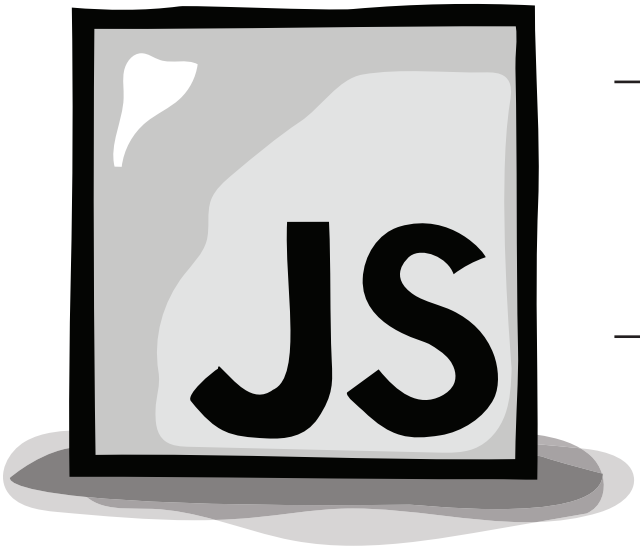


GYMNASIUM

JAVASCRIPT FOUNDATIONS

Lesson 4 Handout

Creating Elements, Fragments, and Templates

ABOUT THIS HANDOUT

This handout includes the following:

- A list of the core concepts covered in this lesson.
- The assignment(s) for this lesson.
- A list of readings and resources for this lesson including books, articles and websites mentioned in the videos by the instructor, plus bonus readings and resources hand-picked by the instructor.
- A transcript of the lecture videos for this lesson

CORE CONCEPTS

1. You can create new HTML element objects using document.createElement, passing in the name of the element you want to create.
2. Elements created this way are not automatically put on the DOM and have no textContent or innerHTML.
3. Add elements to the DOM using the appendChild and insertBefore methods.
4. If an element is already on the DOM, using appendChild or insertBefore will remove it from its current location.
5. Adding and removing elements or changing styles or properties of elements on the DOM triggers a document reflow, where the entire HTML document is re-rendered.
6. Create complex bits of HTML in a document fragment, then add the fragment to the DOM for a single reflow.
7. Templating engines are a way to pre-write chunks of HTML that can be filled in with dynamic data at run time.

ASSIGNMENTS

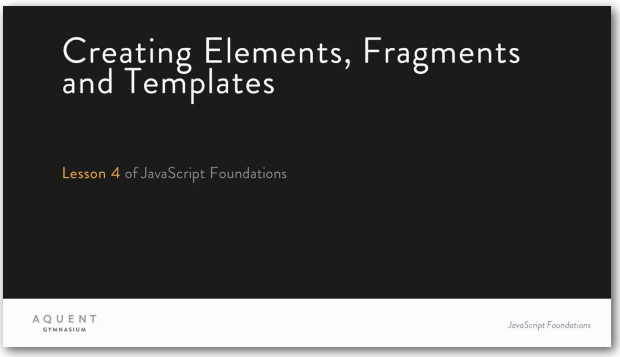
1. Quiz
2. Go to a site of your choice. Open the console and practice creating elements, and adding them to the page using appendChild. Try locating specific points within the DOM to add them using insertBefore. Practice removing the elements you added, or removing existing elements and re-adding them at different positions. Do this until you feel comfortable creating, adding and removing elements?
   Check out the documentation for Mustache.js at this site: https://github.com/janl/mustache.js
   Try some of the more advanced techniques, such as using an array of data to make multiple items, or any other techniques that look interesting or useful.
3. Take a look at Handlebars.js, http://handlebarsjs.com/ and compare and contrast it to Mustache.
   Try converting the mustache demo project from earlier in this lesson, or the final project over to use Handlebars instead of Mustache. Are there any advantages?

RESOURCES

- In addition to the Mustache and Handlebars documentation, you may want to do some research on other templating engines and see what they have to offer.

INTRODUCTION

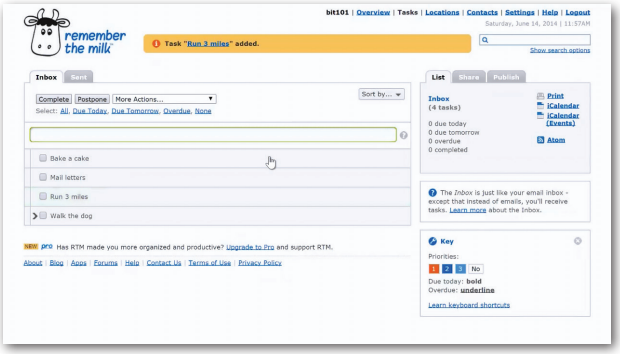
(Note: This is an edited transcript of the Javascript Foundations lecture videos. Some students work better with written material than by watching videos alone, so we're offering this to you as an optional, helpful resource. Some elements of the instruction, like live coding, can't be recreated in a document like this one.)



In Lesson Three, you learned how to manipulate the DOM, or document object model. You now know how to select any element on an HTML page, change its text content or innerHTML, sit and read other properties, and apply any CSS style to that element, all through JavaScript. This is all very powerful, as it allows you to dynamically change the look, and feel of a page, and even the content of that page, to some degree, in response to some user actions, or any other dynamic input.

In this lesson, we'll take that a step further, creating brand new HTML elements, and adding them to a page, removing existing elements from a page, and even moving elements around to rearrange a page. We'll also look at document fragments, which allow you to build up a portion of a page, storing it in memory, and then adding that whole fragment to the page, when you're ready.

This leads into a discussion of templates. A template is a reusable chunk of HTML, in a format that allows you to fill in the blanks with custom data at runtime. The template is filled in with that data, and the resulting HTML is then, added to that page. We'll end up creating an application that uses this form to collect data from a user, and then, uses a templating system to generate this page, based on that data. It even allows the user to generate the page in a number of different styles.



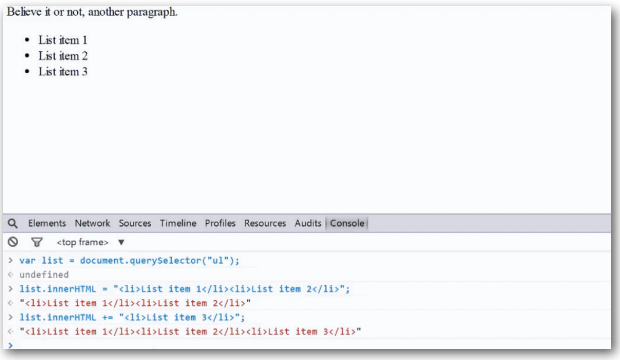
But, let's start off by creating some elements. Now, why would you want to create elements at runtime? Well, one example might be a to-do list, like the popular Remember the Milk site. Here, I can enter a to-do item, and it gets created as a new item, along with a check box. This is all done with JavaScript.

Somewhere there's a function that gets called when you've entered some text and hit the Enter key. That function creates some element objects, fills them with the text you entered and adds those elements to the page. So, creating new elements with JavaScript can be very useful.

First, let's review what we did in lesson three and create some elements, using innnerHTML. We'll see some of the problems with using that method, and how we can overcome them by creating element objects directly.

So find that DOM HTML file we used in lesson three, and open it up in Chrome. There's an unordered list in there, with a few list items. Let's select that list first. We say list = document.querySelector("ul"). Remember that querySelector returns a single element, so that list element is now stored in the variable list.

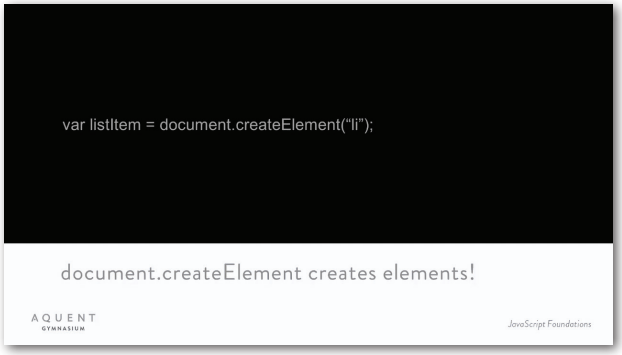
Earlier, we used innerHTML to add some new list items to that list, after we'd emptied it out, something like this. So, that created two new list items, but it also wiped out the ones that were in there, originally. That's kind of cool, but it's very limited. What if you just wanted to add another item to the list? Well, you could append the HTML for a new item onto the existing innerHTML, like this. Since innerHTML returns a string, we can concatenate a new string with a third list item to those items.



But, what if you wanted to insert an item between one and two? Now, things get a bit more complex. You'd have to get two substrings from that innerHTML; the first part containing the first list item, the second part containing everything after that. Which means you'd have to know the index where those two list items separate, then form a new string with the first item, plus your new item, plus the rest of the items. Then assign that string back to the innerHTML. That's a lot of string operations. One small mistake, and your HTML won't work at all.

To make a long story short, manipulating the DOM with all kinds of fancy string manipulation and innerHTML can get very complex, and error-prone. A much better solution for anything, beyond very simple examples, is to create a new element object in JavaScript and add that to the page. You create new element objects with document.createElement.

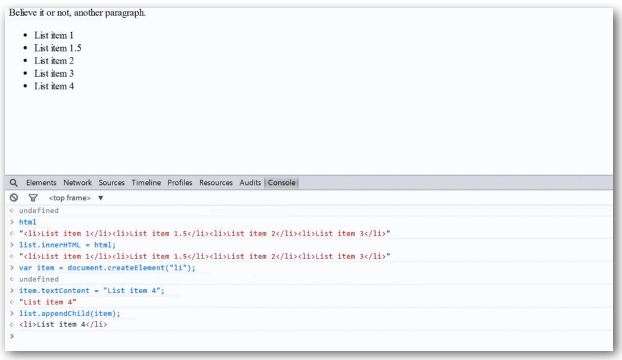
Just pass in a string with the name of the type of the element you want to create. Note that, you simply type the name of the element. Don't use angle brackets, like so. This may actually work in some browsers, but it's not standard, and won't work everywhere.



When you use createElement, you get back a JavaScript element object, just as if you'd used one of the selection methods, such as getElementByID. But, there are two big differences. One is that this element has not yet been added to the HTML page. Simply creating this new element does not alter the document in any way. Nothing has been changed visibly on that page, in the browser. Shortly, you'll see how you can add that element to the document.

The other difference is that this is just a plain, empty element with no attributes, text content, innerHTML, or anything else. It's just as if you'd typed li in an HTML document. So, minimally, if you want to see something on the page, we should give it some text content. Let's go back to the browser and console, and actually create that list item. Now, we can say, item textContent equals list item four. Now, we can add that element to the list.

An easy way to accomplish that is with the appendChild method. We want to append the list item to the list object we have stored in the list variable. So, we say, list appendChild item. And, now that list has a new item. But, that's not really much more powerful than concatenating an innerHTML string. Let's see if we can insert an element between two others.

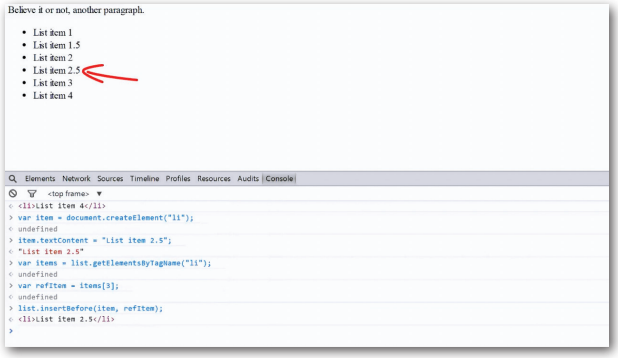


First, let's create a new list item. We'll set the textContent to list item 2.5. So, obviously, we want to put this between items two and three. The tool for this job is the insertBefore method. This method gets called from the parent element that you want to add the new element to.

So, in our example, we'll be saying, list insertBefore. And, the insertBefore method takes two parameters. The first one is easy. It's the element, "did you want to insert"? The second parameter is called, the "reference element." This answers the question, before what?

In our case, we want to insert this new list item before list item three. So, we need to get that element, and pass it in as a parameter, here. But, how do we get it? Well, there are a few ways. One is with the selection methods you already know.

Up to now, we've been calling those methods from the document itself, but most of them can be called on a single element, as well. So, you can say, list getElementsByTagName li, and get only the list item elements that exist as children, of that list element. Remember that, this method will return an array of list item element objects.

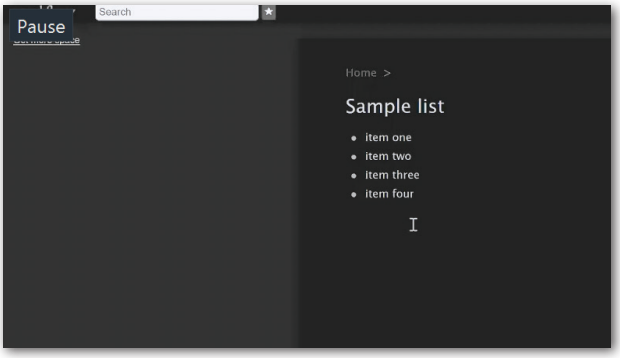


The list item with the text, "list item three" will currently be at index three. So, we can call that, "ref item", as it's the reference item we want to insert before. So, we say, ref item equals items index three. Now, we can insert that new list item before ref item. List insertBefore item comma ref item. And, there it is, item 2.5, right where we wanted it.

So, now you know how to create elements, and add them to a page. No longer is the creation of HTML something that happens just in an editor, before the site goes live. Now, it can happen right in the browser runtime. You'll see plenty, more of this, as we move through the lesson.

But, next up, we'll look at removing, and repositioning elements.

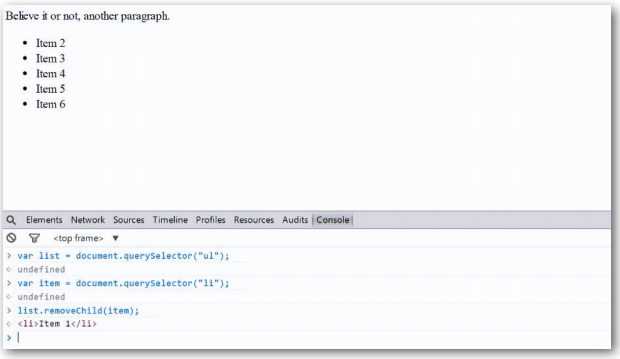
So far, in this lesson, you've seen how to create new HTML elements, and add them to a page. This can make for a very dynamic website. Another part of that dynamism would be removing elements, or even moving them around on a page. As an example of removing and rearranging objects, let's look at another popular site, WorkFlowy.com. This site allows you to create text outlines.



Here, I've created an outline with a few sample items. These are obviously all represented by some internal HTML elements, most likely list items. I can use this context menu to remove one of those items, and that element is removed from the page. Additionally, I can use drag and drop to rearrange the elements on the page. Internally, you can bet that the HTML itself is being altered to reflect the change in item order.

So, let's see how to do this. We've added elements to a document using the append a child, and insert before methods. You might guess that to remove elements, there's a removeChild method, and there is. This method must be called on the parent element that contains the child element you want to remove. So, if you have an unordered list, for example, in a variable called list, and an item stored in a variable called item, you could say, list removeChild item.

Well, that seems simple enough. Let's try it on a live page. Open up our good, old DOM HTML file in a browser, and bring up the console. If you've still got it open from the last video, refresh it so you're back to the original HTML, without all the changes we made. Let's get a reference to that list using queryselector ul and a reference to the first item, with queryselector li. Then, we could say, list removeChild item. And, just like that, that first item is gone.



Now, as simple as that is, there are a few important things to be aware of when using this. First is, as I mentioned, you must call this method on the parent of the element you want to remove. If the child element is not actually a child of the element you're calling it on, not only will the function fail, but it will result in an error. Let's get a reference to the first h1 element on the page by saying query selector h1.

Now, let's try calling list removeChild heading. As you can see, we get an error, because that h1 element is not a child of that list. So, it's always best to check to see if the child

element actually is a child of the parent, before trying to remove it from that parent. You can do so by doing something, like this. The parent element property, as you can guess, points to the actual parent of that child element. You can check to make sure that it is the parent you think it is, before calling removeChild.

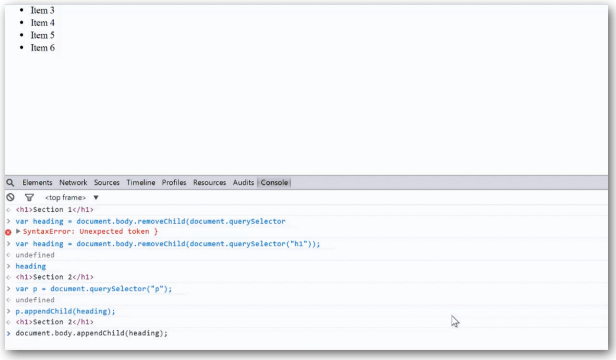
You can use this another way, too. Say, you want to remove an element from the page. You don't know where it is in the structure of that page, and you don't really know, or care, what its parent is. You just want it gone.

For example, let's get rid of that first h1 element that we just stored in the heading variable. Just say, heading parent element removeChild heading. The heading's parent happens to be the document body, but we didn't have to know that, or reference it. We basically just said, whatever your parent is, tell it to remove you. JavaScript can be cold.

Another thing to know is that, the removeChild method will return the element that was removed. You can use this to save a line or two of code, sometimes. With the following line of code, you can both select, and remove the remaining h1 tag. Say, heading equals document body removeChild document queryselector h1. The queryselector selects the H1 element and passes it to removeChild, which removes it, and returns it, so that it can be stored in the heading variable.

So, notice that the section two heading has disappeared, but when I type heading, it's safely stored right there, in that variable. Now, I can select the first paragraph element and say, p-append child heading, and that h1 element is now a child of the first paragraph.

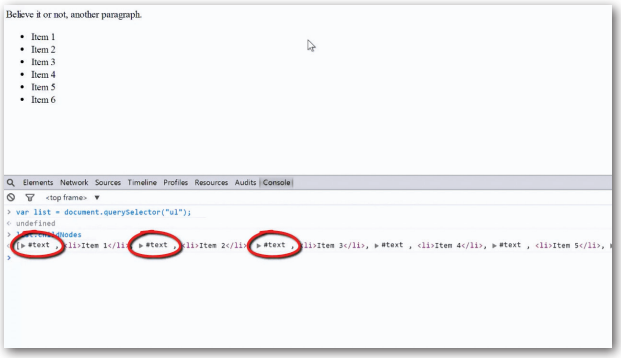
Now, I'm not saying you ever should or would move an h1 tag inside of a paragraph, but you can see how you can start rearranging a document however you want at run time. In fact, you don't even need to remove an element in order to change its position.

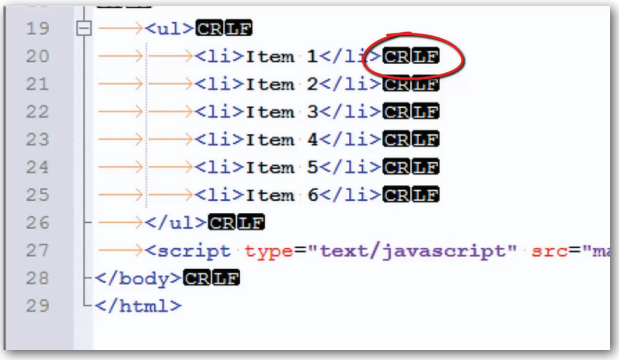


If we now say, document body append child heading, it's automatically removed from whatever parent it had before. In this case, that first paragraph, and appended to the new parent, the body, in this case. So, adding, removing, and repositioning elements is really pretty basic. But, as you'll see in the next couple of videos in this lesson, you can use these functions to create some pretty complex, dynamic behavior.

Now, before we move on, I want to show you a few more ways of accessing, and referencing various child elements. Let's reload this page to clear out all the changes we made to it. Then, select that list. Now, any element will have a property called child nodes, which is a list of all its children. So, we should be able to see all the child list items of this list. Well, we see them, but we also see some other things in there, too. In between each list item, there's some text. What's that all about?

Well, if you look back to the source HTML document, you'll see that each list item is on its own line, meaning that it's followed by a new line character. This character will vary on different systems, but here, I've opened this file up in an editor that will show all the characters, and you can see that I have a carriage return, and a line feed character at the end of each line.

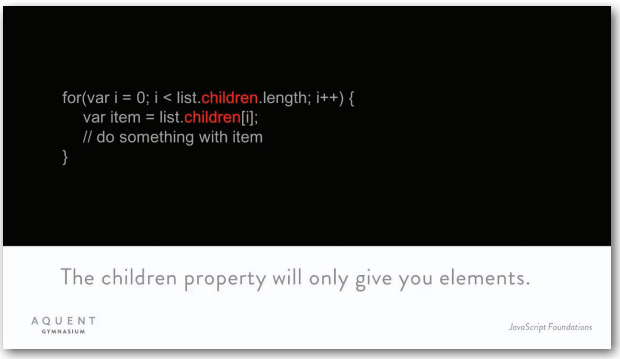




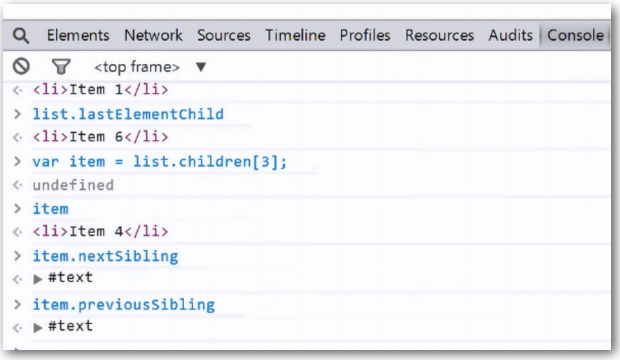
Many lines are also indented a bit with tabs, and here, you can clearly see all those tab characters. That extra text gets preserved by the document object model in what are called, text nodes. Of course, it's ignored when the HTML is rendered, but it's there, nevertheless. In fact, back in the console, we can expand that text note item, and look at its text content, and see the return character, plus the white space, mystery solved.

But, that text can cause some problems. Say, you wanted to use a for loop to run through all those list items, something like this. Well, in this case, every other item would actually be a text node, not the list items you were hoping for. Normally, you don't really care about these line feeds, and tabs, and other various white space between elements. You just want the elements.

Luckily, there is another property called, children. So, we can say, list children, and as you can see, this returns only the list items, which is most likely all you really want. So, you'd be much better off using a for loop with the children property, like this.



Now, in addition to the full array of children, you'll find it's often useful to know the first, and last child of a parent element. So, there are properties for those, first child, and last child. Here, in the console, we'll see what those give us. Uh oh, now, we're back to those pesky text nodes.



Luckily, like child nodes and children, there are alternatives here as well. To hit the first and last element children, use firstElementChild and lastElementChild, much better. Now, let's get a reference to one of those children using list children index three. This will give us the fourth child, labeled, item four.

In keeping with the familial theme, children not only have parents, they have siblings. Yes, that's actually what they're called. So, now, we can say, item nextSibling, and item previousSibling. And, oh no, it's text again. But, you know where this is going. You probably already saw it pop up there, and you're typing it in, already. Item nex-

tElementSibling, and item previousElementSibling. It's also worth mentioning that, there are also parent node, and parent element properties as well.

We already used parent element, earlier in this video. So, along with the various selection methods we covered in the previous lesson, knowing how to access parents, children, and siblings can have you moving around the document with ease.

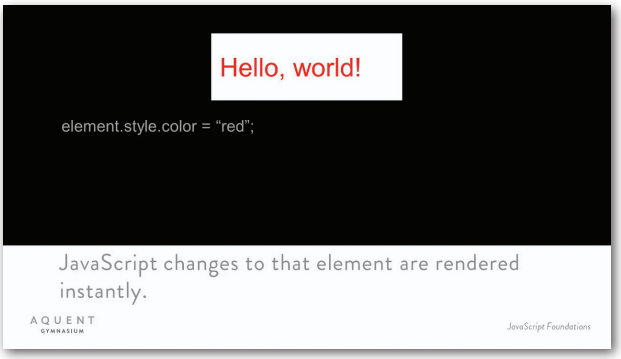
Now, whenever we wind up with multiple methods of doing the same, or similar things, there's the obvious question, which one should you use? And, as usual, the answer is, it depends. In general, you'll probably want to use the methods that avoid text nodes, unless you have some reason to access those, as well. But, beyond that, use whatever works for you.

In general, I'd say that using the children array would probably be faster than the selector methods, which are internally a bit more complex. But, the selector methods give you the ability to be more choosy about the type of results you get, whereas children will return all the children, no matter what type they are.

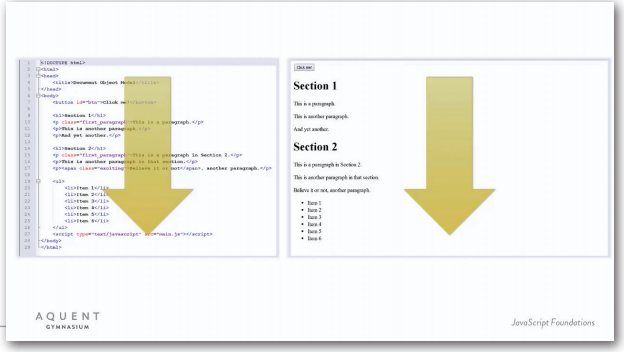
Next up, we'll look at document fragments, which will allow you to manipulate not only single elements, but entire chunks of HTML as a single entity.

In this video, I'm going to introduce you to document fragments, and discuss why you may want to consider using them. First, we'll look at what happens when you alter an HTML element on a page, using JavaScript.

As I pointed out previously, every time you change a property of an element, or change the style of an element, using JavaScript, that element is instantly re-rendered, and displayed with a new property, or style. And, you've also, no doubt, noticed in this lesson that every time you add, or remove, or reposition an element, that change is reflected immediately, as well.



On the surface, this might seem like a positive feature. And, in many cases, it is. There's no need to wait around for the page to update, no need to explicitly trigger an update. Just make your changes, and know that the page instantly reflects what you did. But, this instantaneous updating can also have a downside. To see what I mean, let's consider how HTML layout works.



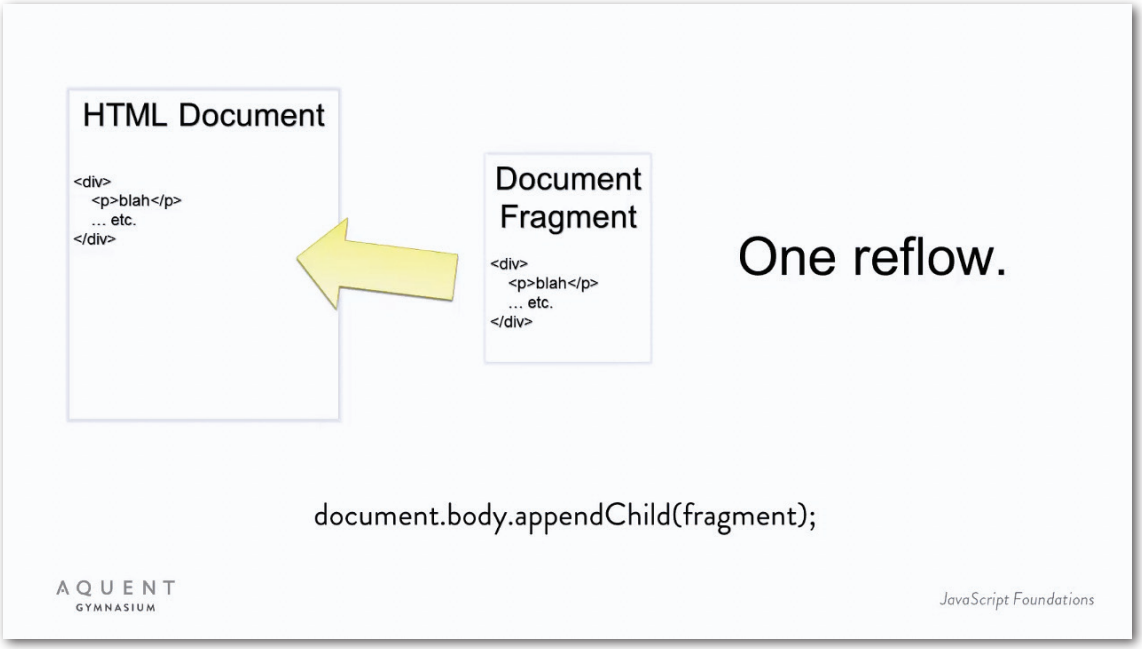
Elements are not absolutely positioned by default. Each element's position is based on where it's located in the document, that element's content, what styles are applied to it, the positions of the various elements surrounding it, the width of the page, et cetera. And, that's stating it very simply. The browser reads in the HTML, examines all that structure, applies any styles, and renders the page.

After that initial rendering, any change to any element can potentially affect the layout of the entire page. Simply changing the font weight of a span to bold will likely make it a bit wider, pushing the other text to the right. This may cause a slightly different word wrap, which can change the height of a paragraph, which can push other elements all over the place. To make a long story, short, any property change you make to an element that results in a visual change, (any style change, and of course, adding or removing or repositioning elements) will cause the browser to lay out the page, anew. We say that, the change triggers a reflow.

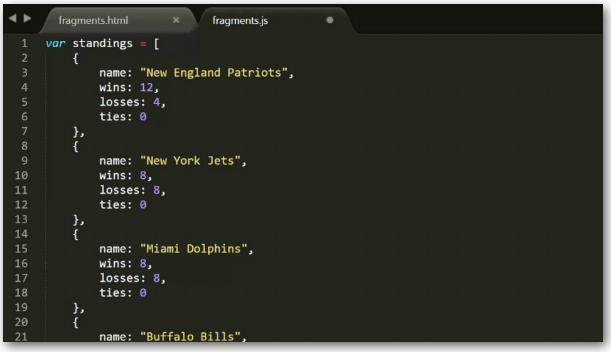
When a reflow happens, the browser needs to re-read the entire HTML document, apply all the CSS, and go through the whole process of working out how every element looks, and where it should be drawn. From your viewpoint, you are just straightening a crooked picture on the wall. From the browser's viewpoint, it's tearing down the entire house, and rebuilding it from scratch.

This can have a serious performance impact on a page. If you are creating, and adding, and changing properties of a lot of elements on a complex page, all those reflows can really slow your program down to the point where it's visibly lagging. So, if you have lots of changes to make in a document, it would be nice if there was a way to tell the browser to hold off on the reflow. And, then, when you're done, apply all those changes once, with a single reflow. That's where document fragments come in.

A document fragment is basically what it sounds like, a part of an HTML document. You create a fragment, add elements to it, set their properties and styles, and work on it, as much as you like. Because the fragment is not part of the DOM, nothing you do to it triggers a reflow, so it's a very optimized workflow. When you're finally done, and ready to display the HTML you've created, you can add the fragment to the DOM, and a single reflow occurs rendering the new content, along with the old.



Let's see this in action, first without fragments, and then using them. I've set up a simple project to get started. You can get the files from this course's downloads. There is an HTML file called, "fragments.html." All this does is load a script called, "fragments.js." In that JavaScript file, you can see that I've already created a single variable called, "standings", which contains an array of objects. This happens to be the 2013 football standings for the AFC East, showing each team's wins, losses, and ties.



The idea is to take this data, and dynamically create an HTML page that shows the results. Now, this is just a static array. But, in a real application, you'd probably be accessing some online web service to get some up-to-date information. That data would come back to you, possibly in a format very much like this array. So, this will get you started in creating pages from dynamic data. In lesson five, we'll be using a real web service, to get real dynamic data.

Now, because we have an array, we can loop through it with a for loop, and get a reference to each object in that array. We'll call that object, "team." The team object will have four properties: name, wins, losses, and ties. We can use the data in those properties to create, and populate some HTML elements. First, the name. Let's use an h2 tag for that. We'll say var heading equals document createElement h2. And, we'll add that right to the document body. And, then, we'll set the headings textContent to team.name.

Then, we'll do the same thing with each statistic; wins, losses, and ties. But, let's use paragraph elements for those. Create the element, add it to the page, and set the textContent. Now, let's see what that looks like. Okay, it probably needs some better styling, but at least all the data is showing up correctly.

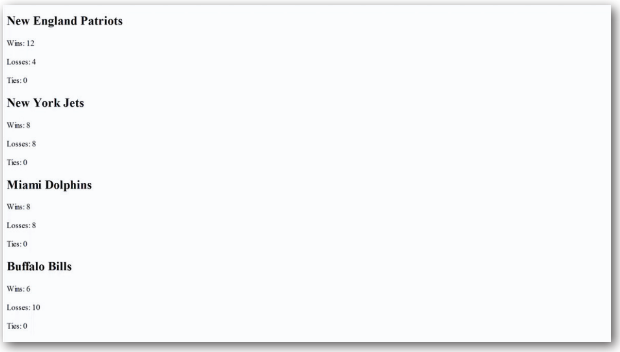
```
for(var i = 0; i < standings.length; i++) {
    var team = standings[i];

    var heading = document.createElement("h2");
    document.body.appendChild(heading);
    heading.textContent = team.name;

    var wins = document.createElement("p");
    document.body.appendChild(wins);
    wins.textContent = "Wins: " + team.wins;

    var losses = document.createElement("p");
    document.body.appendChild(losses);
    losses.textContent = "Losses: " + team.losses;

    var ties = document.createElement("p");
    document.body.appendChild(ties);
    ties.textContent = "Ties: " + team.ties;
}
```



But, let's walk through this from the viewpoint of the browser. Here, when we say appendChild heading, this causes a reflow, and the page is re-rendered. Then, when we set the textContent of that heading, this triggers another reflow. Then, two more reflows for each of the three stats. That's eight reflows per team, or 32 all together. It's kind of like, refreshing the page 32 times.

Now, if you're sharp, you may have noticed that we could have assigned the textContent, before we added each element of the page. And, indeed, this would have cut down the reflows by half, but that's still 16 reflows. And, this is a far simpler page than almost anything you're like to work on in the real world. When you get into far more complex pages, limiting your reflows can be vital to getting decent performance in your application.

So let's use a document fragment, and get it down to a single reflow. Before the loop, we just create a fragment by saying, var fragment equals new Documentfragment. Then, instead of adding each element to the document body, add it to the fragment, instead. Finally, when the fragment is fully populated, add it to the document body. One reflow, same result.

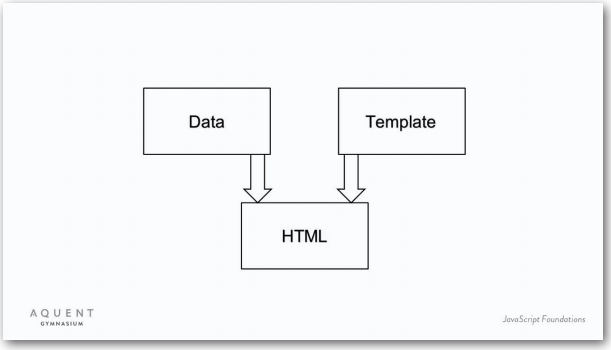
```
30  for(var i = 0; i < standings.length; i++) {
31      var team = standings[i];
32
33      var heading = document.createElement("h2");
34      fragment.appendChild(heading);
35      heading.textContent = team.name;
36
37      var wins = document.createElement("p");
38      fragment.appendChild(wins);
39      wins.textContent = "Wins: " + team.wins;
40
41      var losses = document.createElement("p");
42      fragment.appendChild(losses);
43      losses.textContent = "Losses: " + team.losses;
44
45      var ties = document.createElement("p");
46      fragment.appendChild(ties);
47      ties.textContent = "Ties: " + team.ties;
48  }
49
50  document.body.appendChild(fragment);
```

I like to think of document fragments using a restaurant analogy. The chef of a restaurant wouldn't prepare each bite of food individually, run out to your table, feed it to you, and then, run back to cook the next bite. The meal is prepared, as a whole, and served at once on a single plate. The plate is your fragment. Prepare your meal of elements, and serve it up to the page in one shot.

Okay, no more stupid analogies. Instead, we'll move on to an even more powerful way of creating dynamic documents, using HTML templates.

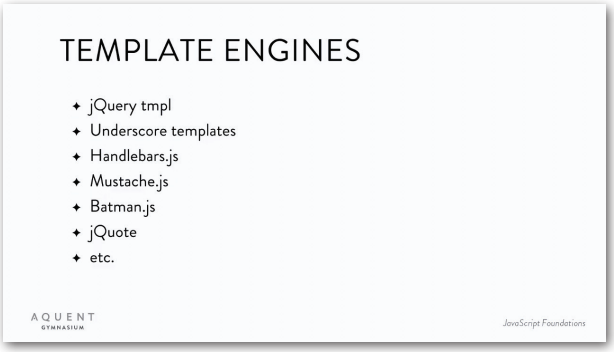
We started out this lesson learning how to create HTML elements, and add them to a page, then how to remove them, and reposition them, and then how to create a whole section of an HTML document in a fragment, and add that to the page.

The last example showed a pretty powerful way of creating somewhat complex HTML content, based on raw data. But still, we needed to create each element object individually in code, and add it to the document fragment, and then add that fragment to the page. Ideally, we wouldn't have to do all that work by hand.



It would be great if we could just take some specially formatted data, and some specially formatted HTML, and somehow just glue them together, and have it all look good. Well, that's actually what templates are all about.

Templates are not a built-in, standard part of HTML, but they fulfill such an important need that many different templating engines have been created over the years. There is, or at least was, a jQuery templating plug-in, though it's not in active development anymore and has largely been overtaken by some other systems. Two that you'll likely hear mentioned often are Handlebars.js and Mustache.js.

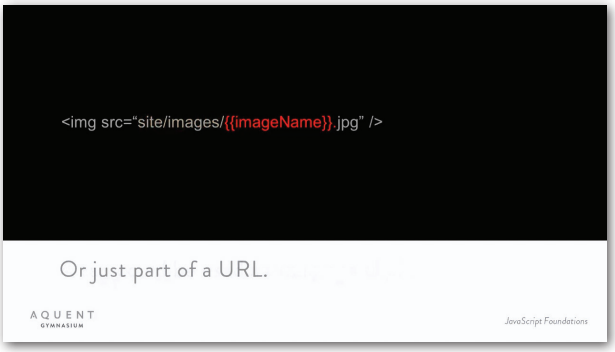


Both of these get their names from the curly brackets that are a key to much of their syntax. Turned on its side, this symbol looks a bit like a mustache; specifically, an old-fashioned, handlebar mustache. These two engines are also, very similar. In fact, you can use Mustache.js templates directly with the Handlebars.js templating engine.

For this video, we're going to use Mustache.js. But, if you find templates useful, I urge you to check out Handlebars and some of the other templating engines that are out there, right now, and discover their strengths, and weaknesses.

The concept behind templates is that you write a bit of HTML that is the template itself, but you use a special syntax to specify values that can be replaced at runtime. In Mustache, these are called tags, and Handlebars calls them expressions. In Mustache, you create a tag by enclosing it in double brackets, or mustaches.

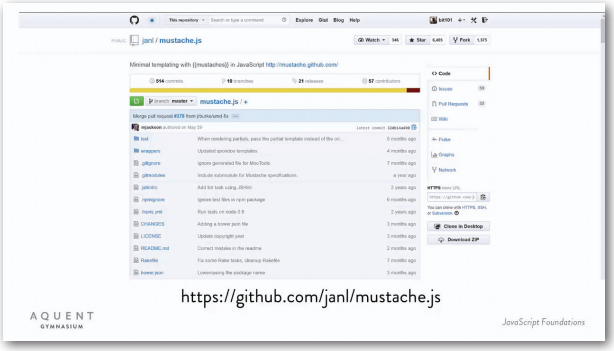
Here's an example of a very minimal template. At runtime, you would use Mustache.js to get this template, and replace name with some dynamic data, probably someone's name, such as Bob. Mustache would then generate HTML, like this, which you could then add to your page.



Now, realize that this can get a lot more powerful than just replacing text content. Everything in an HTML document is represented by text, so you could use template tags to represent a URL to an image, for an example, or just the name of an image, with the rest of the URL already determined. Or, you could do the same thing with audio, or video files, or virtually anything else that can be represented in HTML. In fact, in the project at the end of this lesson, we'll set CSS styles using template tags.

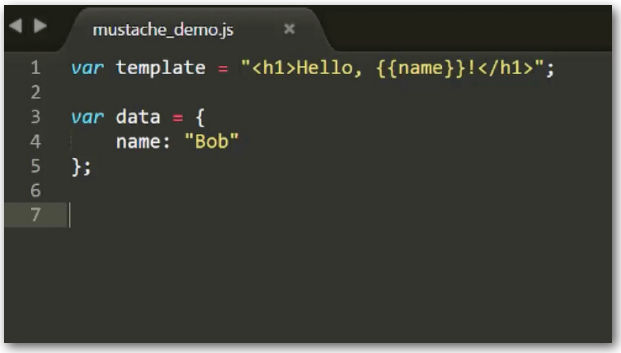
So, how do we go about doing all this? There are a few steps. First, we need to include the Mustache.js library in our page. Then, we need to create a template. We'll need a data source to fill in the template. Then, we need to use Mustache to inject the data into the template, and get back the final HTML. Finally, we can add that HTML to the page.

Adding the library to the page is fairly easy. You need to download the library from this address. The only file you'll actually need is the Mustache.js JavaScript file. It's also included in this lesson's downloadable files. Here's the HTML file for this project. Include the Mustache.js file in the HTML page, like you would any other JavaScript file. Make sure you put it before your own scripts, since your script will depend on Mustache being ready, and loaded, and initialized.



Next, we need a template. A template can be created a few different ways. The simplest is just to create it as a string. Here, in the mustache_demo.js file that's loaded from the HTML, let's create a string, called, template, and simply, create an h1 element with some text in a Mustache tag called, name.

Then, we need a data source. This is going to be an object with a property called, name. Name will be a string. Hopefully, you can see where this is going. The Mustache tag, name, is going to get replaced by the property, name, which contains the string Bob. We put the template together with the data by calling the mustache.render method. This gets past the template, and the data. So, we can just do that in our code.



Now, this HTML variable that we get back will be a string containing the template, with any tags replaced by actual values from the data. Now, there are a few ways to add this string to the document. I'm simply going to create a div, and set the div's inner HTML to this HTML string. Then, I'll append this div to the document body and that should be it. Let's run it in a browser, and see. Great. Hello, Bob!

Now, let's give it some more data. First, let's look at the data object. We'll change the name to firstName, and then we'll give Bob a last name. Now, we can change the template, giving it two tags, firstName and lastName. When we run that, sure enough, Bob's got a last name.

To summarize, Mustache looks through the template for any tags. It gets the name of each tag, and then looks over the data object for a property with the same name. When it finds that, it replaces the tag with a value of that property.

It can actually get a lot more complex than that. Your data object could contain an array. There's a special Mustache syntax for handling arrays that will create a new HTML object for each element in the array.

```
mustache_demo.js
1  var template = "<h1>Hello, {{name}}!</h1>";
2
3  var data = {
4      name: "Bob"
5  };
6
7  var html = Mustache.render(template, data);
8
9  var div = document.createElement("div");
10 div.innerHTML = html;
11 document.body.appendChild(div);
```

So you could use this to create an HTML list with multiple list items from that array, and your data object could have properties that are objects in themselves, and there's another way to handle those kind of nested objects in Mustache. You can even use tags to call functions, and the tag will be replaced by the return value of that function. It's all very powerful, but we'll stick with simple values, here.

Now, I want to show you another way to create a template. Writing HTML within a string can become a real pain in the neck for anything more than a really simple template, like we've used so far. Using strings, you have to keep your whole template on one line, or start concatenating multiple strings, if you want to put them on multiple lines. With all the quotes, it gets really ugly, and hard to read, much less write.

So there's another little trick that will let you write the template, just like you'd write normal HTML. To do this, we'll actually put the template HTML inside a script tag on the page like this. Now, there are a couple things to take note of here. First is that, the script itself, has an ID property. We'll use this to find the script when we get ready to use the template.

```
<script id="template" type="x-tmpl-mustache">
   <h1>Hello, {{name}}</h1>
</script>
```

Mustache template in a script tag.

AQUENT GYMNASIUM — JavaScript Foundations

The second thing is that, the type of the script is x-tmpl-mustache, instead of what you've seen up to now in script tags, text slash JavaScript. If we used JavaScript as a type, here, then the browser would try to execute this stuff as a script. And, obviously, because it is not valid JavaScript, it would just fail. Setting it to this other value doesn't do anything by itself, but it means that the browser will read it, and store the template, but it won't try to execute it, or render it to the page.

```
<script id="template" type="x-tmpl-mustache">
   <h1>Hello, {{name}}</h1>
</script>

var template = document.getElementById("template").innerHTML;
```

Get the element by its id.

AQUENT GYMNASIUM — JavaScript Foundations

So, now we can get that script tag using getElementById, and we can get the HTML inside the script element by using innerHTML. So, let's try that. First, we'll create the script template tag in the HTML. This can be up in the head section because it's not going to do anything when it loads. We then write the template into the script.

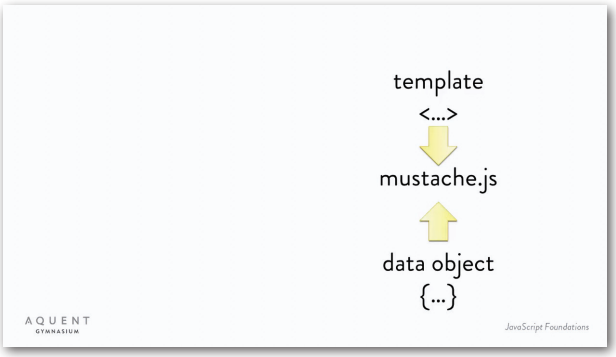
Back in the code, we can say, template equals document. getElementById(" template").innerHTML. This will return a string containing the template, just like before. And, other than that, it should all still work.

```
mustache_demo.html   mustache_demo.js
1  var template = document.getElementById("template").innerHTML
2
3  var data = {
4      firstName: "Bob",
5      lastName: "Smith"
6  };
7
8  var html = Mustache.render(template, data);
9
10 var div = document.createElement("div");
11 div.innerHTML = html;
12 document.body.appendChild(div);
```

Now, as you might guess, there's also a way to store templates, and external files, and load them in at runtime, but that gets a bit more complex. You'll need to use jQuery, or write some other code to load the template in using JavaScript, which is beyond the scope of this lesson. But, even just having the template in a script tag on the page will allow you to write much more complex templates, without the headache of using strings.

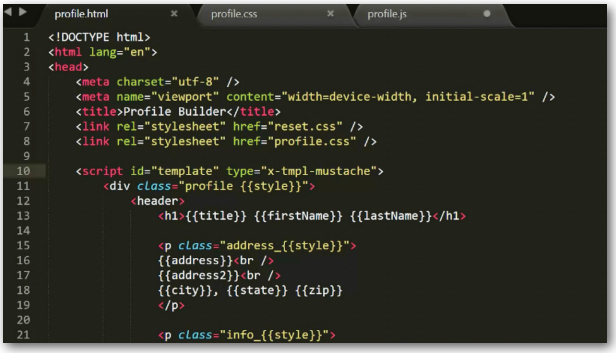
In the final video for this lesson, we'll create a more complex application using Mustache, and incorporating many other principles you've learned throughout the course, so far. When you're ready, let's go.

Well, let's get down to business and use a bunch of what we learned in this lesson to create a real-world application. The concept of this application will be a profile builder. The principles here might be useful if you're working on a site that allowed users to create a resume, or a profile for a social network type of site. The user is presented with a form to fill out. When the preview button is pressed, JavaScript on the page will grab the value's input into that form, and use them to populate an object.



The object will be the data object that's used to render a Mustache template, and display the results on the page. The form will even contain a style drop down so that the users can see his or her profile rendered in a few different styles. We'll be selecting elements, reading their values, creating, and calling functions, creating objects, populating templates, and adding, and removing elements from the page. In the real application the entered data would probably also be sent to a server to be stored in a database. But, server communication will be covered in the next lesson, so this data won't be saved anywhere.

Now, I've enlisted the help of a professional designer for this project, which means that the HTML and CSS is a bit more sophisticated than I would ordinarily create for a demo. But, I think it's good for you to get practice working with designers and writing JavaScript to address more complex pages. You need to be able to read the HTML and CSS, and sort out what's important for your application and what's not, which elements you'll need to select and manipulate, and what elements you'll need to listen for events on, et cetera.



Now, up at the top of the document here, you can see that we have the template in a script tag. But, let's skip over that for the moment. Down here in the body, you'll see that we have a form, with a whole bunch of elements.

There's a select element, which creates a drop-down list. Then, several text inputs, a couple of text area, another select, and a button. Note that, all of the input elements have IDs, so we'll be able to get their values from the script. Of course, if the designer didn't give everything IDs, you might need to add some of them on your own, in order to select those elements.

Also, note that the last select element allows the user to choose between four different style options. Below the form, there's an empty div with an idea of "preview". This is where we'll put the populated template.

```
    <h1>Profile Builder</h1>
</header>

<section role="region">
    <h2>Name:</h2>
    <select id="title">
        <option value=""> </option>
        <option value="Mr.">Mr.</option>
        <option value="Miss">Miss</option>
        <option value="Mrs.">Mrs.</option>
        <option value="Ms.">Ms.</option>
    </select>
    <input type="text" id="firstName" placeholder="First
    <input type="text" id="lastName" placeholder="Last N
</section>
```

```
    <select id="style">
        <option value="formal">Formal</option>
        <option value="light">Light</option>
        <option value="dark">Dark</option>
        <option value="colorful">Colorful</option>
    </select>
</section>

<section role="region">
    <h2>Preview</h2>
    <p>
        <button id="previewBtn"><strong>Build Profile</strong></button>
    </p>
</section>

</div>

<div id="preview" role="article"></div>

<script src="mustache.js"></script>
<script src="profile.js"></script>
```

And, finally, there are some script tags to load Mustache.js in our main script file. Now, let's go back out of the template. This has an ID of "template," and you can see that many of the tags have obvious connections to the form elements, such as title, first name, last name, address, et cetera.

But, also notice that these style attributes have Mustache tags in them, as well. The main div in the template has a class of "profile", plus the tag style. Say the user chooses the formal style from the drop down. This will be transformed into class equals quote profile space formal, so we'll get both of those classes.

```
<script id="template" type="x-tmpl-mustache">
    <div class="profile {{style}}">
        <header>
            <h1>{{title}} {{firstName}} {{lastName}}</h1>

            <p class="address_{{style}}">
            {{address}}<br />
            {{address2}}<br />
            {{city}}, {{state}} {{zip}}
            </p>

            <p class="info_{{style}}">
            email: <span class="contact_{{style}}">{{email}}</
            web site: <span class="contact_{{style}}">{{websit
            </p>
        </header>

        <section role="region">
```

But, look at this address paragraph. It has a style of "address_ style". This will become address_formal. Now if we look in the profile CSS file that this page uses, down here a bit, you can see a profile class. But, below that, a formal class, and then, address formal, contact formal, interest formal, et cetera. The same holds true for all of the other style choices.

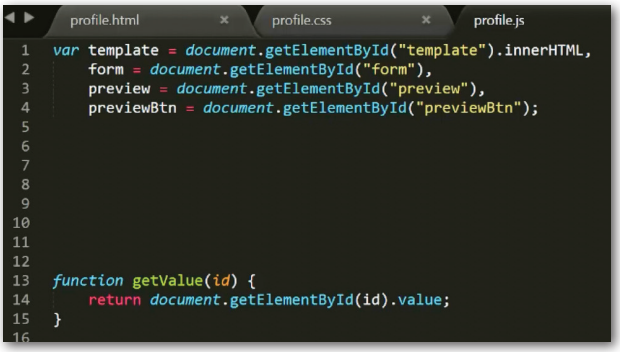
profile.html | profile.css | profile.js

```
var template = document.getElementById("template").innerHTML,
    form = document.getElementById("form"),
    preview = document.getElementById("preview"),
    previewBtn = document.getElementById("previewBtn");
```

So the idea is that, the style itself is part of the variables data that gets filled in the template. Okay, time to jump over to the code. Now, it can be hard to know where to start, when facing a blank page for an application. One good strategy is to first decide on and create some variables. What values or objects are you going to need throughout the program? I'm thinking I want to get a reference to a few important items from the HTML; the template data, the div that holds the form, the empty div that will eventually put the preview, and the preview button.

I use selection methods to get all of those, along with the inner HTML for the template. Next, think of what actions are going to need to happen. I know that when the user clicks the button, I'll need to get all the data from the form, and then, use Mustache to render the template with that data. So, first, let's think about getting all that data. We're going to need to access all of those form elements by their IDs, and then, get the value of each one.

So rather than writing getElementById over, and over, and over, for every single one of those form elements, I'm going to create a function called, getValue. We'll take an ID as a parameter. Inside that, I'll call getElementById to get the element with that ID, and return its value.



Now, if we want, say, the first name value, we can just say, getValue, first name. Now, I'm also thinking that I want a way to fill in this form programmatically for testing purposes. So, I need a way to set values for all those elements.

I'll create another function, setValue. This will take an ID in a value, and it will set the value of the element with that ID. This allows us to make a function that will automatically fill the entire form out, just for testing. I don't have to go through, and manually type in all the values, for all those fields every time I need to test the application.

But, I also don't want to hardcode values in the HTML itself, as it would be a pain to edit the HTML again to remove all those values, once I was ready to make the app go live. This fill-form function will supply default test values for all the test fields while I'm working on the app. It uses the setValue function we just created. When I'm ready to publish the app, I can just comment out, or remove that function, and the form will remain empty. I'll call the function here, and we can run the app in the browser, and see that it's filled in with our test data.

Now we need one more function that will grab all the values from those fields, and use them to populate a data object that can be used with the template. We can use a lot of shorthand here, first, simply returning a blank object. Then, we can start adding in properties using getValue. I'll add title, first name, last name, and then I'll just paste all the rest in, because you don't need to sit here, and watch me type.

Now we just need to tie all this together when the user clicks the button. So, add a click listener to the preview button, and that we'll call, getFormData, storing that object in a variable called data. We'll then use the template, plus this data, in a call to Mustache.render. The result of that will be a string stored in a variable called HTML. We could assign that string of HTML to the preview div's inner HTML and that's that.

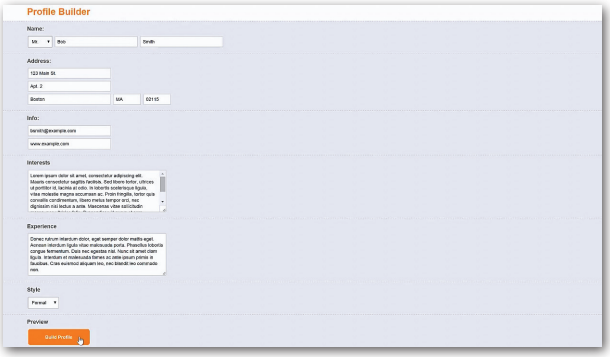
But let's remove the form, so that we can see the result clearly. We can just say, document.body.removeChild form. Now all that's left here, is the preview with the populated template. Let's try it out. The form is there, all ready to go, so you don't have to fill anything in.

```
fillForm();

previewBtn.addEventListener("click", function(){
    var data = getFormData(),
        html = Mustache.render(template, data);

    preview.innerHTML = html;
    document.body.removeChild(form);
});

function getFormData() {
    return {
        title: getValue("title"),
        firstName: getValue("firstName"),
        lastName: getValue("lastName"),
        style: getValue("style"),
        address: getValue("address"),
        address2: getValue("address2"),
        city: getValue("city"),
        state: getValue("state"),
        zip: getValue("zip"),
        email: getValue("email"),
        website: getValue("website")
```



You can accept the default test values, or fill some fields in by yourself if you want. Then click preview, and there's our template, with the default normal style. Refresh the page, and you're back to the main form. This time choose the light style, and there that is. Try the dark style, refresh again, choose colorful.

Let's change the name this time. Yep, that's colorful and has the name we just input. This wraps up our lesson on advanced DOM manipulation. We've gone from creating individual elements, to populating entire templates with dynamic data objects. In lesson five, we'll reach outside our little sandbox, and start communicating with the outside world.

But, before that, you have an assignment to do. First of all, as usual, do your quiz. Second, go to a site of your choice, open the console, and practice creating elements, and then adding them to the page using appendChild. Try locating specific points within the DOM to add them using insertBefore. Practice removing the elements you added, or removing existing elements, and re-adding them in different positions.

Do this until you feel comfortable creating, adding, and removing elements. Three, check out the documentation for Mustache.js at this site. Try some of the more advanced techniques, such as using an array of data to make multiple items, or any other techniques that look interesting, or useful.

Four, take a look at Handlebars.js and compare and contrast it to Mustache. Try converting the Mustache demo project earlier in this lesson, or the final project, to use Handlebars instead of Mustache. Are there any advantages?

ASSIGNMENT #2:

Go to a site of your choice, open the console and practice creating elements, and adding them to the page using appendChild.

Try locating specific points within the DOM to add them using insertBefore.

Practice removing the elements you added, or removing existing elements and re-adding them at different positions.

Do this until you feel comfortable creating, adding and removing elements.

AQUENT GYMNASIUM JavaScript Foundations

ASSIGNMENT #3:

Check out the documentation for Mustache.js at this site:

https://github.com/janl/mustache.js

Try some of the more advanced techniques, such as using an array of data to make multiple items, or any other techniques that look interesting or useful.

AQUENT GYMNASIUM JavaScript Foundations

ASSIGNMENT #4:

Take a look at Handlebars.js

http://handlebarsjs.com/

Compare and contrast it to Mustache. Try converting the mustache demo project from earlier in this lesson, or the final project over to use Handlebars instead of Mustache. Are there any advantages?

AQUENT GYMNASIUM JavaScript Foundations