CSE3026: Web Application Development Document Object Model (DOM)

Scott Uk-Jin Lee

9.1: Global DOM Objects

The six global DOM objects

Every Javascript program can refer to the following global objects:

name	description
`document`	current HTML page and its content
`history`	list of pages the user has visited
`location`	URL of the current HTML page
`navigator`	info about the web browser you are using
`screen`	info about the screen area occupied by the browser
`window`	the browser window

The `window` object

the entire browser window; the top-level object in DOM hierarchy

- technically, all global code and variables become part of the `window` object
- properties:
  - `document`, `history`, `location`, `name`
- methods:
  - `alert`, `confirm`, `prompt` (popup boxes)
  - `setInterval`, `setTimeout`, `clearInterval`, `clearTimeout` (timers)
  - `open`, `close` (popping up new browser windows)
  - `blur`, `focus`, `moveBy`, `moveTo`, `print`, `resizeBy`, `resizeTo`, `scrollBy`, `scrollTo`

Popup windows with `window.open`

```
window.open("http://foo.com/bar.html", "My Foo Window",
            "width=900,height=600,scrollbars=1");
```

- `window.open` pops up a new browser window
- THIS method is the cause of all the terrible popups on the web!
- some popup blocker software will prevent this method from running

The `document` object

the current web page and the elements inside it

- properties:
  - `anchors`, body, `cookie`, `domain`, `forms`, `images`, `links`, `referrer`, `title`, `URL`
- methods:
  - `getElementById`
  - `getElementsByName`
  - `getElementsByTagName`
  - `close`, `open`, `write`, `writeln`
- complete list

The `location` object

the URL of the current web page

- properties:
  - `host`, `hostname`, `href`, `pathname`, `port`, `protocol`, `search`
- methods:
  - `assign`, `reload`, `replace`
- complete list

The `navigator` object

information about the web browser application

- properties:
  - `appName`, `appVersion`, `language`, `cookieEnabled`, `platform`, `userAgent`
  - complete list
- Some web programmers examine the `navigator` object to see what browser is being used, and write browser-specific scripts and hacks:

```
if (navigator.appName === "Microsoft Internet Explorer") { ...
```

  - (this is poor style; you should not need to do this)

The `screen` object

information about the client's display screen

- properties:
  - `availHeight`, `availWidth`, `colorDepth`, `height`, `pixelDepth`, `width`
  - complete list

The `history` object

the list of sites the browser has visited in this window

- properties:
  - `length`
- methods:
  - `back`, `forward`, `go`
- complete list
- sometimes the browser won't let scripts view `history` properties, for security

Unobtrusive JavaScript

- JavaScript event code seen previously was *obtrusive*, in the HTML; this is bad style
- now we'll see how to write *unobtrusive* JavaScript code
  - HTML with minimal JavaScript inside
  - uses the DOM to attach and execute all JavaScript functions
- allows separation of web site into 3 major categories:
  - **content** (HTML) - what is it?
  - **presentation** (CSS) - how does it look?
  - **behavior** (JavaScript) - how does it respond to user interaction?

Obtrusive event handlers (bad)

```
<button onclick="okayClick();">OK</button>
```

```
// called when OK button is clicked
function okayClick() {
    alert("booyah");
}
```

OK

- this is bad style (HTML is cluttered with JS code)
- goal: remove all JavaScript code from the HTML body

Attaching an event handler in JavaScript code

```
// where element is a DOM element object
element.onevent = function;
```

```
<button id="ok">OK</button>
```

```
var okButton = document.getElementById("ok");
okButton.onclick = okayClick;
```

OK

- it is legal to attach event handlers to elements' DOM objects in your JavaScript code
  - notice that you do **not** put parentheses after the function's name
- this is better style than attaching them in the HTML
- Where should we put the above code?

When does my code run?

```
<html>
    <head>
        <script src="myfile.js" type="text/javascript"></script>
    </head>
    <body> ... </body> </html>
```

```
// global code
var x = 3;
function f(n) { return n + 1; }
function g(n) { return n - 1; }
x = f(x);
```

- your file's JS code runs the moment the browser loads the `script` tag
  - any variables are declared immediately
  - any functions are declared but not called, unless your global code explicitly calls them
- at this point in time, the browser has not yet read your page's `body`
  - none of the DOM objects for tags on the page have been created yet

A failed attempt at being unobtrusive

```
<html>
    <head>
        <script src="myfile.js" type="text/javascript"></script>
    </head>
    <body>
        <div><button id="ok">OK</button></div>
```

```
// global code
document.getElementById("ok").onclick = okayClick;   // error: null
```

- problem: global JS code runs the moment the script is loaded
- script in `head` is processed before page's `body` has loaded
  - no elements are available yet or can be accessed yet via the DOM
- we need a way to attach the handler after the page has loaded...

The `window.onload` event

```
// this will run once the page has finished loading
function functionName() {
    element.event = functionName;
    element.event = functionName;
    ...
}

window.onload = functionName;   // global code
```

- we want to attach our event handlers right after the page is done loading
  - there is a global event called `window.onload` event that occurs at that moment
- in `window.onload` handler we attach all the other handlers to run when events occur

An unobtrusive event handler

```
<button id="ok">OK</button>   <!-- look Ma, no JavaScript! -->
```

```
// called when page loads; sets up event handlers
function pageLoad() {
    document.getElementById("ok").onclick = okayClick;
}

function okayClick() {
    alert("booyah");
}

window.onload = pageLoad;  // global code
```

OK

Common unobtrusive JS errors

- many students mistakenly write `()` when attaching the handler

```
window.onload = pageLoad~~()~~;
window.onload = pageLoad;

okButton.onclick = okayClick~~()~~;
okButton.onclick = okayClick;
```

  - our **JSLint** checker will catch this mistake
- event names are all lowercase, not capitalized like most variables

```
window.~~onLoad~~ = pageLoad;
window.onload = pageLoad;
```

Anonymous functions

```
function(parameters) {
    statements;
}
```

- JavaScript allows you to declare **anonymous functions**
- quickly creates a function without giving it a name
- can be stored as a variable, attached as an event handler, etc.

Anonymous function example

```
window.onload = function() {
    var okButton = document.getElementById("ok");
    okButton.onclick = okayClick;
};

function okayClick() {
    alert("booyah");
}
```

OK

- or the following is also legal (though harder to read and bad style):

```
window.onload = function() {
    var okButton = document.getElementById("ok");
    okButton.onclick = function() {
        alert("booyah");
    };
};
```

Unobtrusive styling

element.classList.add(“highlighted”);
element.classList.remove(“class”);
element.classList.contains(“class”);
element.classList.toggle(“class”);

ie6미만에서는 동작하지 않는다.

```
function okayClick() {
    ~~this.style.color = "red";~~
    this.className = "highlighted";
}
```

```
.highlighted { color: red; }
```

- well-written JavaScript code should contain as little CSS as possible
- use JS to set CSS classes/IDs on elements
- define the styles of those classes/IDs in your CSS file

9.2: The Dom Tree

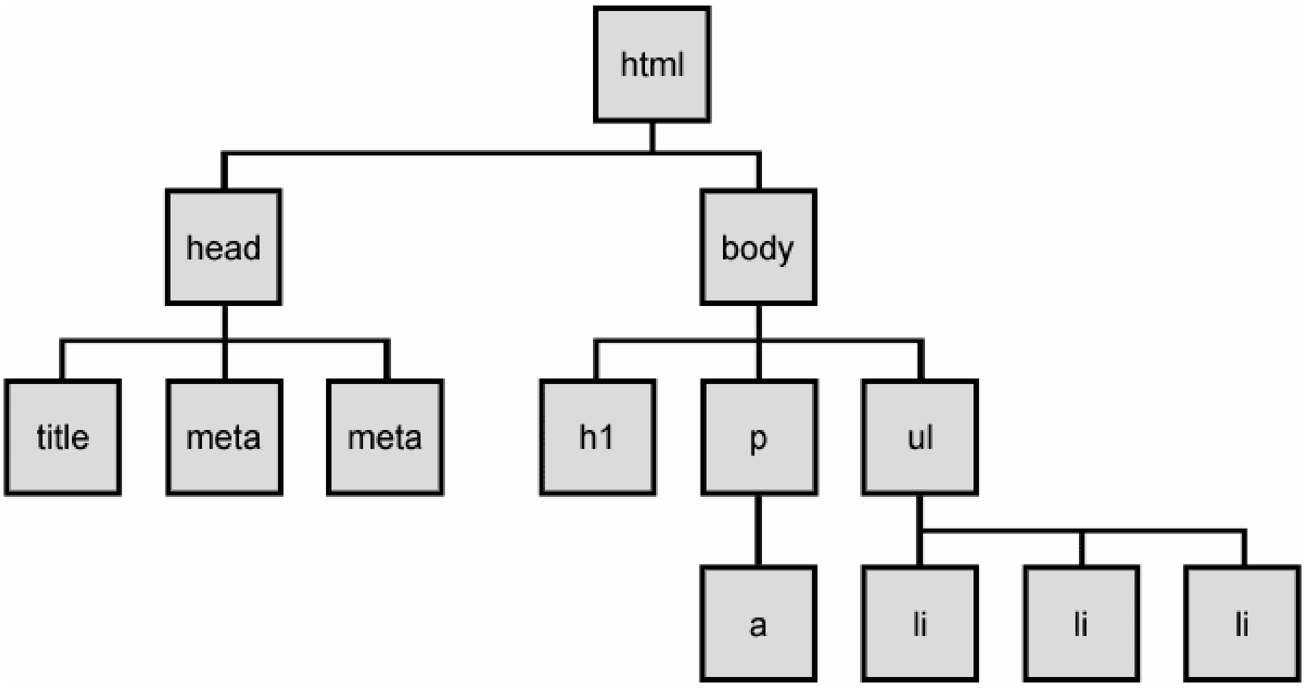
- 9.1: Global DOM Objects
- **9.2: The DOM Tree**

Complex DOM manipulation problems

How would we do each of the following in JavaScript code? Each involves modifying each one of a group of elements …

- When the Go button is clicked, reposition all the `div`s of class `puzzle` to random `x`/`y` locations.
- When the user hovers over the maze boundary, turn all maze walls red.
- Change every other item in the `ul` list with `id` of `TAs` to have a gray background.

The DOM tree

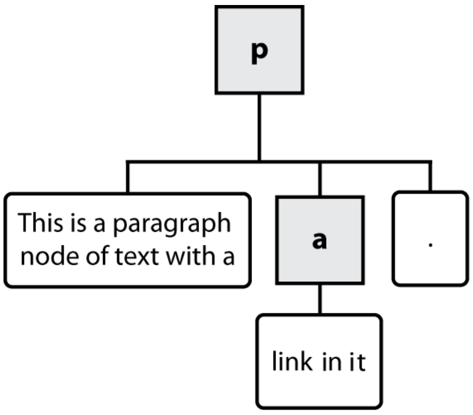


- The elements of a page are nested into a tree-like structure of objects
  - the DOM has properties and methods for traversing this tree

Types of DOM nodes

```
<p>
    This is a paragraph of text with a
    <a href="/path/page.html">link in it</a>.
</p>
```

- ■ **element nodes** (HTML tag)
  - can have children and/or attributes
- □ **text nodes** (text in a block element)
- ● **attribute nodes** (attribute/value pair)
  - text/attributes are children in an element node
  - cannot have children or attributes
  - not usually shown when drawing the DOM tree



Traversing the DOM tree

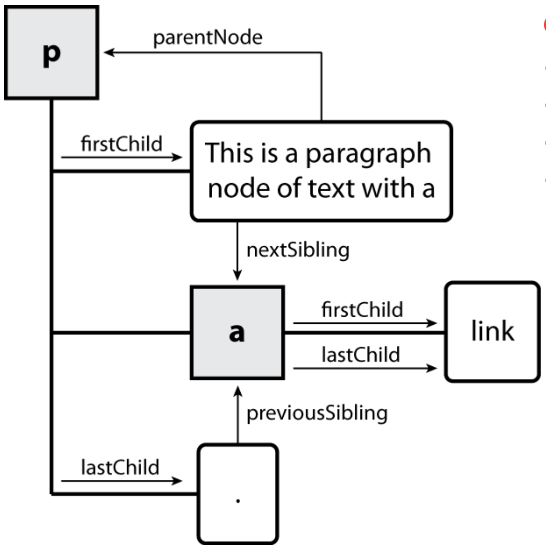
every node's DOM object has the following properties:

name(s)	description
`firstChild`, `lastChild`	start/end of this node's list of children
`childNodes`	array of all this node's children
`nextSibling`, `previousSibling`	neighboring nodes with the same parent
`parentNode`	the element that contains this node

- complete list of DOM node properties
- browser incompatiblity information (IE6 sucks)

DOM tree traversal example

```
<p id="foo">This is a paragraph of text with a
    <a href="/path/to/another/page.html">link</a>.</p>
```



document.getElementById("id")
.firstChild
.nextSibling
.firstChild
.textContent

Element vs. text nodes

```
<div>
    <p>
        This is a paragraph of text with a
        <a href="page.html">link</a>.
    </p>
</div>
```

- Q: How many children does the `div` above have?
- A: 3
    - an element node representing the <p>
    - two *text nodes* representing `"\n\t"` (before/after the paragraph)
- Q: How many children does the paragraph have? The `a` tag?

Selecting groups of DOM objects

- methods in `document` and other DOM objects (* = HTML5):

name	description
`getElementsByTagName`	returns array of descendents with the given tag, such as `"div"`
`getElementsByName`	returns array of descendents with the given `name` attribute (mostly useful for accessing form controls)
`querySelector` *	returns the first element that would be matched by the given CSS selector string
`querySelectorAll` *	returns an array of all elements that would be matched by the given CSS selector string

Getting all elements of a certain type

highlight all paragraphs in the document:

```
var allParas = document.querySelectorAll("p");
for (var i = 0; i < allParas.length; i++) {
    allParas[i].style.backgroundColor = "yellow";
}
```

```
<body>
    <p>This is the first paragraph</p>
    <p>This is the second paragraph</p>
    <p>You get the idea...</p>
</body>
```

Complex selectors

highlight all paragraphs inside of the section with ID `"address"`:

```
// var addrParas = document.getElementById("address").getElementsByTagName("p");
var addrParas = document.querySelectorAll("#address p");
for (var i = 0; i < addrParas.length; i++) {
    addrParas[i].style.backgroundColor = "yellow";
}
```

```
<p>This won't be returned!</p>
<div id="address">
    <p>1234 Street</p>
    <p>Atlanta, GA</p>
</div>
```

Creating new nodes

name	**description**
`document.createElement("`*tag*`")`	creates and returns a new empty DOM node representing an element of that type
`document.createTextNode("`*text*`")`	creates and returns a text node containing given text

```
// create a new <h2> node
var newHeading = document.createElement("h2");
newHeading.innerHTML = "This is a heading";
newHeading.style.color = "green";
```

- merely creating a node does not add it to the page
- you must add the new node as a child of an existing element on the page...

Modifying the DOM tree

Every DOM element object has these methods:

name	description
`appendChild(`*node*`)`	places given node at end of this node's child list
`insertBefore(`*new*`, `*old*`)`	places the given new node in this node's child list just before `old` child
`removeChild(`*node*`)`	removes given node from this node's child list
`replaceChild(`*new*`, `*old*`)`	replaces given child with new node

```
var p = document.getElementById(document.createElement("p"));
p.innerHTML = "A paragraph!";
document.getElementById("main").appendChild(p);
```

Removing a node from the page

```
function slideClick() {
    var bullets = document.getElementsByTagName("li");
    for (var i = 0; i < bullets.length; i++) {
        if (bullets[i].innerHTML.indexOf("children") >= 0) {
            bullets[i].parentNode.removeChild(bullets[i]);
        }
    }
}
```

- each DOM object has a `removeChild` method to remove its children from the page

DOM versus `innerHTML` hacking

Why not just code the previous example this way?

```
function slideClick() {
    document.getElementById("thisslide").innerHTML += "<p>A paragraph!</p>";
}
```

- Imagine that the new node is more complex:
  - ugly: bad style on many levels (e.g. JS code embedded within HTML)
  - error-prone: must carefully distinguish " and '
  - can only add at beginning or end, not in middle of child list

```
function slideClick() {
    this.innerHTML += "<p style='color: red; " +
            "margin-left: 50px;' " +
            "onclick='myOnClick();'>" +
            "A paragraph!</p>";
}
```