DOM and Events - Part 2

Building Modern Web Applications - VSP2022

Karthik Pattabiraman
Kumseok Jung

Recap from Lecture 1

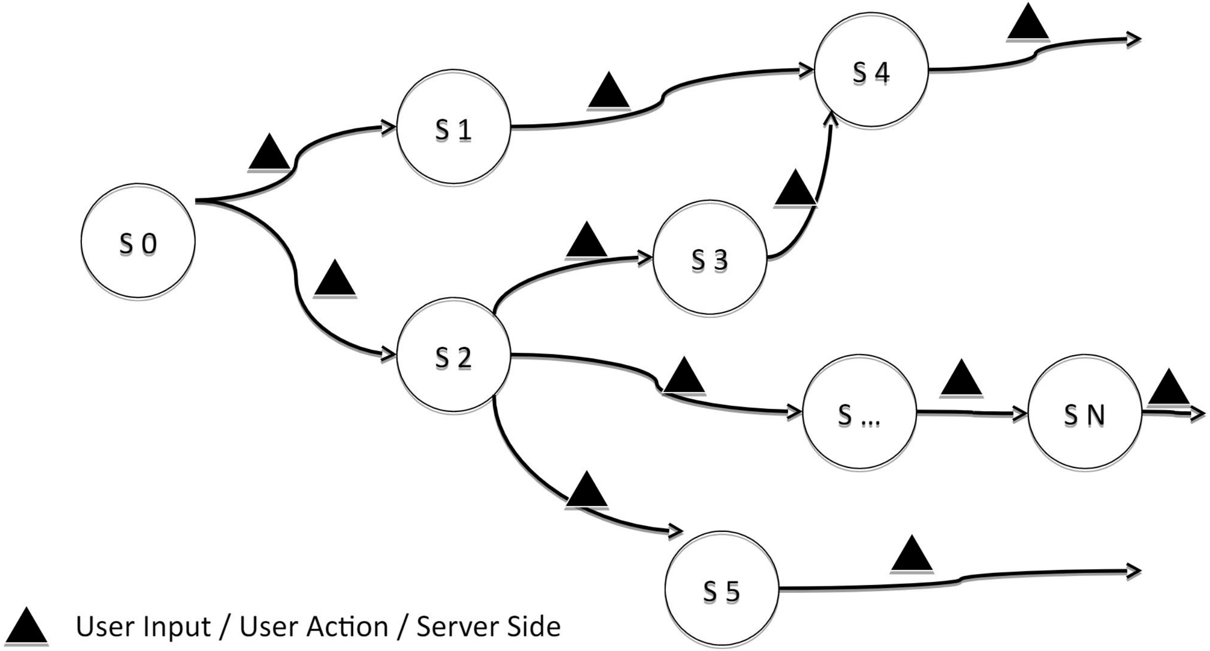
1. **Recap from Lecture 1**
2. DOM APIs
3. DOM Traversal
4. DOM Manipulation

Recap from Lecture 1: DOM

- 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: an evolving entity

DOM is highly dynamic!



Why study DOM interactions?

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

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!

DOM APIs

1. Recap from Lecture 1
2. **DOM APIs**
3. DOM Traversal
4. DOM Manipulation

Selecting HTML 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

Selecting HTML 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

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

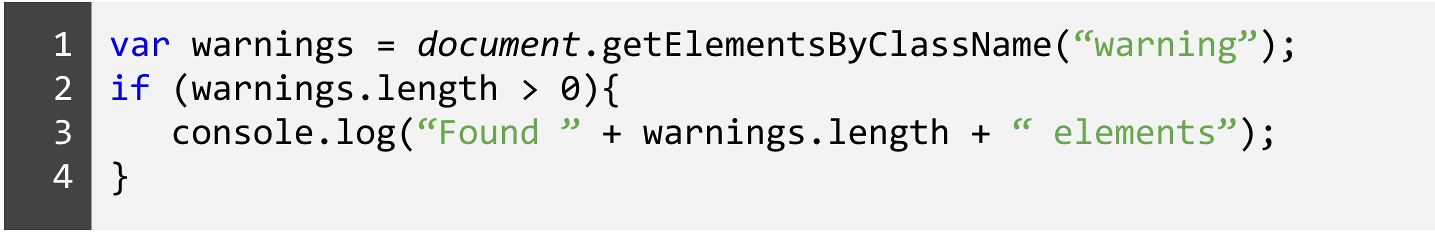
Method 2: `getElementsByTagName`

- 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)

```
var images = document.getElementsByTagName(“img”);
for (var i = 0; i < images.length; i++){
   images[i].style.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



```
var warnings = document.getElementsByClassName(“warning”);
if (warnings.length > 0){
   console.log(“Found ” + warnings.length + “ elements”);
}
```

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 through 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 examples

```
“#nav”        // Any element with id=“nav”

“div”         // Any <div> element

“.warning”    // Any element with “warning” class

“#log span”   // Any <span> 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 and 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

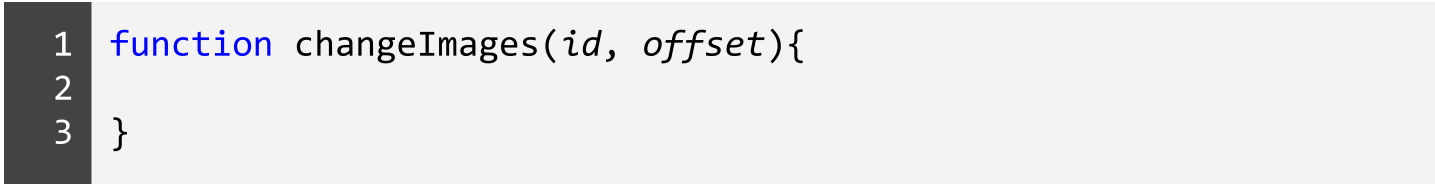
```
var log = document.getElementById(“log”);
var error = log.getElementsByClassName(“error”);
if (error.length === 0){ … }

```

Class Activity

lectures/lecture-3/changeImage.html
lectures/lecture-3/changeImage.js

- 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 `offset`. At each `offset`, the images must be “rotated”, i.e., `image0` will become `image1`, `image1` will become `image2`, etc.



```
function changeImages(id, offset){

}
```

- To repeat the execution of a given function `f` at a specific interval (e.g. 1000 ms): `setInterval(1000, f);`

DOM Traversal

1. Recap from Lecture 1
2. DOM APIs
3. **DOM Traversal**
4. DOM Manipulation

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

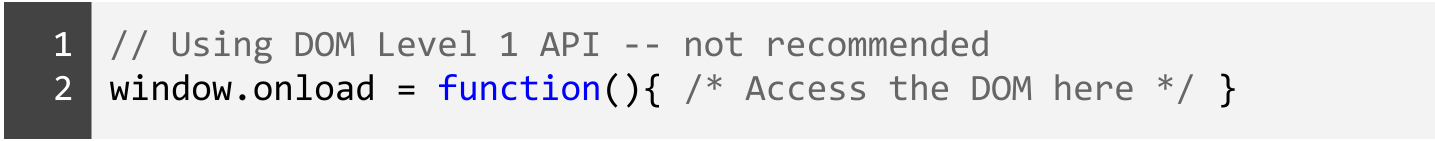
Before accessing or manipulating the DOM...

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 (we'll get back to events later...)
- You can give a callback function to execute upon proper loading of the DOM.
- Your DOM manipulation code should go inside that function



```
// Using DOM Level 1 API -- not recommended
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 last child 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’
  - Element node: 1
  - Text node: 3
  - Comment node: 8
  - Document node: 9
- `nodeValue`: Textual content of Text of comment node
- `nodeName`: Tag name of a node, converted to upper-case

Exercise: 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

```
function search(node, text){
    /* ... */
};
var result = search(window.document, “Hello world!”);
```

Exercise: Find a Text Node

Solution:

```
function search(node, text){
   if (node.nodeType === 3 && node.nodeValue === text){
      return true;
   }
   else if (node.childNodes){
      for (var i = 0; i < node.childNodes.length; i++){
         var found = search(node.childNodes[i], text);
         if (found) return found;
      }
   }
   return false;
};
var result = search(window.document, “Hello world!”);
```

Class Activity

lectures/lecture-3/domcollect.html
lectures/lecture-3/domcollect.js

- 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?

DOM Manipulation

1. Recap from Lecture 1
2. DOM APIs
3. DOM Traversal
4. **DOM Manipulation**

Adding and removing nodes

- 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

Creating New and Copying Existing DOM Nodes

- Creating New DOM Nodes
  - Using either *document*.createElement(“element”) OR *document*.createTextNode(“text content”)

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

```
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`: 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 n: `removeChild`

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

- Replacing a node n with a new node: `replaceChild`

```
var edit = document.createTextNode(“[redacted]”);
n.parentNode.replaceChild(edit, n);
```

Class Activity

lectures/lecture-3/domadd.html
lectures/lecture-3/domadd.js

- Write a function newdiv that takes two parameters: a node n and a string id. The function should replace node n by making it a child of a new <div> element with id = id.

```
/* function to replace a node n by making it a child of a new
   <div> element with id = “id” */
function newdiv(n, id){
   /* ... */
};
```

Class Activity

lectures/lecture-3/domadd.html
lectures/lecture-3/domadd.js

- Write a function `newdiv` that takes two parameters: a node `n` and a string `id`. The function should replace node `n` by making it a child of a new `<div>` element with id = `id`.

```
/* 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);
};
```