**Week 5 Notes** – JavaScript and Document Object Model (DOM)

**THE DOCUMENT OBJECT MODEL**

* DOM was created as an interface to view or change the state and content of a web page

What Can Be Done with the DOM?

*Access Nodes:*

The main purpose of DOM is to access nodes (i.e. elements) in an HTML doc. This will allow us to view or change the content of a mode.

*Document Traversal*:

Traverse the tree of elements representing a page. Could be used to traverse down a list of nested list to find the deepest list and change its style.

If you inspect element on a webpage you will be looking at the DOM interface. Essentially, it is the way a browser exposes a page to access and modify it.

**DOM NAVIGATION**

Trees

*Nodes:*

A tree is made from nodes, each node has one parent, though a node can have an unlimited number of children. A node w/o children is called a *leaf* (leaves are found at the tips of branches… haha). In HTML trees, the order of siblings matters because it determines which get rendered first (usually higher) on the page.

Trees all have a single root node. If you start from any node and continue to visit the parents of the nodes you end up on, you will end up at the root. No cycles in trees, a node cannot have itself as a parent.

Navigating a Tree

*Common Movements*:

Move up to a parent node (only 1 parent). Move to a child node (can be multiple children), have to get a list of children then find the one in that list to move to. Move to a sibling node (more complex), have to move to the parent node since siblings don’t contain references to each other.

*Traversing the DOM*:

The DOM exposes a parent node interface for any node which has children. Has the following properties:

firstElementChild – First child node which is also an element node. Ignores text nodes.

lastElementChild – last child node which is also an element node

nextElementSibling – contains the next node, which is both an element and a child of the same parent as | current node

previousElementSibling – contains previous node, which is both an element and a child of… etc

child – contains all the children which are element nodes

The functions should allow for functional navigation of the entire structure of an HTML document.

Searching for Nodes

Could find a node by traversing the whole tree and looking for it, but there’s better ways

*getElementById:*

Element IDs are unique, so this will always return a single element. The function accepts a string of the ID of the element and returns that element. If you have more than one element with the same ID there is no guarantee what will happen. Just avoid having more than one element with the same ID.

*getElementsByTagName:*

Returns an array-like collection of nodes with a particular tag name. e.g.

document.body.getElementsByTagName("div"); will return a collection of all divs in the body of the document.

*getElementsByClassName*:

Provide a class name. Called from an element node. e.g. to find everything in a page with “warning” class, call via document.body.getElementsByClassName("warning");

**DOM NODES**

Node Contents

*textContent*

Contains all the content of a node in string representation. Setting a node’s textContent to “” will clear out all of its text and child nodes, so be careful because you can accidentally delete a lot of your pages content

*Style*

Lets you change the style of a node. With JavaScript this can be used to make a page visually interactive, and the *style* attribute is a JavaScript object. Property names of the object correspond to style names e.g.

myNode.style.color = "red"; will style myNode to have red text.

*className*

Element’s class name, stored in this property. You can append an additional to an element with something like: += " newClass"

Adding Nodes

Adding Text (Parsed or Otherwise) ----

Eloquent JavaScript (Textbook) uses createTextNode to add text to elements. You can also use textContent, which replaces all the content of a node with text not parsed as HTML. Can update content with text that will be parsed as HTML via innerHTML, e.g:

var displayMe = "<span>Hello!</span>";

document.getElementById('html').innerHTML = displayMe;

document.getElementById('textContent').textContent = displayMe;

textContent displays the span tags whereas innerHTML parses them so they are added to DOM and not displayed

Adding Elements ----

*Create Element*

To create elements, use document.createElement , e.g:

var addMe = document.createElement("div");

creates a new div held by the addMe variable (though it does not exist in the document yet)

*Insert Element*

A common way is via appendChild. This adds the node as the last child of the node the method was called from. E.g. :

document.getElementById("bigTable").appendChild(addMe);

would append the addMe element as the last child of the element with the id ‘bigTable’

*Removing Element*

Need to get reference to the element and its parent, then call removeChild from the parent node and pass the node you want to remove as an argument.

**DOM EVENTS**

Event driven programming! These do stuff when the user does stuff (more than just enter in a number or something).

Click Events

Adding and Removing ---

The most common event is the click event, fired when the mouse is pressed and released on the same element.

Generally, event handling does not get more difficult than this. Functions bound to buttons can get more complex but in terms of handlers, this is really the most complex.

Propagation and Defaults ---

An event will propagate up the DOM tree until it hits the root. This can be prevented by calling the stopPropagation method of the event in question.

The other method to be aware of is the preventDefault method. This prevents the default action from happening. e.g. a click handler attached to a link calling preventDefault will stop the browser from following that link. Another example is on the submit button of forms so that the browser doesn’t refresh when a form is submitted.

Load Events

The ‘load’ event is called when a resource and all of its dependent resources are done loading. If you want JS to use some resource as soon as it’s ready, you can attached the ‘load’ handler.

Common mistake is to attach it to the window’s ‘load’ event, which requires everything including large images to load before it is fired. Related event: ‘DOMContentLoaded’ is often where you want to put stuff that needs to manipulate the DOM as soon as the page is loaded. e.g.

<script>

document.addEventListener("DOMContentLoaded", function(event) {

console.log("DOM fully loaded and parsed");

});

</script>

The Event Object

You can pass an event to an event handler by giving the handler an argument in its declaration, e.g.

function someEventHandler(myEvent){...} allows the event that triggered it to be accessed via myEvent. Within that function you could call myEvent.preventDefault; to prevent the default action for that event.