The DOM Is a Tree

[(https://github.com/learn-co-curriculum/phase-0-the-dom-is-a-tree)](https://github.com/learn-co-curriculum/phase-0-the-dom-is-a-tree) [(https://github.com/learn-co-curriculum/phase-0-the-dom-is-a-tree/issues/new)](https://github.com/learn-co-curriculum/phase-0-the-dom-is-a-tree/issues/new)

Learning Goals

- Describe how the DOM works as a tree
- Define the computer science version of "Tree"
- Ask the DOM to find or "select" an HTML element or elements in the rendered page

Introduction

DOM programming is using JavaScript to:

1. Ask the DOM to find an HTML element or elements in the rendered page
2. Remove the selected element(s) or add a new element next to the selected element
3. Adjust a property of the selected element(s)

In previous lessons we were given the command to find the HTML element we wanted:

```
document.querySelector(selector);
```

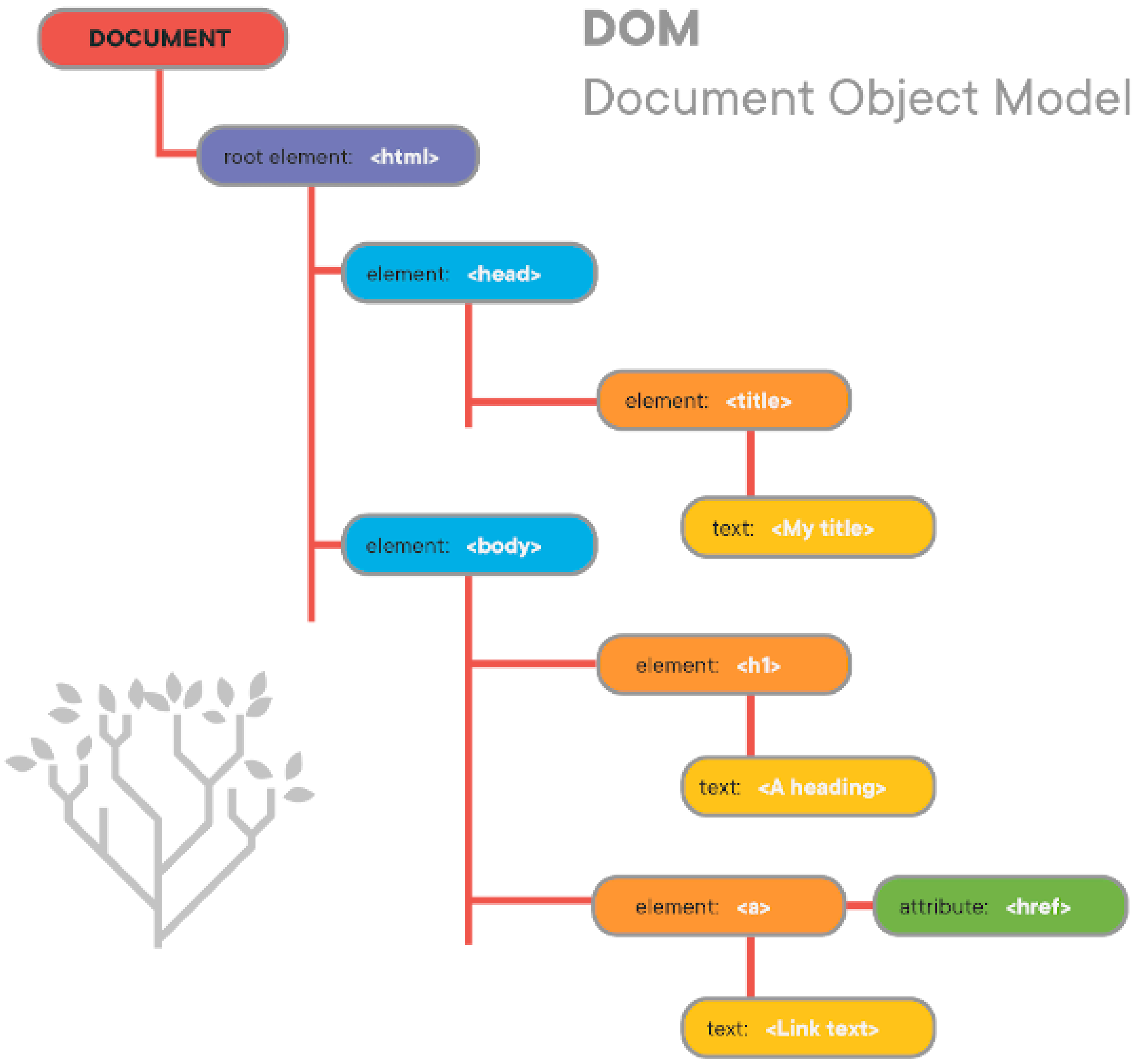
The *selector* is like a query string that lets us find things within an HTML page. What is the syntax of this *selector*? How does the *selector* navigate through our document to find the DOM nodes that we want to work with (update, move, even delete!)?

To understand those queries or *selectors*, we first need to talk about how the DOM tree (i.e. what we see in the 'Elements' panel of our DevTools) is used to help the DOM's `methods` find the right nodes.

Define the Computer Science Version of "Tree"

What do we mean when we say that the DOM is a tree? Trees make a good metaphor for the DOM because almost everyone has seen a tree. Starting at the bottom, you can climb up the tree and out to the farthest — and smallest — branches. The thicker a branch is, the stronger its connections are and the more it holds within it. Likewise, the thinner a branch is, the less it holds inside.

The DOM works basically the same way, except we usually talk about the root as being at the top of the DOM and the leaves being the most deeply nested HTML elements. So basically, we can imagine a tree upside down.



The HTML for this "tree" would be:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Title</title>
  </head>
  <body>
    <h1>A heading</h1>
    <a href="http://example.com">Link text</a>
  </body>
</html>
```

Describe How the DOM Works as a Tree

Every tree can contain subtrees, which we can treat independently of their parent trees. They repeat the pattern and appearance of the full tree, despite being a smaller part of a tree, like branches. Every child has experienced this sense of wonder when they take a fallen branch and stick it in the ground and think that they've planted their own tree.

Practically speaking, the DOM begins at `<html>` , but for now we should avoid changing what's between the `<head></head>` tags. Most of the time, we will look at the DOM subtree with its root at `<body>` and only change things that will be visible on the page. We might also deal with subtrees. For example, if we have

```
<body>
  <div>
    <p>Hi!</p>
  </div>

  <div>
    <p>Bye!</p>
  </div>
</body>
```

Our tree looks like this:

```
         body
         /  \
       div   div
       /       \
      p         p
     /           \
  "Hi!"         "Bye!"
```

Similarly, if we had a DOM subtree that looked like

```html
<div>
  <div>
    <h1>Hello!</h1>
  </div>

  <div>
    <h5>Sup?</h5>
  </div>
</div>
```

The tree would look like:

```
          div
         /  \
       div   div
       /       \
     h1         h5
     /           \
  "Hello!"      "Sup?"
```

Finding HTML Elements

In creating the HTML for a page, including *metadata* for a node (e.g., a `class` or `id` attribute) will not only provide useful information about that node, but will also make it and its children easier to find. The more specific the metadata is, the more helpful it is for finding the desired element.

For the following exercises, you can experiment with any web page you like. It's fun to change *The New York Times* or Facebook.

Finding a Node

JavaScript exposes a few ways of finding DOM nodes, either directly or in stages, courtesy of the `document` object. We will introduce three here, in order from most to least specific: `getElementByID()` , `getElementsByClassName()` , and `getElementsByTagName()` .

`document.getElementById()`

This method provides the quickest access to a node, but it requires that we know a very specific piece of information — its `id` . This method can only return one element, since CSS `id` s are expected to be unique.

Given the following DOM tree:

```
<div>
  <h5 id="greeting">Hello!</h5>
</div>
```

We could find the `h5` element with `document.getElementById('greeting')` .

Notice how the `id` that we pass to `getElementById` is identical to the `id` in `<h5 id="greeting">` .

Note: You can use either single(`''`) or double(`""`) quotes around the `id` within the parentheses in `document.getElementById('yourIDGoesHere')` , as long as you use the same kind to open and close them!

Try it out!

Open up your DevTools and find an element on the page that has an `id` attribute. Then open up your console, type `document.getElementById('theIdOfTheElement')` , and check out your handy-dandy DOM node.

`document.getElementsByClassName()`

This one is also very commonly used in DOM programming.

This method finds elements by their `className` . Unlike the previous method, class names do not need to be unique, so this method returns an `HTMLCollection` of all the elements with the given class. An `HTMLCollection` is an array-like structure containing a list of elements. You can iterate over an `HTMLCollection` with a simple `for` loop.

Given the following DOM tree:

```
<!-- the `className` attribute is called `class` in HTML  -->
<div>
  <div class="banner">
    <h1>Hello!</h1>
  </div>

  <div class="banner">
    <h1>Sup?</h1>
  </div>

  <div class="banner">
    <h5>Tinier heading</h5>
  </div>
</div>
```

We could find all of the elements with the class name "banner" by calling `document.getElementsByClassName('banner')` .

Try it out!

Inspect your web page again, this time making note of a `class` . Get all elements with that `class` and give 'em a look. On the returned object you can use the `.length` property to find out how many came back.

If you recall the `for` loop syntax you might try to write a loop which prints out the `innerHTML` property of every element in the collection. You might find doing so much easier if you save the results of `document.getElementsByClassName()` to a variable:

```
const elements = document.getElementsByClassName("yourClassNameHere");
```

`document.getElementsByTagName()`

You can use this method if you *don't* know an element's `id` or `class` , but you *do* know its tag name (the tag name is the thing between the `<>` , e.g., `div` , `h1` , `header` , `article` etc.). Since tag names aren't unique, this method also returns an `HTMLCollection` .

Try it out!

Explore the DOM in the console by typing `document.getElementsByTagName('div')` . You can iterate through these elements using a simple `for` loop as well.

Finding a Node Without Knowing Anything About It

What if we don't have an `id` or `className` to help us find a particular element? This is where our knowledge of trees comes in handy!

Given the following DOM tree:

```
<main>
  <div>
    <div>
      <p>Hello!</p>
    </div>
  </div>
  <div>
    <div>
      <p>Hello!</p>
    </div>
  </div>
  <div>
    <div>
      <p>Hello!</p>
    </div>
  </div>
</main>
```

How would we go about changing only the second "Hello!" to "Goodbye!"?

Here we're going to use a mix of different `methods` to accomplish the goal.

Let's start by getting the `<main>` element

```
const main = document.getElementsByTagName("main")[0];
```

We can get the children of `main` using `main.children`. This returns an `HTMLCollection`, so we can get the second child with `main.children[1]`.

```
const secondChild = main.children[1];
```

Next, we can get our `<p>` element. To constrain the search to just the children of the second child, we can call `getElementsByTagName()` **directly on `secondChild`**:

```
const p = secondChild.getElementsByTagName("p")[0];
```

And lastly we can change an attribute on the `p` node:

```
p.textContent = "Goodbye!";
```

Obviously, this way of accessing that text isn't very efficient and won't work on all pages but it does a good job of demonstrating the basic tools available to us for finding and manipulating HTML elements.

Conclusion

Understanding the tree structure of the DOM helps us navigate all kinds of trees. In subtrees and branches we can find the nodes we need by IDs, class names or tag names, or by using element attributes like `children`. Once we've selected our elements, we can use JavaScript to manipulate them. By using these techniques, we can start to build a richer user experience.

Resources

- **MDN - Document Object Model (https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model)**