Lecture 5: DOM Manipulation

CPEN322 - Building Modern Web Applications - Winter 2021-1

Karthik Pattabiraman

The University of British Columbia
Department of Electrical and Computer Engineering
Vancouver, Canada

Electrical and Computer Engineering

Thursday September 30, 2021

Outline

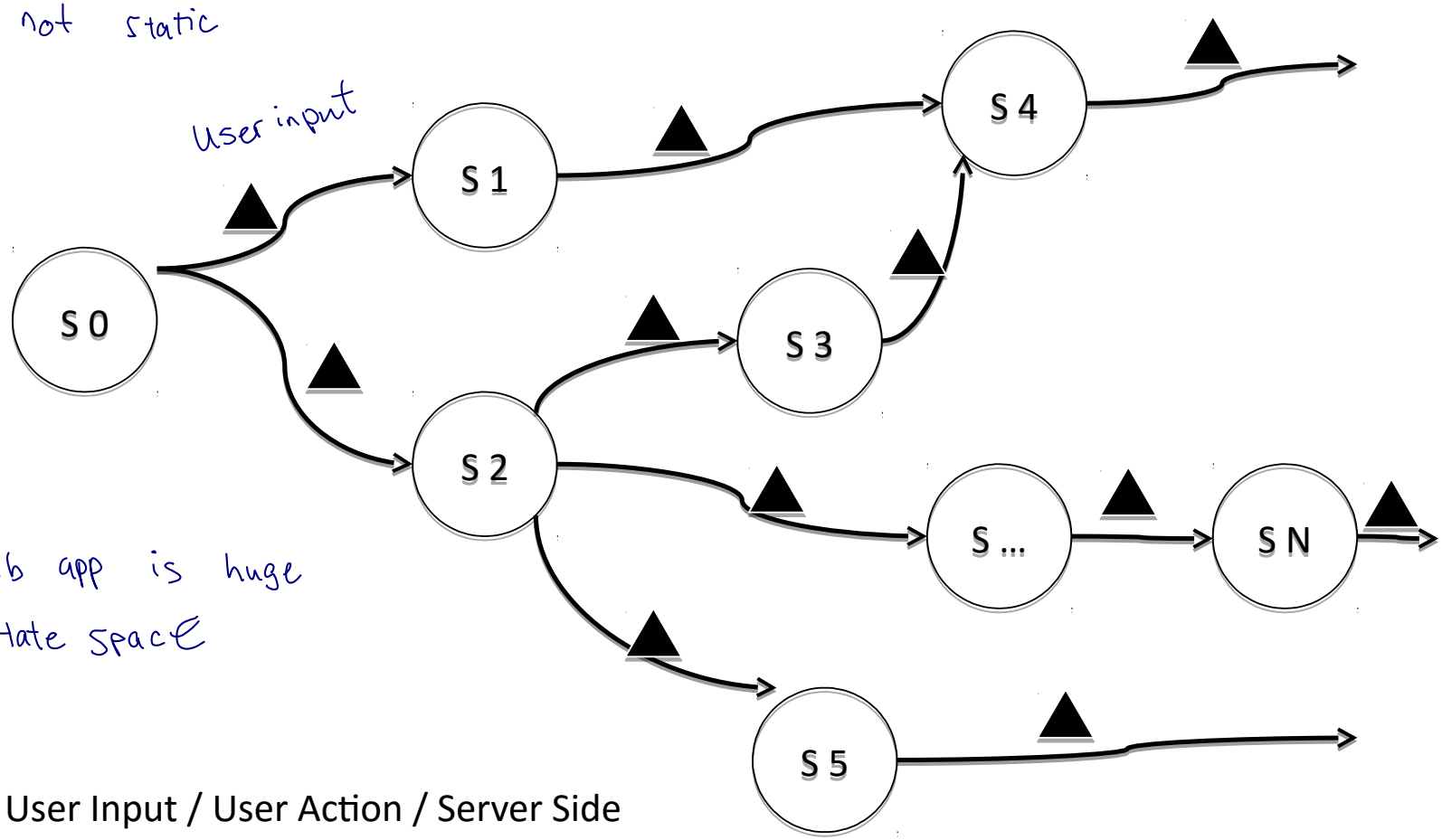
DOM: Recap

- Hierarchical representation of the contents of a web page – initialized with static HTML
- Can be manipulated from within the JavaScript code (both reading and writing)
- Allows information sharing among multiple components of web application

DOM as an evolving entity

DOM is highly dynamic!

- State machine.
- not static



- Web app is huge State space

User Input / User Action / Server Side

Why Study DOM Interactions?

- Needed for JS code to have any effect on webpage (without reloading the page) → HTML
- Uniform API/interface to access DOM from JS → Standard for most protocols
- Does not depend on specific browser platform

NOTE

- We'll be using the native DOM APIs for many of the tasks in this lecture – No other framework / lib.
- 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 !

↳ mostly today, not before.

Selecting DOM elements

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

- You hardcode structure of DOM
- Not portable if any ch.

Methods to Select DOM Elements

- With a specified id
- With a specified tag name
- With a specified class
- With generalized CSS selector → Most general

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

```
1  var name = "Section1";
2  var id = document.getElementById(name);
3  if (id == null)
4     throw new Error("No element found: " + name);
```

- You should check if return null
- treat as any JS object

Method 2: *getElementsByTagName*

But things inside can be.

- 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) → cannot add to doc.

Example: Hide all images in the document

→ get all imgs

```
1  var imgs = document.getElementsByTagName("img");
2  for (var i=0; i<images.length; i++) {
3      imgs[i].display = "none";
4  }
```

← iterate over array

act as JS object

hide image - em

Method 3: *getElementsByClassName*

- R|O.

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

Example

```
1  var warnings = document.getElementsByClassName("
       warning");
2  if (warnings.length > 0) {
3     // do something with the warnings list here
4  }
```

Check arr is not empty

Important point: Live Lists

→ ptr to all elements w/ tag / class

- 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 (list elements can change though)
- Make a copy if you're iterating through the lists (and modifying the list elements)
  - Corner case
  - but annoying

will change

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)


```
1  "#nav"                // Any element with id=nav
2
3  "div"               // Any <div> element
4
5  ".warning"            // Any element with "warning" class
6
7  "#log span"             // Any <span> descendant of id="log"
8
9  "#log > span"          // Any span child element of id="log"
10
11 "body>h1:first-child" // first <h1> child of <body>
12
13 "div, #log"           // All div elements, element with id="log"
```

- powerful, but hard to debug

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

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

log subtree search for E/R only.

Class Activity

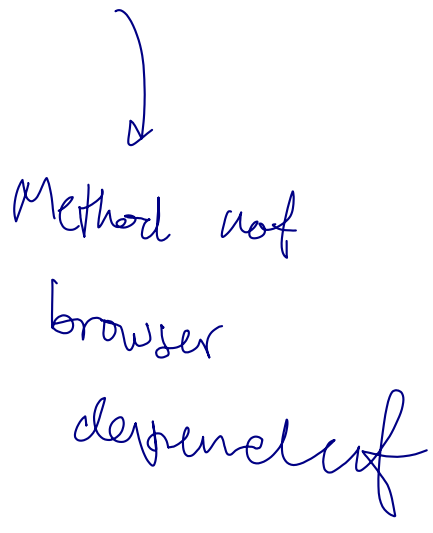
- Assume the page contains a div element with id id, which contains a series of images (img nodes).
- Write a function that takes two arguments, id and interval. At each interval, the images must be "rotated", i.e., image0 will become image1, image1 will become image2, etc.

```
1  function changeImages(id, interval) {
2
3  }
```

DOM Traversal

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



Before accessing or manipulating the DOM...

Problem

- When your JS code executes, the page might not have finished loading
  - ⇒The DOM tree might not be fully instantiated / might change!

window.onload

- *Event* that gets fired when the DOM is fully loaded (see previous lecture for more information on events)
- Like any other event – you specify a callback function
- Your DOM manipulation code should go inside that function

↳ not earlier

```
1 // DOM Level 1 way shown below -- not recommended!. How to
      do it with DOM Level 2?
2 window.onload = function() { /* Access the DOM here... */ }
```

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 → everything else.
- Text nodes: 3 -> Para, text → it is a node w/ no children
- Comment node: 8

nodeValue

Textual content of Text of comment node

nodeName

→ tag.

Tag name of a node, converted to upper-case

Example: Find a Text Node

→ sensitive to exact text.

- 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

```
1  function search(node, text) {
2      /* ... */
3  };
4
5  var result = search(window.document, "Hello world!");
```

Searching Recursively for a Text Node


```
1  function search(node, text) {
2      var found = false;
3      if (node.nodeType==3) {
4          if (node.nodeValue === text) found = true;
5      } else { // textNodes cannot have children
6          var cn = node.childNodes;
7          if (cn) {
8              for (var i=0; i < cn.length; i++) {
9                  found = found || search(cn[i], text);
10             }
11         }
12     }
13     return found;
14 };
15
16 var result = search(window.document, "Hello world!");
```

text node, no children.

recursive call on all children

Detail: if you find first time, stops recurs. and goes up tree.

↳ or in Javascript - short circuit exec. → cuts down on doing srch

Class Activity

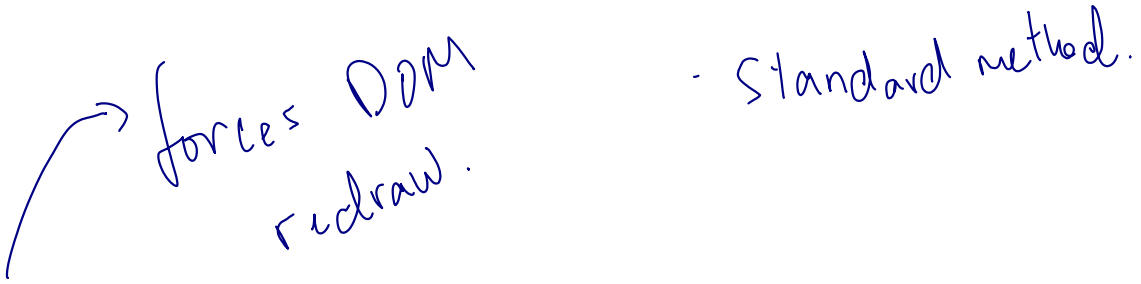
- Write a function that will traverse the DOM tree rooted at a node with a specific 'id', and checks if any of its sibling nodes and itself in the document is a text node, and if so, concatenates their text content and returns it.
- Can you generalize it so that it works for the entire subtree rooted at the sibling nodes ?

Modifying 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

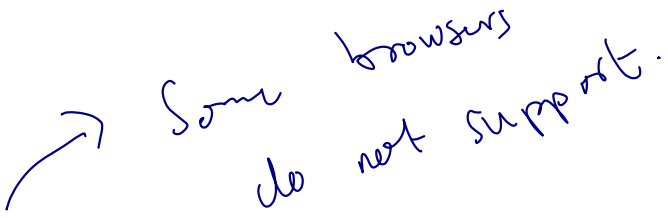
- Assume that you want to change the URL of an image object in the DOM with id="myimage" after a 5 second delay to "newImage.jpg"

```
1  var myImage = document.getElementById("myimage");
2  setTimeout( function() {
3      myImage.src ="newImage.jpg";
4  }, 5000 );
```

Source file of image

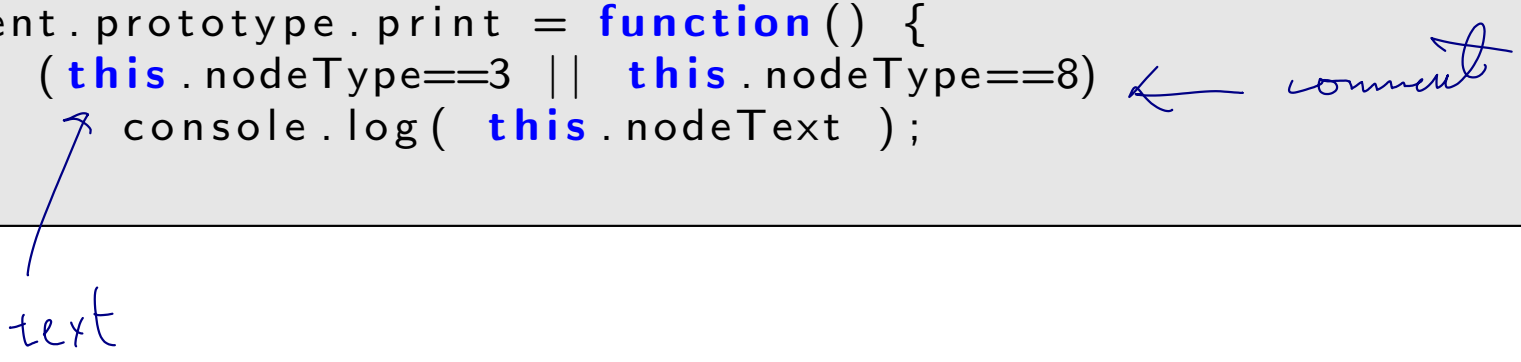
- Know property
- Sometimes non-standard

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 !

```
1  Element.prototype.print = function() {
2      if (this.nodeType==3 || this.nodeType==8)
3          console.log( this.nodeText );
4  }
```



Example: Adding new attributes to DOM elements

– for most save browsers.

- 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 !

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

- internal use only

Accessing the raw HTML of a node


- avoid, bad idea

- lots of bugs

- 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

```
1  // HTML: <p id="myP">I am a paragraph.</p>
2  // JS code:
3  var e = document.getElementById("myP");
4  console.log( e.innerHTML );
5  e.innerHTML = "Don't do this !";
```

- wipes out DOM tree & reparse to DOM
- breaks DOM abstraction.
- Applies to other parts of ref. to DOM tree.

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.

Adding and removing nodes

Creating New and Copying Existing DOM Nodes

Creating New DOM Nodes

- Using either document.createElement("element") OR document.createTextNode("text content")

```
1  var newNode = document.createTextNode("hello");
2  var elNode = document.createElement("h1");
```

Copying Existing DOM Nodes: use *cloneNode*

- Single argument can be true or false
  - True: deep copy (recursively copy all descendants)
- new node can be inserted into a different document

```
1  var existingNode = document.getElementByid("my");
2  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

Similar, except that it inserts the node before the one that is specified as the second argument (*lastChild* if it's null)

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

Removing and replacing nodes

Removing a node *n*: *removeChild*

```
1  n.parentNode.removeChild(n);
```

Replacing a node *n* with a new node: *replaceChild*

```
1  n.parentNode.replaceChild(
2      document.createTextNode("[redacted]"),
3      n);
```

Example to put it all together


```
1  // function to replace a node n by making it a child of a
       new "div" element with id = "id"
2  function newdiv(n, id) {
3      var div = document.createElement("div");
4      div.id = id;
5      n.parentNode.replaceChild(div, n);
6      div.appendChild(n);
7  };
```

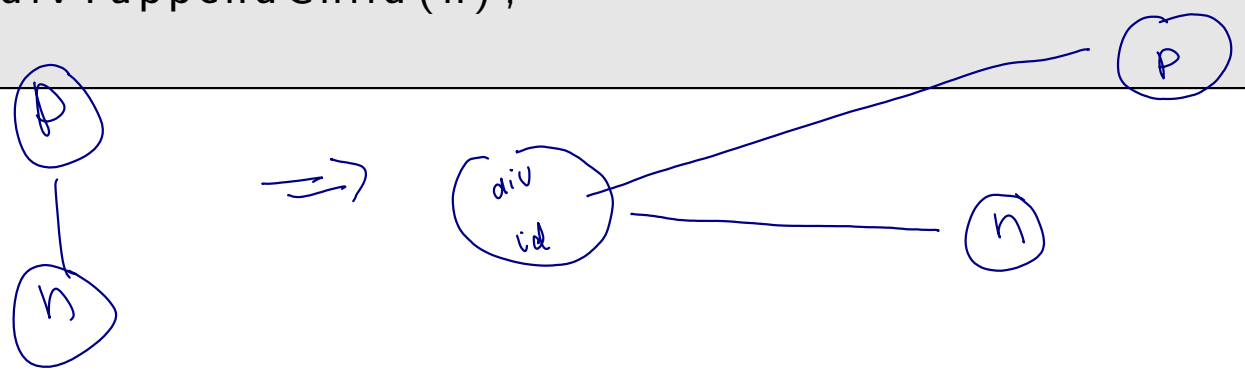


Table of Contents