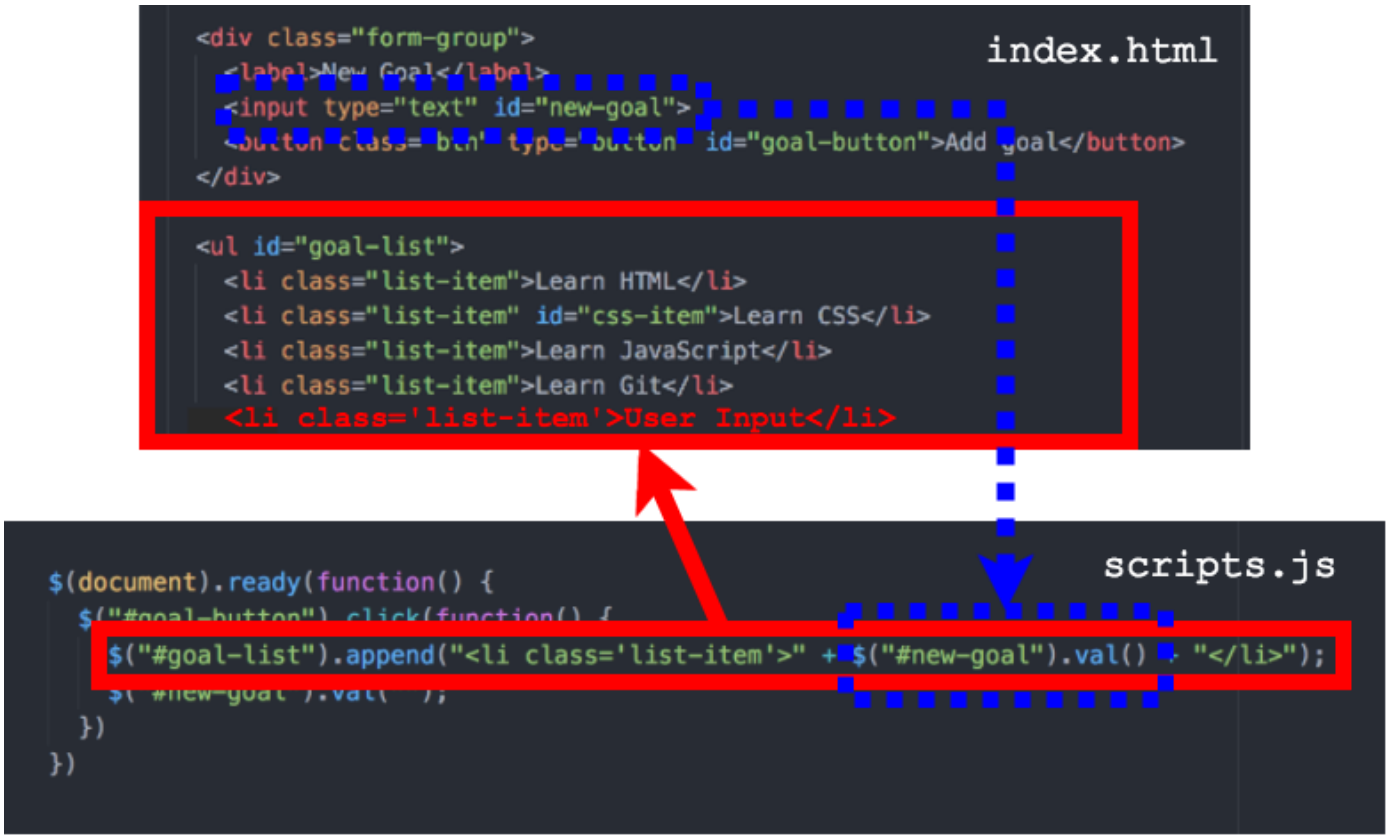
```
$(document).ready(function() {
  $("#goal-button").click(function() {
    $("#goal-list").append("<li class='list-item'>" + $("#new-goal").val() + "</li>");
    $("#new-goal").val("");
  })
})
```

1. Once activated, the code within it runs:

scripts.js

```
$(document).ready(function() {
  $("#goal-button").click(function() {
    $("#goal-list").append("<li class='list-item'>" + $("#new-goal").val() + "</li>");
    $("#new-goal").val("");
  })
})
```

1. The first line within the click listener block *appends* new content to the HTML element on our page with the `goal-list` id. It adds both the HTML `<li class='list-item'>`, whatever *custom* content the user typed into the form field with the `new-goal` id, then a closing `</li>` tag. In that order. The user's form content is retrieved by calling `$("#new-goal").val()`:



1. The second line, `$("#new-goal").val("");` simply empties the form field, so the user may enter another goal into the form, if they'd like.
2. Every time the user clicks the button, steps 3 through 5 occur again.

Now that we have an introductory understanding of what's going on, let's try it out. Save your JavaScript file, and reload your page.

Can you add new elements to your list? If so, congratulations! You've built your first interactive JavaScript web application!

In the next and final lesson, we'll learn how to save our work, and host it on GitHub, an industry standard cloud storage solution.

A Note on JavaScript & jQuery

Also, before we move on, we'd like to mention that jumping into JavaScript and jQuery is *tough*. Don't worry if you don't fully understand everything we just completed. That's entirely normal! Our full-time Intro to Programming students often spend about a week learning and practicing before they report having a decent grasp on these concepts. If you're interested in further developing your JavaScript skills after today, we'll provide a list of resources in the last lesson of this workshop.