Element Selection Lesson Learning Objectives

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Given HTML that includes `<div id=”catch-me-if-you-can”>HI!</div>`, write a JavaScript statement that stores a reference to the HTMLDivElement with the id “catch-me-if-you-can” in a variable named “divOfInterest”.
2. Given HTML that includes seven SPAN elements each with the class “cloudy”, write a JavaScript statement that stores a reference to a NodeList filled with references to the seven HTMLSpanElements in a variable named “cloudySpans”.
3. Given an HTML file with HTML, HEAD, TITLE, and BODY elements, create and reference a JS file that in which the JavaScript will create and attach to the BODY element an H1 element with the id "sleeping-giant" with the content "Jell-O, Burled!".
4. Given an HTML file with HTML, HEAD, TITLE, SCRIPT, and BODY elements with the SCRIPT's SRC attribute referencing an empty JS file, write a script in the JS file to create a DIV element with the id "lickable-frog" and add it as the last child to the BODY element.
5. Given an HTML file with HTML, HEAD, TITLE, SCRIPT, and BODY elements with no SRC attribute on the SCRIPT element, write a script in the SCRIPT block to create a UL element with no id, create an LI element with the id "dreamy-eyes", add the LI as a child to the UL element, and add the UL element as the first child of the BODY element.
6. Write JavaScript to add the CSS class "i-got-loaded" to the BODY element when the window fires the DOMContentLoaded event.
7. Given an HTML file with a UL element with the id "your-best-friend" that has six non-empty LIs as its children, write JavaScript to write the content of each LI to the console.
8. Given an HTML file with a UL element with the id "your-worst-enemy" that has no children, write JavaScript to construct a string that contains six LI tags each containing a random number and set the inner HTML property of ul#your-worst-enemy to that string.
9. Write JavaScript to update the title of the document to the current time at a reasonable interval such that it looks like a real clock.

Hello, World DOMination: Element Selection And Placement

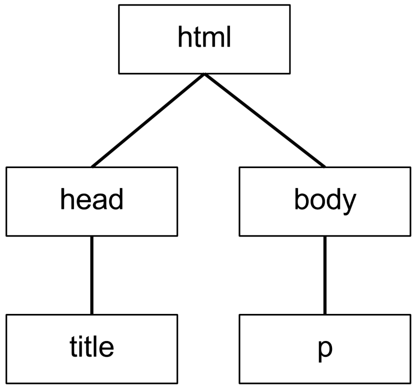
The objective of this lesson is to familiarize yourself with the usage and inner workings of the DOM API. When you finish this lesson, you should be able to:

- Reference and manipulate the DOM via Javascript
- Update and create new DOM elements via Javascript
- Change CSS based on a DOM event
- Familiarize yourself with the console

What is the DOM?

TheDocument Object Model, or DOM, is an object-oriented representation of an HTML document or Web page, meaning that the document is represented as objects, or nodes. It allows developers to access the document via a programming language, like Javascript.

The DOM is typically depicted as a tree with a specific hierarchy. (See the image below.) Higher branches represent parent nodes, while lower branches represent child nodes, or children. More on that later.



Referencing the DOM

The DOM API is one of the most powerful tools frontend developers have at their disposal. Learning how to reference, create, and update DOM elements is an integral part of working with Javascript. We'll start this lesson by learning how to reference a DOM element in Javascript.

Let’s assume we have an HTML file that includes the following `div`:

HTML

```
<div id="”catch-me-if-you-can”">HI!</div>
```

Because we've added the element to our HTML file, that element is available in the DOM for us to reference and manipulate. Using JavaScript, we can reference this element by scanning the document and finding the element by its id with the method document.getElementById(). We then assign the reference to a variable.

Javascript

```
const divOfInterest = document.getElementById(“catch-me-if-you-can”)
```

Now let’s say that our HTML file contains seven `span`elements that share a class name of `cloudy`, like below:

HTML

```
<span class="“cloudy”"></span>
<span class="“cloudy”"></span>
<span class="“cloudy”"></span>
<span class="“cloudy”"></span>
<span class="“cloudy”"></span>
<span class="“cloudy”"></span>
<span class="“cloudy”"></span>
```

In Javascript, we can reference all seven of these elements and store them in a single variable.

Javascript

```
const cloudySpans = document.querySelectorAll("span.cloudy");
```

While `getElementById`allows us to reference a single element,`querySelectorAll`references all elements with the class name “cloudy” as a static `NodeList`(*static*meaning that any changes in the DOM do not affect the content of the collection). Note that a NodeList is different from an array, but it is possible to iterate over a NodeList as with an array usingforEach(). Refer to the MDN doc onNodeListfor more information.

Using `forEach()`on a NodeList:

Javascript

```
const cloudySpans = document.querySelectorAll("span.cloudy");

cloudySpans.forEach(span => {
  console.log("Cloudy!");
});
```

Creating New DOM Elements

Now that we know how to reference DOM elements, let's try creating new elements. First we'll set up a basic HTML file with the appropriate structure and include a reference to a Javascript file that exists in the same directory in the`head`.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
    <script type="text/javascript" src="example.js"></script>
  </head>
  <body></body>
</html>
```

In our example.js file, we'll write a function to create a new `h1`element, assign it an id, give it content, and attach it to the body of our HTML document.

Javascript

```
const addElement = () => {
  // create a new div element
  const newElement = document.createElement("h1");
  // set the h1's id
  newElement.setAttribute("id", "sleeping-giant");

  // and give it some content
  const newContent = document.createTextNode("Jell-O, Burled!");

  // add the text node to the newly created div
  newElement.appendChild(newContent);

  // add the newly created element and its content into the DOM
  document.body.appendChild(newElement);
};
// run script when page is loaded
window.onload = addElement;
```

If we open up our HTML file in a browser, we should now see the words `Jell-O Burled!` on our page. If we use the browser tools to inspect the page (right-click on the page and select "inspect", or hotkeys fn + f12), we notice the new `h1` with the id we gave it.

Hello, World DOMination: Inserting Elements in JS File and Script Block

Let's practice adding new elements to our page. We'll create a second element, a `div`with an id of `lickable-frog`, and append it to the `body`like we did the first time. Update the Javascript function to append a second element to the page.

Javascript

```
const addElements = () => {
  // create a new div element
  const newElement = document.createElement("h1");

  // set the h1's id
  newElement.setAttribute("id", "sleeping-giant");

  // and give it some content
  const newContent = document.createTextNode("Jell-O, Burled!");

  // add the text node to the newly created div
  newElement.appendChild(newContent);

  // add the newly created element and its content into the DOM
  document.body.appendChild(newElement);

  // append a second element to the DOM after the first one
  const lastElement = document.createElement("div");
  lastElement.setAttribute("id", "lickable-frog");
  document.body.appendChild(lastElement);
};
// run script when page is loaded
window.onload = addElements;
```

Notice that our function is now called `addElements`, plural, because we're appending two elements to the `body`. Save your Javascript file and refresh the HTML file in the browser. When you inspect the page, you should now see two elements in the `body`, the `h1`and the `div` we added via Javascript.

Referencing a JS File vs. Using a Script Block

In our test example above, we referenced an external JS file, which contained our function to add new elements to the DOM. Typically, we would keep Javascript in a separate file, but we could also write a script block directly in our HTML file. Let's try it. First, we'll delete the script source so that we have an empty script block.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript">
      //Javascript goes here!
    </script>
  </head>
  <body></body>
</html>
```

Inside of our script block, we'll:

- create a `ul`element with no id
- create an `li`element with the id `dreamy-eyes`
- add the `li`as a child to the `ul`element
- add the `ul`element as the first child of the `body`element.

HTML

```html
<!DOCTYPE html>
<html>
  <head>
    <title>My Cool Website</title>
    <script type="text/javascript">
      const addListElement = () => {
        const listElement = document.createElement("ul");
        const listItem = document.createElement("li");
        listItem.setAttribute("id", "dreamy-eyes");
        listElement.appendChild(listItem);
        document.body.prepend(listElement);
      };
      window.onload = addListElement;
    </script>
  </head>
  <body></body>
</html>
```

Refresh the HTML in your browser, inspect the page, and notice the `ul`and `li`elements that were created in the script block.

Hello, World DOMination: Adding a CSS Class After DOM Load Event

In our previous JS examples, we used `window.onload`to run a function after the window has loaded the page, which ensures that all of the objects are in the DOM, including images, scripts, links, and subframes. However, we don't need to wait for stylesheets, images, and subframes to finish loading before our JavaScript runs because JS isn't dependent on them. And, some images may be so large that waiting on them to load before the JS runs would make the user experience feel slow and clunky. There is a better method to use in this case:`DOMContentLoaded`.

According to MDN, "the DOMContentLoaded event fires when the initial HTML document has been completely loaded and parsed, without waiting for stylesheets, images, and subframes to finish loading."

We'll use DOMContentLoaded to add CSS classes to page elements immediately after they are loaded. Let's add the CSS class "i-got-loaded" to the `body`element when the window fires the DOMContentLoaded event. We can do this in the script block or in an external JS file, as we did in the examples above.

Javascript

```
window.addEventListener("DOMContentLoaded", event => {
  document.body.className = "i-got-loaded";
});
```

After adding the Javascript above, refresh the HTML in your browser, inspect the page, and notice that the `body`element now has a class of "i-got-loaded".

Hello, World DOMination: Console.log, Element.innerHTML, and the Date Object

In this section, we'll learn about how to use `console.log`to print element values. We'll also use `Element.innerHTML`to fill in the HTML of a DOM element. Finally, we'll learn about the Javascript Date object and how to use it to construct a clock that keeps the current time.

Console Logging Element Values

Along with the other developer tools, the console is a valuable tool Javascript developers use to debug and check that scripts are running correctly. In this exercise, we'll practice logging to the console.

Create an HTML file that contains the following:

HTML

```
<!DOCTYPE html>
<html>
  <head> </head>
  <body>
    <ul id="your-best-friend">
      <li>Has your back</li>
      <li>Gives you support</li>
      <li>Actively listens to you</li>
      <li>Lends a helping hand</li>
      <li>Cheers you up when you're down</li>
      <li>Celebrates important moments with you</li>
    </ul>
  </body>
</html>
```

In the above code, we see an id with which we can reference the `ul`element. Recall that we previously used `document.querySelectorAll()`to store multiple elements with the same class name in a single variable, as a NodeList. However, in the above example, we see only one id for the parent element. We can reference the parent element via its id to get access to the content of its children.

Javascript

```
window.addEventListener("DOMContentLoaded", event => {
  const parent = document.getElementById("your-best-friend");
  const childNodes = parent.childNodes;
  for (let value of childNodes.values()) {
    console.log(value);
  }
});
```

In your browser, use the developer tools to open the console and see that the values of each `li`have been printed out.

Using Element.innerHTML

Thus far, we have referenced DOM elements via their id or class name and appended new elements to existing DOM elements. Additionally, we can use the inner HTML property to get or set the HTML or XML markup contained within an element.

In an HTML file, create a `ul`element with the id "your-worst-enemy" that has no children. We'll add some JavaScript to construct a string that contains six `li`tags each containing a random number and set the inner HTML property of`ul#your-worst-enemy`to that string.

HTML

```html
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="example.js"></script>
  </head>
  <body>
    <ul id="your-worst-enemy"></ul>
  </body>
</html>
```

Javascript

```javascript
// generate a random number for each list item
const getRandomInt = max => {
  return Math.floor(Math.random() * Math.floor(max));
};

// listen for DOM ready event
window.addEventListener("DOMContentLoaded", event => {
  // push 6 LI elements into an array and join
  const liArr = [];
  for (let i = 0; i < 6; i++) {
    liArr.push("<li>" + getRandomInt(10) + "</li>");
  }
  const liString = liArr.join(" ");

  // insert string into the DOM using innerHTML
  const listElement = document.getElementById("your-worst-enemy");
  listElement.innerHTML = liString;
});
```

Save your changes, and refresh your browser page. You should see six new list items on the page, each containing a random number.

Inserting a Date Object into the DOM

We've learned a lot about accessing and manipulating the DOM! Let's use what we've learned so far to add extra functionality involving the Javascript Date object.

Our objective is to update the title of the document to the current time at a reasonable interval such that it looks like a real clock.

We know we'll be starting with an HTML document that contains an empty title element. We've learned a couple of different ways to fill the content of an element so far. We could create a new element and append it to the title element, or we could use `innerHTML`to set the HTML of the title element. Since we don't need to create a new element nor do we care whether it appears last, we can use the latter method.

Let's give our title an id for easy reference.

HTML

```html
<title id="title"></title>
```

In our Javascript file, we'll use the Date constructor to instantiate a newDate object.

```javascript
const date = new Date();
```

Javascript

```javascript
window.addEventListener("DOMContentLoaded", event => {
  const title = document.getElementById("title");
  const time = () => {
    const date = new Date();
    const seconds = date.getSeconds();
    const minutes = date.getMinutes();
    const hours = date.getHours();

    title.innerHTML = hours + ":" + minutes + ":" + seconds;
  };
  setInterval(time, 1000);
});
```

Save your changes and refresh your browser. Observe the clock we inserted dynamically keeping the current time in your document title!

What We Learned:

- What the DOM is and how we can access it
- How to access DOM elements using `document.getElementById()` and `document.querySelectorAll()`
- How to create new elements with `document.createElement()` and `document.createTextNode`, and append them to existing DOM elements with `Element.appendChild()`
- The difference between `window.onload` and `DOMContentLoaded`
- How to access children nodes with `NodeList.childNodes`
- Updating DOM elements with `Element.innerHTML`
- The Javascript Date object

Brush Up On Your HTML

You'll be using a lot of HTML in the following days (weeks, months, years), so might as well get a leg up by reacquainting yourself with HTML.

The definitive resource on the Internet for HTML, CSS, and JavaScript is the Mozilla Developer Network. Go there and work through, at a minimum, the following sections:

- The following sections in Introduction to HTML
    - Getting started with HTML
    - What's in the head? Metadata in HTML
    - HTML text fundamentals
    - Creating hyperlinks
- The following section in Multimedia and embedding
    - Images in HTML
- The following section in HTML forms
    - Your first HTML form
    - How to structure an HTML form
    - The native form widgets
    - Client-side form validation
    - Styling HTML forms