Lesson Weekend

Introduction to Programming (/introduction-to-programming) / Arrays and Looping (/introduction-to-programming/arrays-and-looping) / Adding and Removing HTML Elements

Text Cheat sheet

In the last course section we learned two ways to add and remove content from our webpages:

- Use inline styles to show and hide HTML elements. With this method we update the `style` attribute of an HTML element.
- Use the `innerText` property to change the text of elements like paragraphs, headings, and others.

However, both of these ways are limited — we have to have the HTML element created in our HTML before we can show, hide, or otherwise change it. Let's increase our toolkit and learn how to actually add and remove HTML elements from our scripts! We'll do this by using methods from the `document` and `Element` objects (Web APIs). At the end of this lesson, we'll also list links to additional methods and properties that can also be used to add HTML elements and text content.

You can optionally open your DevTools and try out the code that we cover in this lesson.

Creating New Elements

To create new HTML elements, we'll use the `document.createElement()` (https://developer.mozilla.org/en-US/docs/Web/API/Document/createElement) method. This method has one required parameter for an HTML tag name in a string, like `"p"`, `"form"`, `"img"` and so on. (There's also an optional parameter, but we won't cover it because it's not useful to us right now.) Let's look at some examples! We'll put these into the DevTools console.

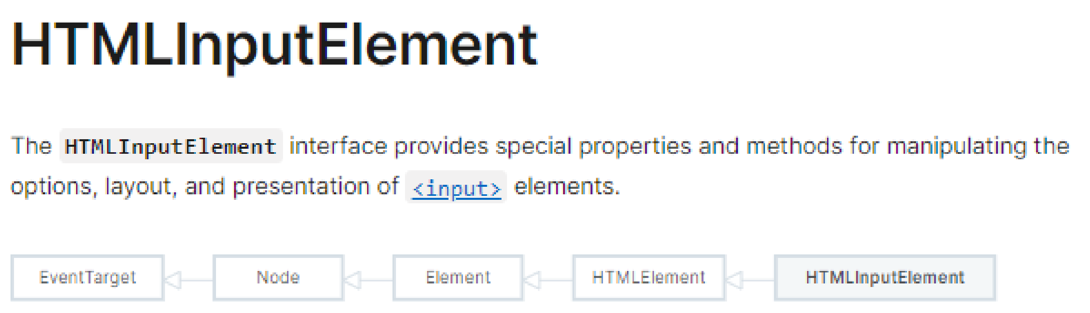
```
> const pElement = document.createElement("p");
> pElement;
<p></p>
> Object.prototype.toString.call(pElement);
"[object HTMLParagraphElement]"
```

There's a couple things to notice in the above code snippet. First, the `document.createElement()` method creates HTML element objects, the same category of objects that we've been working with thus far. In our example, since we're creating a paragraph element, that means we're working with an `HTMLParagraphElement` object. To see a complete list of all HTML element objects, visit the MDN documentation on the HTML DOM API (https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API#html_element_interfaces_2).

Also notice that the `document.createElement()` method returns an empty paragraph element. If we want to add text or an attribute, we need to do that as a separate step.

A Review of Inheritance and Browser Structures

Keep in mind that HTML element objects, like `HTMLParagraphElement` or `HTMLImageElement`, inherit functionality from multiple other objects. MDN offers a helpful graphic on the reference page for each HTML element object to show us this chain of inherited functionality. For example, if we look at the `HTMLInputElement` reference page (https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement), we'll see this image in the initial description:


HTMLInputElement

The `HTMLInputElement` interface provides special properties and methods for manipulating the options, layout, and presentation of `<input>` elements.

This tells us that some properties and methods are inherited from these objects:

- `HTMLElement`
- `Element`
- `Node`
- `EventTarget`

We've worked with properties and methods from all of the objects in this list, except for the `Node` object, and we'll learn about that one soon.

All of these objects (and many others) combine to make up the functionality of the DOM/HTML DOM, and all of these collectively are categorized as Web APIs (https://developer.mozilla.org/en-US/docs/Web/API), the tools and standards that describe how web browsers function.

Web APIs are divided into two categories:

- Interfaces (https://developer.mozilla.org/en-US/docs/Web/API#interfaces), which are simply object types, like `window`, `Element`, `HTMLCollection`, `NodeList`, etc.
- Specifications (https://developer.mozilla.org/en-US/docs/Web/API#specifications), like the DOM (https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model), which name the standards and functionality of different web tools. Specifications (like the DOM and others) are always made up of one or more interfaces (object types).

Hopefully this is a helpful conceptual review of the web browser structures and tools we are using in our code. When we talk about this information, we're absolutely "looking under the hood" of how web browsers function. There's so much to Web APIs! If you look at the MDN documentation homepage for Web APIs (https://developer.mozilla.org/en-US/docs/Web/API), you'll see a long long list, and it's highly likely you won't work with the majority of Web APIs in your entire career as a developer. However, it's important to know the basic structures of the web so that we can know where to reference the information we may need in the future.

If this information is overwhelming, don't worry, you'll come to understand it in time. For now, just keep your focus on driving: using the objects, methods, properties, and JavaScript that we've learned about so far.

Adding Attributes

As a quick refresher, if we want to add, remove, or get attributes to any HTML element, we need to use these methods:

- `Element.setAttribute()` (https://developer.mozilla.org/en-US/docs/Web/API/Element/setAttribute)
- `Element.removeAttribute()` (https://developer.mozilla.org/en-US/docs/Web/API/Element/removeAttribute)
- `Element.getAttribute()` (https://developer.mozilla.org/en-US/docs/Web/API/Element/getAttribute)

Adding Text

We'll use the `Element.append()` (https://developer.mozilla.org/en-US/docs/Web/API/Element/append) method to add text to an element. We call this method on the element that we want to add text to, and the argument we pass in will be string with the text we want to add. Let's look at the code!

```
> const pElement = document.createElement("p");
> pElement;
<p></p>
> pElement.append("text");
> pElement;
<p>text</p>
> pElement.innerText;
"text"
```

Adding and Removing Elements to/from the DOM

`Element.append()`

The `Element.append()` (https://developer.mozilla.org/en-US/docs/Web/API/Element/append) method also works to add elements to the DOM. The `Element.append()` method will add a new element inside and at the end of the element we're targeting.

The argument that we pass into the element represents the HTML element that we're adding to the DOM. Let's see how this works! You can try the following code in the DevTools on just about any webpage, since just about every webpage uses a div element. In the following code snippet, we're simply grabbing the first div that appears in the DOM and adding a paragraph element after it.

```
> const pElement = document.createElement("p");
> pElement.append("text");
> const firstDiv = document.querySelector("div");
> firstDiv.append(pElement);
```

With this first example, since we always nest other elements inside of divs, the `Element.append()` method adds the new element to the end of the inside of the div. We'll see the same behavior with ULs and OLs.

For elements that we shouldn't nest other HTML elements inside of, like adding a paragraph tag inside of a heading, the `Element.append()` method will still add the new element inside of that element. This is no good. Say we had a H2 heading element and we appended a paragraph tag inside of it, this is what the H2 element would look like:

```
> const pElement = document.createElement("p");
> pElement.append("text and more text");
> const firstH2 = document.querySelector("h2");
> firstH2;
<h2>Best Chocolate Chip Cookies</h2>
> firstH2.append(pElement);
> firstH2;
<h2>Best Chocolate Chip Cookies<p>text and more text</p></h2>
```

We do not want that! So be careful about your use cases.

This detail about `Element.append()` can be hard to remember and may cause some bugs in our code if we forget how to use it properly. However, we shouldn't worry about memorizing the different use cases for `Element.append()`. Instead, we should develop good habits around referencing information and debugging. A good workflow here would be:

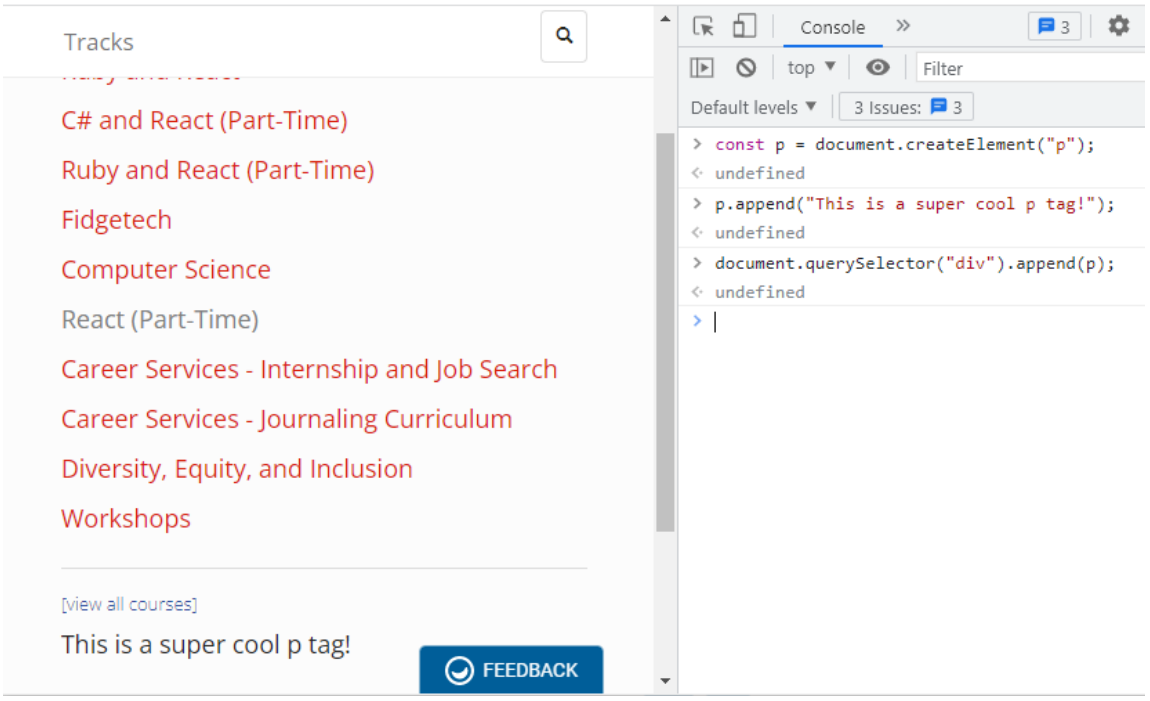
- Reference the documentation.
- Try out code in the DevTools console before you add it to your scripts.
- After you try it out in the DevTools console, inspect the newly added elements in the Elements tab of the DevTools console.

Since this workflow is so vital, let's review an example of using the Elements tab.

Review: Using the DevTools Elements Tab to Inspect Elements

In the following image, we've created a new p tag with the text `"This is a super cool p tag!"`. We then appended that p tag to the first div on the homepage of LHTP at learnhowtoprogram.com/tracks. We can see this new paragraph actually on the webpage!

```
> const p = document.createElement("p");
> p.append("This is a super cool p tag!");
> document.querySelector("div").append(p);
```



This is a super cool p tag!

```
> const p = document.createElement("p");
< undefined
> p.append("This is a super cool p tag!");
< undefined
> document.querySelector("div").append(p);
< undefined
>
```

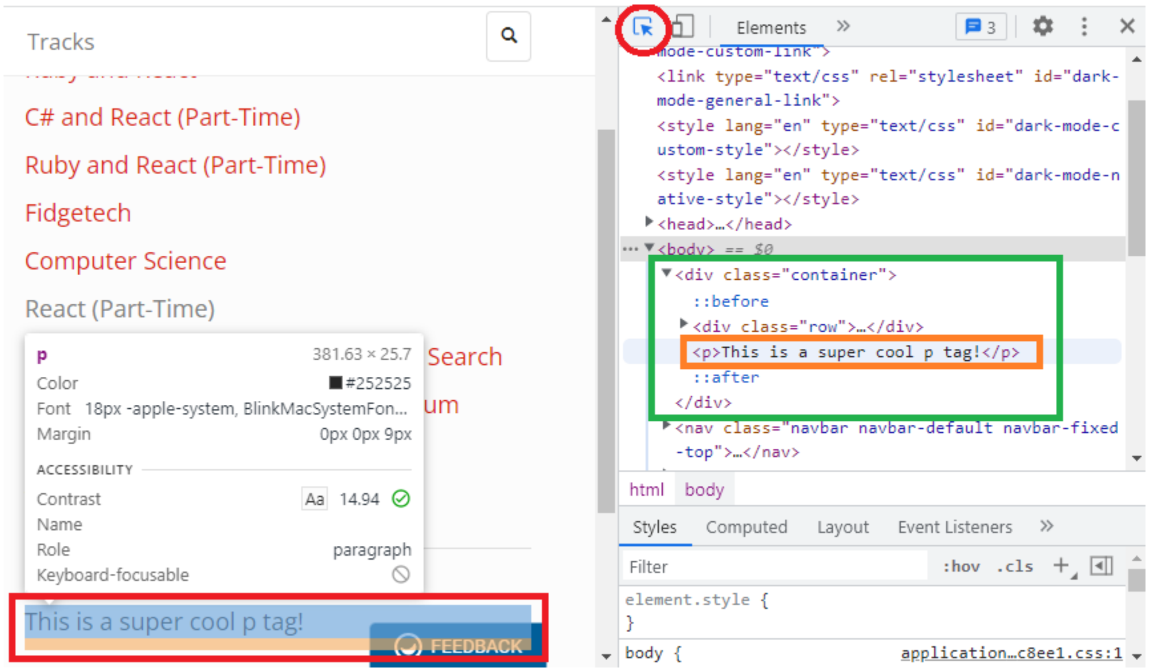
But how do we know where exactly this new paragraph has been added in the DOM? Did we add it inside of the div or after the div? We can use the DevTools Elements tab to inspect our HTML and find the exact location of the newly added paragraph. The following image shows the results of doing just that.

Here's the breakdown of the different highlights in the image:

- First, we enable the tool that allows us to select an element on the webpage to inspect. This button is highlighted by the red circle. When we hover over different elements on the webpage with our mouse cursor, the DevTools inspector will automatically highlight the corresponding element in the HTML inside of the DevTools window.
- We've selected the new paragraph `"This is a super cool p tag!"` on the webpage with our cursor, which is highlighted in the red

rectangle.

- The inspector has done its job, by highlighting the HTML for the p tag in the DevTools console. This is circled in the orange rectangle.
- The green rectangle highlights the first div on the webpage. We have now verified that the code `document.querySelector("div").append(p);` actually adds the p tag to the inside of the div at the end.



```
mode-custom-link">
<link type="text/css" rel="stylesheet" id="dark-mode-general-link">
<style lang="en" type="text/css" id="dark-mode-custom-style"></style>
<style lang="en" type="text/css" id="dark-mode-native-style"></style>
▶<head>…</head>
··· ▼<body> == $0
  ▼<div class="container">
      ::before
    ▶<div class="row">…</div>
     <p>This is a super cool p tag!</p>
      ::after
   </div>
  ▶<nav class="navbar navbar-default navbar-fixed-top">…</nav>
```

```
element.style {
}
body {                          application…c8ee1.css:1
```

`Element.prepend()`

The `Element.prepend()` (https://developer.mozilla.org/en-US/docs/Web/API/Element/prepend) method works in the exact same way as `Element.append()`, only it adds a new element at the beginning and inside of the element we call it on. This is great for `divs`, `ul`s, `ol`s, `body`, `head`, and other elements that we nest HTML elements inside of.

If I wanted to add a new list item `li` at the beginning of an unordered list, I would use the `Element.prepend()` method:

```
> const liElement = document.createElement("li");
> liElement.append("list item");
> const firstUL = document.querySelector("ul");
> firstUL.prepend(liElement);
```

Note that you can't print one element multiple times with prepend or append. If we do, we'll move the same element around, to the beginning or end of an element, but we won't print multiple copies of the same element. For example, the following code would only leave the list item `li` at the bottom of the unordered list:

```
> const liElement = document.createElement("li");
> liElement.append("text");
> const firstUl = document.querySelector("ul");
> firstUl.prepend(liElement);
> firstUl.append(liElement);
```

If we want a list item added before and after the heading we're targeting, we'll have to create two list item elements:

```
> const li1 = document.createElement("li");
> const li2 = document.createElement("li");
> li1.append("text");
> li2.append("other text");
> const firstUl = document.querySelector("ul");
> firstUl.prepend(li1);
> firstUl.append(li2);
```

`Element.before()` and `Element.after()`

To add an element before or after an element in the DOM we can use the `Element.before()` (https://developer.mozilla.org/en-US/docs/Web/API/Element/before) or `Element.after()` (https://developer.mozilla.org/en-US/docs/Web/API/Element/after) methods. These work in much the same way as `Element.append()` and `Element.prepend()`.

We can add an element before or after another element like so:

```
> const p1 = document.createElement("p");
> const p2 = document.createElement("p");
> p1.append("text");
> p2.append("other text");
> const firstH2 = document.querySelector("h2");
> firstH2.before(p1);
> firstH2.after(p2);
```

We can only use a newly created element once in the DOM. In the following example the newly created paragraph element will be added after the H2 tag.

```
> const p1 = document.createElement("p");
> p1.append("some text");
> const firstH2 = document.querySelector("h2");
> firstH2.before(p1);
> firstH2.after(p1);
```

Removing an Element

If we want to remove an element from the DOM, we simply have to get it with a `document` method and call the `Element.remove()` (https://developer.mozilla.org/en-US/docs/Web/API/Element/remove) method on it.

```
> const firstDivOnPage = document.querySelector("div");
> firstDivOnPage.remove();
```

Adding Multiple Elements to the DOM at Once

We can add multiple elements at once with the `Element.append()`, `Element.prepend()`, `Element.after()`, and `Element.before()` methods by passing in a series of arguments, each one representing an HTML element.

Here is an example with `Element.append()`:

```
> const ul = document.createElement("ul");
> ul.setAttribute("id", "iceCream");
> document.querySelector("div").append(ul);
> const liOne = document.createElement("li");
> const liTwo = document.createElement("li");
> const liThree = document.createElement("li");
> liOne.append("Chocolate");
> liTwo.append("Vanilla");
> liThree.append("Strawberry");
> document.getElementById("iceCream").append(liOne, liTwo, liT
hree);
```

Now we have an unordered list of favorite ice cream flavors randomly added to the end of the inside of the first div of the webpage we're on.

More to Explore on MDN

There are more methods and properties to explore that help us add text and HTML to our webpages. We won't cover all of them in the curriculum, but we will review some of them in upcoming lessons. As always, you are welcome to explore them on your own — and you may just see code examples online that use these tools!

This first set of methods and properties return `Node` (https://developer.mozilla.org/en-US/docs/Web/API/Node) or `Node`-related objects, or belong to the `Node` object itself. We'll revisit these in the Intermediate JavaScript course, and the list that follows is just a sampling of many more methods and properties. Remember `Node` is an object in the chain of inherited functionality for HTML element objects like `HTMLInputElement` and `HTMLHeadingElement`!

- `document.createTextNode()` (https://developer.mozilla.org/en-US/docs/Web/API/Document/createTextNode)
- `Node.appendChild()` (https://developer.mozilla.org/en-US/docs/Web/API/Node/appendChild)

- `Node.childNodes` (https://developer.mozilla.org/en-US/docs/Web/API/Node/childNodes)
- `Node.parentElement` (https://developer.mozilla.org/en-US/docs/Web/API/Node/parentElement)

This second set of methods and properties we will not cover in the curriculum:

- `Element.insertAdjacentHTML()` (https://developer.mozilla.org/en-US/docs/Web/API/Element/insertAdjacentHTML)
    - Important security considerations for using this method. (https://developer.mozilla.org/en-US/docs/Web/API/Element/insertAdjacentHTML#security_considerations
- `Element.innerHTML` (https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML)
    - Important security considerations for using this method, which make it a less favorable approach to adding HTML to the DOM (https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML#security_considerations)

If you're not interested in exploring more, you certainly don't need to! All of the methods to add and remove HTML elements to/from the DOM that we've covered in this lesson will meet our needs for the websites we build.

Previous (/introduction-to-programming/arrays-and-looping/document-query-methods-that-return-collections)
Next (/introduction-to-programming/arrays-and-looping/debugging-in-javascript-using-a-linter)

Lesson 10 of 50
Last updated February 28, 2023

disable dark mode

Epicodus (http://www.epicodus.com)