JavaScript DOM Manipulation

- One of the most powerful things we can do with JS is access and manipulate the DOM.
    - Remember that the DOM (Document Object Model) is just the browser's in-memory representation of a web page.

- JS provides a number of mechanisms for working with the DOM, and there are two principal classes we want to be aware of when working with the DOM:
    - **`Node`** – a generic node in the DOM tree
        - This includes nodes representing text and other things
    - **`Element`** – a node in the DOM tree specifically representing an HTML element
        - `Element` inherits from `Node`.
        - There are also several subclasses of `Element` that represent specialized HTML elements, like links, images, input fields, etc.

- These two classes will provide us with many of the methods and properties we'll use to access and modify the DOM.

Node properties for stepping around the DOM

- One of the most basic ways we can navigate through the DOM is to move from one known node to an adjacent node, for example moving from a parent node to one of its children, or moving to a sibling of a particular node.

- To be able to navigate in this way, we need a starting point. One nice starting point that JS provides us with is a global object representing the entire HTML document. Sensibly, this object is called `document`.

- Right off the bat, the `document` object has two properties that we can use to jump into the DOM:
    - **`document.documentElement`** – gives access to the `<html>` element.
    - **`document.body`** – gives access to the `<body>` element.

- Once we have hold of a particular node (like `document.body`), it gives us several properties that we can use to jump to adjacent nodes in the DOM:
    - **`childNodes`** – an array of all of a Node's child Nodes
    - **`firstChild`** – a Node's first child Node
    - **`lastChild`** – a Node's last child Node
    - **`nextSibling`** – a Node's next sibling Node
    - **`previousSibling`** – a Node's previous sibling Node
    - **`parentNode`** – a Node's parent Node

- For example, we could obtain and loop through an array of all of the children of the `<body>` element like this:

```
var bodyChildren = document.body.childNodes;
bodyChildren.forEach(function (child) {
    // Do something with the child node here.
});
```

- The properties above are great if we already have a hold of a particular DOM node and want to step around to nearby nodes. However, while it is possible to use these properties to find arbitrary nodes in the DOM, it would be pretty difficult to do so.

An easier way to find elements

- Luckily, JS provides us a set of functions that makes it much easier to find specific nodes in the DOM based on different HTML attributes (e.g. class or ID) of those elements. We'll look at 5 of these functions.
    - To use all of the functions we'll explore here, we need some node as a starting point, since each of the functions finds nodes that are *descendants* of a particular node. Often, we'll use the `document` node as the starting point from which to call these functions, since *all* nodes are descendants of `document`.

- The first function we'll look at is `getElementById()`, which allows us to find any single node in the DOM based on the `id` attribute assigned to it in the HTML.

- For example, let's say we had HTML that contained this element:

```
<button id="my-button">Click me!</button>
```

- We could access that button in the JS like this:

```
var myButton = document.getElementById('my-button');
```

- Another important function for accessing arbitrary elements in the DOM is `getElementsByClassName()`, which allows us to find *all* nodes in the DOM that have a specified `class` value.

- For example, let's say we have several images in our HTML, some of which have a given class and some of which do not, something like this:

```
<img class="special-image" src="..." />
<img src="..." />
<img class="special-image" src="..." />
<img src="..." />
```

- In this case, we could access all of the images with the `special-image` class as follows:

```
var specialImages =
    document.getElementsByClassName('special-image');
```

- Here, `specialImages` will be an array-like object that contains all of the images with the `special-image` class (and *not* the images who don't have that class). We could loop through the elements in this array, for example, like this:

```
for (int i = 0; i < specialImages.length; i++) {
     // Do something with specialImages[i].
}
```

    - Importantly, the array-like object returned by `getElementsByClassName()` is *live* and is automatically updated as the DOM changes!
        - Specifically, if you add elements to the DOM with the same class or remove one or more of them from the DOM, *those objects will be automatically added/removed to/from the "array".*

- Note that `getElementsByClassName()` uses only a *single* class value, whereas HTML elements can be assigned multiple class values.

    - When operating on elements with multiple class values, `getElementsByClassName()` just checks to see if the specified class is among *any* of the class values for a given node.

- For example, say we have the following HTML, where there are several `<li>` elements that share the common class `navbar-item` but some of which also have one or more different class values as well:

```
<ul>
    <li class="navbar-item navbar-title">...</li>
    <li class="navbar-item navbar-menu">...</li>
    <li class="navbar-item">...</li>
    <li class="navbar-item navbar-right">...</li>
</ul>
```

- In this case, we could *still* access all of these `<li>` elements with `getElementsByClassName()`:

```
var navbarItems =
    document.getElementsByClassName('navbar-item');
```

- A lesser-used but still-important function for grabbing collections of elements is `getElementsByTagName()`. This function grabs all elements in the HTML that have a certain *tag* (e.g. `<a>`, `<li>`, `<img>`, etc.).

- For example, if we had several images in the HTML as in the example above, we could grab *all* of them (both with and without the `special-image` class) as follows:

```
var allImages = document.getElementsByTagName('img');
```

- Here, `allImages` would be the same kind of array-like object that's returned by `getElementsByClassName()`.
    - Again, this "array" is *live* and is automatically updated as the DOM changes.

- Sometimes, we need to select elements in such a way that is too complex to represent using a single ID, class, or tag. For these situations, JS provides us

with two functions, `querySelector()` and `querySelectorAll()`, that allow us to select elements using arbitrary CSS-like selectors.
    - `querySelector()` selects the first element matching the specified selector, while `querySelectorAll()` selects all matching elements.

- For example, let's say we have two lists in our HTML, set up like this:

```
<ul id="navbar-list">
    <li>...</li>
    <li>...</li>
    <li>...</li>
    <li>...</li>
</ul>

<ul>
    <li>...</li>
    <li>...</li>
    <li>...</li>
    <li>...</li>
</ul>
```

- Now, let's say we want to select only the `<li>` elements within the `navbar-list`. Because they have no class applied to them, we can't use `getElementsByClassName()`. However, using `getElementsByTagName()` would give us the elements from the second list as well.

- In this case, we can use `querySelectorAll()` to give us *only* the `<li>` elements that are descendants of the `navbar-list`:

```
var navbarItems =
    document.querySelectorAll('#navbar-list li');
```

- Again, this will give us a *live* array-like object similar to the one we get from `getElementsByClassName()` and `getElementsByTagName()`.

- As noted above, all of the functions we've been examining here can be used to find descendants of *any* node, not just `document`. For example, an alternative way to access only the `<li>` elements within the `navbar-list` in the example above would be to use a combination of `getElementById()` and

`getElementsByTagName()`, like so:

```
var navbarList = document.getElementById('navbar-list');
var navbarItems = navbarList.getElementsByTagName('li');
```

The properties of DOM elements

- Once we have found the DOM elements we're interested in using the methods described above, we'll to be able to use those elements somehow. Often, we'll want to read or modify the properties of these elements. There are many element properties we can access, some of which we can modify, and we'll explore those here.

- The very first element property we'll explore is also possibly the most dangerous: `innerHTML`. This property gives us access to the *HTML* content of an element and can be both read and written to.

- For example, let's say we have a little `<div>` element in our HTML that contains an paragraph:

```
<div id="content">
    <p>This is a paragraph.</p>
</div>
```

- In JS, we could access that element and read its HTML content like so:

```
var content = document.getElementById('content');
console.log(content.innerHTML);
```

- This would print "`<p>This is a paragraph.</p>`".

- For now, just remember that writing to `innerHTML` can be very dangerous. We'll see why in a second.

- Another node property available for accessing and setting content is `textContent`, which provides only the text contained within a node and any of its descendants. For example, we could read the text content of our `<div>` from above like so:

```
console.log(content.textContent);
```

- This would print "`This is a paragraph.`".

- You can also write to `textContent`, and unlike `innerHTML`, it is basically safe to do so, since content written to `textContent` is not parsed as HTML. One caveat, though, is that writing to `textContent` for a node will eliminate any existing HTML within that node.

- For example, if we wrote to `textContent` of our `<div>` above, it would eliminate the nested `<p>` element:

```
content.textContent = "Hello";
```

- This would leave us with essentially the following HTML:

```
<div id="content">Hello</div>
```

- Some specialized elements have unique properties representing their specialized HTML attributes. For example, in JS, an element corresponding to an `<a>` element in the HTML will have an `href` property, an `<img>` element will have a `src` property.
    - These properties can be both read and written to.

- Similarly an element corresponding to an input field (e.g. `<input>` or `<textarea>`) will have a `value` property that we can use to access (or modify) the value the user entered into the field.

- There are other properties that we can use to determine the size of an element in the page. There are three sets of width and height properties, and to understand the differences between all of these it's important to remember the following things:
    - An element is typically composed of its content, its padding, its border, and its margin.
    - If the element's content overflows the element and is allowed to scroll, a scroll bar can also be added inside the element's border.
    - It the element's content overflows and is allowed to scroll, the size of the content is larger than the size of the displayed element.

- With those facts in mind, the three sets of size properties each element has are:
    - **`clientWidth`**, **`clientHeight`** – these give the width and height of an element's *visible* content (i.e. not content scrolled out of view) including its padding.
    - **`offsetWidth`**, **`offsetHeight`** – these give the width and height of the entire visible extent of an element, including visible content, padding, scrollbars, and borders, but excluding margins.
    - **`scrollWidth`**, **`scrollHeight`** – if some of an element's content is scrolled out of view, these give the entire size of the all of the element's content. Otherwise, they just give the size of the visible content.

- We can access these properties like so (using our `<div>` element from above):

```
console.log(content.clientWidth, content.clientHeight);
```

- Related to `scrollWidth` and `scrollHeight` are `scrollLeft` and `scrollTop`. These provide, respectively, the number of pixels an element's content has been scrolled leftwards and upwards.
    - Related again are the properties `window.scrollX` and `window.scrollY`, which respectively provide the number of pixels the entire *document* has been scrolled horizontally and vertically within the browser window.

- Another way to get size and position information about an element is to use `getBoundingClientRect()`, which returns an object representing the rectangle containing the element within the viewport, e.g.:

```
console.log(content.getBoundingClientRect());
```

Creating and inserting new DOM content

- There are a few ways we can create new content and insert it into the DOM.

- Above, we mentioned that you *could* write to `innerHTML` to insert content into the DOM, but that doing so was dangerous.

- Writing to `innerHTML` is particularly dangerous when we don't know exactly what's contained in the value we're writing to `innerHTML`, like when we have a value that was provided directly by the user.

- For example, let's say that we've got an input field on our page that the user can put a value into to have us plug into the page somewhere (e.g. writing a tweet in Twitter), and let's say they enter a particularly sneaky/malicious value. We can simulate this by storing such a value into a variable:

```
var userSuppliedValue =
    "<img src=x onerror=\"alert('Uh oh.')\" >";
```

    - There's a lot going on here that we haven't gotten to yet, but the gist is that the user is entering HTML code for an `<img>` element. The element has an attribute that specifies a JS function to run if there is an error loading an image, and it also has a `src` value that is specifically chosen to produce an image loading error.

- If we plug this into our page directly using `innerHTML`, it will result in the user's JS code being run (try all of this in your browser's JS console to see what happens):

```
var content = document.getElementById('content');
content.innerHTML = userSuppliedValue;
```

- This is ***BAD***. In this case, the user specified JS code that didn't do much evil, but you can imagine them doing worse. This method for exploiting sites has a name: ***cross-site scripting*** (or ***XSS***).

- We need to protect against XSS at all costs, which is why `innerHTML` should only be written to when you are 101% positive that the value being written to it could not possibly contain any unescaped user-supplied content.

- Thankfully, there are other, safer methods for incorporating new content into the DOM.

- The first method we'll look at for creating content is `document.createElement()`. This creates an empty HTML element of a specified type. For example, we could create an empty `<div>` like this:

```
var newDiv = document.createElement('div');
```

- An empty `<div>` doesn't do us much good, though. We want to be able to add content and modify HTML attributes.

- To create simple text content, we can use `document.createTextNode()`. For example, we could create text to put into our new `<div>` like this:

```
var divText =
    document.createTextNode("This is our div's content.");
```

- Note, though that this just creates a DOM note that contains text. It doesn't do anything with that node. If we wanted to insert the new text node into our new `<div>`, we would use `appendChild()`:

```
newDiv.appendChild(divText);
```

- If we now examined our new `<div>`, we'd see something like this:

```
<div>This is our div's content.</div>
```

- We could go further, adding classes to our new `<div>` using its `classList` property:

```
newDiv.classList.add('container');
newDiv.classList.add('cat-container');
```

- If we wanted to remove a class, we could do that too:

```
newDiv.classList.remove('container');
```

- We can also set another arbitrary HTML attribute with `setAttribute()`. For example, if we wanted to add an ID, we could do this:

```
newDiv.setAttribute('id', 'main-cat-container');
```

- We can also apply inline styles using the `style` property:

```
newDiv.style.color = 'orange';
newDiv.style.border = '2px solid purple';
```

- Note that using the `style` property to apply styles is best avoided when possible. A better practice, if you know in advance what styles you'll want to apply programmatically in the JS, is to encode those styles into a class-based rule in your CSS and then to add that class to the element using its `classList`.

- For example, instead of applying the styles above directly, we could put this in our CSS:

```
.ugly {
    color: orange;
    border: 2px solid purple;
}
```

- And then we could programmatically add that class to our element:

```
newDiv.classList.add('ugly');
```

- Often, we'll want to build more complicated elements, for example by nesting them within each other. We can do this similar to the way we inserted text into an element above.

- For example, if we wanted to create a `<div>` that contained a `<p>`, which itself contained some text, we could do this:

```
var newP = document.createElement('p');
var text = document.createTextNode("Here we'll see a
cat.");
var newDiv = document.createElement('div');
newP.appendChild(text);
newDiv.appendChild(newP);
```

- This would give us HTML that looked something like this:

```
<div>
    <p>Here we'll see a cat.</p>
</div>
```

- If we further wanted to add an image within the `<div>`, we could do so like this:

```
var newImg = document.createElement('img');
img.src = "http://placekitten.com/320/320/";
newDiv.appendChild(newImg);
```

- That would give us this HTML:

```
<div>
    <p>Here we'll see a cat.</p>
    <img src="http://placekitten.com/320/320/" />
</div>
```

- Importantly, the DOM elements we've been creating are not automatically added to our page. We need to explicitly find the place where we want to add them and insert them there.

- For example, if we wanted to insert our new `<div>` at the end of the `<body>` element, we could do this:

```
document.body.appendChild(newDiv);
```

- If we wanted to insert our element within an element nested somewhere within the `<body>`, we could use one of the methods we saw above to find the element into which we wanted to insert our new element and insert it there, like this:

```
var parentElement = document.findElementById('some-id');
parentElement.appendChild(newDiv);
```

- There are methods other than appendChild() for inserting an element into the DOM, as well. For example, we could use `insertBefore()`:

```
parentElement.insertBefore(
    newDiv,
    parentElement.firstChild
);
```

- Or, we could use `replaceChild()`:

```
parentElement.replaceChild(
    newDiv,
    parentElement.firstChild
);
```

- Finally, if we wanted to remove a node, we could use its parent's `removeChild()` method:

```
newDiv.parentNode.removeChild(newDiv);
```

- Alternatively, most modern browsers support the `remove()` method:

```
newDiv.remove();
```