DOM Manipulation

Lecture 5 – CPEN400A

Based on the book “JavaScript: The Definitive Guide”, David Flanagan, O’Reilly

Recap: Last Lecture

- Window object
- Timeouts and Intervals
- Event handling
- Event propagation through DOM

Outline

- **Selecting DOM elements**
- Traversing the DOM structure
- Modifying DOM elements
- Adding/creating new DOM elements

NOTE

- We'll be using the native DOM APIs for many of the tasks in this lecture
- Though many of these can be simplified using frameworks such as jQuery, it is important to know what's "under the hood"
- We assume a standards compliant browser !

Motivation: Selecting elements

- You can access the DOM from the object window.document and traverse it to any node
- However, this is slow – often you only need to manipulate specific nodes in the DOM
- Further, navigating to nodes this way can be error prone and fragile
  - Will no longer work if DOM structure changes
  - DOM structure changes from one browser to another

Three Methods to Select DOM elements

- With a specified id
- With a specified tag name
- With a specified class
- With generalized CSS selector

Method 1: getElementById

- Used to retrieve a **single** element from DOM
  - IDs are unique in the DOM (or at least must be)
  - Returns null if no such element is found

- Example:

```
var name = “Section1”;
var id = document.getElementById(name);
if (id == null)
    throw new Error(“No element found: “ + name);
```

Method 2: getElementsByTagName

- Retrieves multiple elements matching a given tag name (‘type’) in the DOM
- Returns a *read-only* array-like object (empty if no such elements exist in the document)
- Example: Hide all images in the document

```
var imgs = document.getElementsByTagName(“img”);
for (var i=0; i<images.length; i++) {
    imgs[i].display = “none”;
}
```

Method 3: getElementsByClassName

- Can also retrieve elements that belong to a specific CSS class
  - More than one element can belong to a CSS class

- Example:

```
var warnings = document.getElementsByClassName(“warning”);
if (warnings.length > 0) {
        // do something with the warnings list here
}
```

Important point: Live Lists

- Both getElementsByClassName and getElementsByTagName return live lists
  - List can change after it is returned by the function if new elements are added to the document
  - List cannot be changed by JavaScript code adding to it or removing from it directly though

- Make a copy if you're iterating thro' the lists

Selecting elements by CSS selector

- Can also select elements using generalized CSS selectors using querySelectorAll() method
  - Specify a selector query as argument
  - Query results are not “live” (unlike earlier)
  - Can subsume all the other methods

- querySelector() returns the first element matching the CSS query string, null otherwise

CSS selector syntax: Examples (Recap)

#nav // Any element with id=nav

div // Any element

.warning // Any element with “warning” class

#log span // Any descendant of id=“log”

#log > span // Any span child element of id=“log”

body>h1:first-child // first <h1> child of <body>

div, #log // All div elements, element with id=“log”

Invocation on DOM subtrees

- All of the above methods can also be invoked on DOM elements not just the document
  - Search is confined to subtree rooted at element
- Example: Assume element with id="log" exists

```
var log = document.getElementById("log");
var error = log.getElementsByClassName("error");
if (error.length ==0) { … }
```

Class activity

- Write a function that takes two arguments, ‘id’ and ‘interval’ and rotates the images rooted at the node with ID=‘id, and does this periodically every interval milli-seconds

```
function changeImage(id, interval) {


}
```

Outline

- Selecting DOM elements
- **Traversing the DOM structure**
- Modifying DOM elements
- Adding/creating new DOM elements

Traversing the DOM

- Since the DOM is just a tree, you can walk it the way you'd do with any other tree
  - Typically using recursion

- Every browser has minor variations in implementing the DOM, so should not be sensitive to such changes
  - Traversing DOM this way can be fragile

Properties for DOM traversal

- *parentNode*: Parent node of this one, or *null*
- *childNodes*: A read only array-like object containing all the (live) child nodes of this one
- *firstChild, lastChild:* The first and lastChild of a node, or *null* if it has no children
- *nextSibling, previousSibling*: The next and previous siblings of a node (in the order in which they appear in the document)

Other node properties

- nodeType: ‘kind of node’
  - Document nodes: 9
  - Element nodes: 1
  - Text nodes: 3
  - Comment node: 8
- nodeValue
  - textual content of Text of comment node
- nodeName
  - Tag name of a node, converted to upper-case

Example: Find a text node

- We want to find the DOM node that has a certain piece of text, say *'text'*
- Return true if text is found, false otherwise
- We need to recursively walk the DOM looking for the text in all text nodes

Code of the example

```
function search(node, text) {
	var found = false;
	if (node.nodeType==3) {
		if (node.nodeValue === text) found = true;
	} else { // textNodes cannot have children
		var cn = node.childNodes;
		if (cn) {
			for (var i=0; i < cn.length; i++) {
				found = found || search(cn[i], text);
			}
		}
	}
	return found;
};

var result = search(window.document, “Hello world!”);
```

Class activity

- Write a function that will traverse the DOM tree rooted at a node with a specific ‘id’, and checks if any of its sibling nodes and itself in the document is a text node, and if so, concatenates their text content and returns it.

- Can you generalize it so that it works for the entire subtree rooted at the sibling nodes ?

Outline

- Selecting DOM elements
- Traversing the DOM structure
- **Modifying DOM elements**
- Adding/creating new DOM elements

Modifying DOM elements

- DOM elements are also JavaScript Objects (in most browsers) and consequently can have their properties read and written to
  - Can extend DOM elements by modifying their prototype objects
  - Can add fields to the elements for keeping track of state (E.g., visited node during traversals)
  - Can modify HTML attributes of the node such as width etc. – changes reflected in browser display

Element interface

- It is bad practice to modify the Node object directly, so instead JavaScript exposes an *Element* interface. Objects that implement the *Element* interface can be modified
  - Hierarchy of Element objects e.g., HTMLTextElement, HTMLdivElement
  - Element object derives from Node object and has access to its properties

Example: Changing visible elements of a node

- Assume that you want to change the URL of an image object in the DOM with id="myimage" after a 5 second delay to "newImage.jpg"

```
var myImage =
document.getElementById("myimage");
setTimeout( function() {
                    myImage.src ="newImage.jpg";
        }, 5000 );
```

Example: Extending DOM element's prototype

- Let's add a new print method to Node that prints the text to console if it's a text/comment node
  - This may break some frameworks, so proceed with caution !

```
Element.prototype.print = function() {
    if (this.nodeType==3 || this.nodeType==8)
            console.log( this.nodeText );
}
```

Example: Adding new attributes to DOM elements

- You can also add new attributes to DOM nodes, but these will not be rendered by the web browser (unless they're HTML attributes)
  - Caution: may break frameworks such as jquery !

```
var e = document.getElementById("myelement");
e.accessed = true;
// accessed is a non-standard HTML attribute
```

Accessing the raw HTML of a node

- You can retrieve the raw HTML of a DOM node using it's innerHTML property
  - can modify it from within JavaScript code, though this is considered bad practice and is deprecated

HTML: I am a paragraph.

JS code: var e = document.getElementById("myP");

console.log(e.innerHTML);

e.innerHTML = "Don't do this !";

document.write

- This also deprecated
- Quick and dirty method to insert a string into the document at the location of the script that invoked it while parsing the document

- Cannot be used within callback functions or event handlers – will replace the page's DOM

Class activity

- Add a field to each DOM element of type ‘div’ that keeps track of how many times the div is accessed through the document.getElementById method, and make sure to initialize the value of this field for all div’s in the document to 0 when the document is initially loaded.

Outline

- Selecting DOM elements
- Traversing the DOM structure
- Modifying DOM elements
- **Adding/creating new DOM elements**

Creating New DOM Nodes

- You can create a new DOM node using either document.createElement(“element”) OR document.createTextNode(“text content”)

```
var newNode = document.createTextNode(“hello”);
var elNode = document.createElement(“h1”);
```

Copying existing DOM nodes

- To copy an existing DOM node, use *cloneNode*
  - Single argument can be true or false
  - True means make a deep copy (i.e., recursively copy all the descendants)
  - new node can be inserted into a different document

```
var existingNode = document.getElementByid(“my”);
var newNode = existingNode.cloneNode( true );
```

Inserting nodes

- *appendChild* adds a new node as a child of the node it is invoked on. node becomes *lastChild*
- *insertBefore* is similar, except that it inserts the node before the one that is specified as the second argument (*lastChild* if it's null)

```
var s = document.getElementByID("my");
s.appendChild(newNode);
s.insertBefore(newNode,  s.firstChild);
```

Removing and replacing nodes

- Removing a node is done by *removeChild.* To remove a node n, you've got to call

n.parentNode.removeChild(n);

To replace a node 'n' with a new node, do

n.parentNode.replaceChild(

document.createTextNode("[redacted]"),

n);

Example to put it all together

```
// function to replace a node n by making it a child of a new ‘div’ element with id = “id”
function newdiv(n, id) {
    var div = document.createElement(‘div’);
    div.id = id;
    n.parentNode.replaceChild(div, n);
    div.appendChild(n);
};
```

Using Document Fragments

- A document fragment is a container for other nodes
  - Parent node of a fragment is always *null*
  - Can have children like regular DOM nodes

- Can be passed in place of node to *insertBefore, appendChild, replaceChild etc.* for operating on the children of the fragment (not the fragment itself)

```
var frag = document.createDocumentFragment();
```

Example of using Fragments

- Function to reverse order of children of node ‘n’

```
function reverse(n) {
    var f = document.createDocumentFragment();
    while (n.lastChild) {
            var c  =n.lastChild;
            n.removeChild(c);
            f.appendChild(c);
    }
    n.appendChild(f);  // move children of f back to n
}
```

Class Activity

- Write a method to collect all elements that match a specific critereon in the DOM subtree rooted at a node ‘n’ determined by a function f – move these nodes en-masse as children of the parent node of ‘n’ (i.e., as siblings of n)

Outline

- Selecting DOM elements
- Traversing the DOM structure
- Modifying DOM elements
- **Adding/creating new DOM elements**