Creating and Inserting DOM Nodes Lab

- Due No Due Date
- Points 1
- Submitting a website url

 (https://github.com/learn-co-curriculum/phase-0-the-dom-modifying-elements-lab)  (https://github.com/learn-co-curriculum/phase-0-the-dom-modifying-elements-lab/issues/new)

Learning Goals

- Create DOM elements programmatically
- Add elements to the DOM
- Update elements using `innerHTML`
- Change properties on DOM nodes
- Remove elements from the DOM

Introduction

Now that you have an understanding of the DOM and powerful tools for selecting the right elements, it's time to learn how to:

1. Create new nodes
2. Delete nodes
3. Update node properties

If you haven't already, **fork and clone** this lab into your local environment. Navigate into its directory in the terminal, then run `code .` to open the files in Visual Studio Code.

Create DOM Elements Programmatically

`document.createElement()`

Creating an element in JavaScript is an easy process. You simply call `document.createElement('tagName')`, where `tagName` is the name of any valid HTML tag (`'p'`, `'div'`, `'span'`, etc.).

Open the `index.html` file in your browser and open up the browser's console. In the console, enter:

```
const element = document.createElement("div");
```

Then take a look at the Elements tab. The element doesn't show up on the page. Why not?

Add Elements to the DOM

To get an element to appear in the DOM, we have to `append()` it to an existing DOM node. To go back to our tree metaphor, we have to glue our new leaf onto a branch that's already there. We can start as high up on the tree as `document.body`, or we can find a more specific element using any of the methods we've learned for traversing the DOM.

append()

Let's append `element` to `body` to start:

```
document.body.append(element);
```

(Recall that `element` is a variable containing the `div` we created above.)

Now if you look at the Elements tab, you'll see our new (empty) `<div>` nested inside the `body` element.

Next, let's create an unordered list:

```
const ul = document.createElement("ul");
```

To populate our unordered list, we'll use a `for` loop to create three `li`'s, give them some content, and append them to the `ul`:

```
for (let i = 0; i < 3; i++) {
  const li = document.createElement("li");
```

```
    li.textContent = (i + 1).toString();
    ul.append(li);
  }
```

In each iteration of our loop, we calculate the value `i + 1` (an integer), turn it into a string using JavaScript's `toString()` method, and assign the result as the value of the `li` 's `textContent` attribute.

> Note: although the `textContent` attribute must be a string, the code would still work even if we didn't use the `toString()` method — JavaScript will turn the value into a string for us. However, for clarity and completeness, it is best to set it to a string value explicitly.

Finally, we'll append the `ul` to the `div` we created:

```
element.append(ul);
```

You should now see the unordered list rendered on the page, and see the new elements in the "Elements" tab, like this:

```
<div>
  <ul>
    <li>1</li>
    <li>2</li>
    <li>3</li>
  </ul>
</div>
```

Note that, each time we create a new element, we create a variable and save a reference to it. That makes it quite easy to make additional updates and to add it to the DOM using `append()` .

Add Elements to the DOM via `innerHTML`

Creating elements and then appending them into the DOM is a multi-step process. It's also the ***safest and most reliable way*** to add content to the DOM. Most repeated code can be removed by using variables, functions and loops. It's The Right Way.

That said, however, there's another process that will also work, using `Element.innerHTML`. Inside our loop above, we create an `li` element and set its `textContent` property to a simple number (expressed as a string) that represents the current iteration through the loop. Once we append the `li` to the `ul` and the `ul` to the document `body`, we see our unordered list in the browser window. This is a perfectly valid way to use `textContent` to add content to the DOM — essentially, `textContent` changes only what text shows up inside a DOM element:

```
li.textContent = "Hi there!";
// => <li>Hi there!</li>
console.log(li.textContent);
// => "Hi there!"
```

Imagine, however, that we want to add content that's more complicated.

Assume our HTML includes a `main` element with an `id` of "main." We can grab that element and set its `innerHTML` attribute to any HTML we like:

```
const main = document.getElementById("main");
main.innerHTML =
  "<h1>Poodles!</h1><h3>An Essay into the Pom-Pom as Aesthetic Reconfiguration of the Other from a post-Frankfurt School
```

Here we are using one big, long string, complete with multiple HTML tags, to create the following HTML in the DOM:

```
<main id="main">
  <h1>Poodles!</h1>
  <h3>
    An Essay into the Pom-Pom as Aesthetic Reconfiguration of the Other from a
    post-Frankfurt School Appropriationist Perspective
  </h3>
  <p><em>By: Byron Q. Poodle, Esq., BA.</em></p>
</main>
```

This process works but it is **not** recommended for several reasons. First, it's more error-prone, and the errors can be difficult to find. Second, it can negatively impact site performance. Finally, if you're inserting user-derived data (e.g., comments) into the DOM using `innerHTML`, you can expose

your site to the risk of users **injecting malicious code** (https://www.reddit.com/r/learnjavascript/comments/9502x5/is_innerhtml_still_considered_bad/e3p31go/?utm_source=share&utm_medium=web2x&context=3) .

Programmatically creating and appending elements is safer and more efficient, and it results in code that's easier to read, easier to debug, and easier to maintain.

Change Properties on DOM Nodes

We can change the appearance of a DOM node using its `style` attribute. Try this out in the console:

```
const element = document.getElementById("main");
element.style.height = "300px";
element.style.backgroundColor = "#27647B";
```

You've changed what's on the screen!

Feel free to set as many properties as you'd like — this is a good chance to look around and explore different properties of DOM elements.

Let's adjust the display. Add some text:

```
element.textContent = "You've changed what's on the screen!";
```

Then change the style to see the effect:

```
element.style.fontSize = "24px";
element.style.marginLeft = "30px";
element.style.lineHeight = 2;
```

Perhaps the most common way to change how things appear in the DOM is by changing an element's `class` attribute. As you know from CSS, we often change the way a bit of rendered HTML appears by adding or removing a class.

For example, we could create an `alert` class that turns the text red (using the CSS `color` attribute) and makes it big (using the CSS `font-size` attribute). We can then use JavaScript to first grab the element and then add the class by updating the element's `className` property. This has the same effect as setting the `class` property in the HTML. The `className` property expects a `String` with one or more class names, separated by spaces:

```
element.className = "pet-listing dog";
```

Check out the Elements tab to see the effect of this change:

```
<main id="main" class="pet-listing dog"></main>
```

Another way to accomplish the same thing is by using the **`Element.classList` property (https://developer.mozilla.org/en-US/docs/Web/API/Element/classList)**. This property has `.add()` and `.remove()` methods that can be used as follows:

```
element.classList.remove("dog");
element.classList.add("cat", "sale");
```

This approach allows you to easily add and remove classes programmatically, without having to create a long string of class names.

```
<main id="main" class="pet-listing cat sale"></main>
```

Separation of Concerns

An important thing to bear in mind is that we only want to use JavaScript to change the appearance of an element when we need to make a change dynamically, i.e., in response to user actions. This goes back to a fundamental programming concept about separating concerns between technologies:

- HTML defines the structure of the website (not appearance or functionality)
- JavaScript defines functionality of the website (not structure or styling)
- CSS defines the visualization and style of the website (not structure or functionality)

Defining the base CSS should still happen in the CSS files that are loaded into the DOM when the page is opened.

Remove Elements from the DOM

We know how to add elements and change their attributes. What if we want to remove an element from a page?

`removeChild()`

We use `removeChild()`, as you might guess, to remove a particular child of an element:

```
someElement.removeChild(someChildElement);
```

Let's take a look at a more complex example:

```
const ul = document.getElementsByTagName("ul")[0];
const secondChild = ul.querySelector("li:nth-child(2)");
ul.removeChild(secondChild);
```

Here you can see the power of `querySelector()`: we can use it to find the second `li` element of `ul`. We then pass that element as the argument to our `removeChild` method, which removes the element from our `ul`.

What if we want to remove the whole unordered list (`ul`)?

`element.remove()`

We can just call `remove()` on the element itself:

```
ul.remove();
```

And it's gone!

Instructions

From this point forward, many of the labs will work a little differently from ones you've done before. Specifically, the tests will mock the process of 1) running JavaScript code in the browser and 2) seeing the results of that code represented in the DOM. Take a look at `test/indexTest.js` to see the tests' descriptions of the changes your code should be making to the DOM elements.

Note that you do not need to create functions for this lab. Just create the line or lines of JavaScript necessary to pass each test. As usual, you will write your code in the `index.js` file.

One final note: the last test in the `test/indexTest.js` file is looking for the text "YOUR-NAME is the champion" (with your name — or whatever text you choose — in place of YOUR-NAME) inside your newly created DOM node. While there are a number of ways you could accomplish this, you should use either the **`textContent`** (https://developer.mozilla.org/en-US/docs/Web/API/Node/textContent) or **`innerHTML`** (https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML) property to set the text inside your DOM node. The `innerText` property would technically work as well; however, the tests won't pass if you use this approach, and generally it's not a good practice to use `innerText` when setting the contents of an element. **This StackOverflow answer** (https://stackoverflow.com/a/35213639) does a good job explaining some differences between these properties.

Resources

- **document.createElement()** (https://developer.mozilla.org/en-US/docs/Web/API/Document/createElement)
- **append()** (https://developer.mozilla.org/en-US/docs/Web/API/Element/append)
- **removeChild()** (https://developer.mozilla.org/en-US/docs/Web/API/Node/removeChild)
- **element.remove()** (https://developer.mozilla.org/en-US/docs/Web/API/ChildNode/remove)
- **classList Property** (https://developer.mozilla.org/en-US/docs/Web/API/Element/classList)