Web Programming Step by Step

Chapter 8
The Document Object Model (DOM)

8.1: 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`

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`, `browserLanguage`, `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 (8.1.1)

- 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)

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

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

output

OK

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

Attaching an event handler in JavaScript code

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

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

output

OK

- it is legal to attach event handlers to elements' DOM objects in your JavaScript code
- this is better style than attaching them in the XHTML
- Where should we put the above code?

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>
```

```JS
// global code
var okButton = document.getElementById("ok");
okButton.onclick = okayClick;    // error: okButton is undefined
```

- 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 just as the page finishes loading

The `window.onload` event (8.1.1)

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

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

- 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

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

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

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

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

output

OK

Common unobtrusive JS errors

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

```JS
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

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

Anonymous functions (8.1.2)

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

- 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");
}
```
JS

output

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");
  };
};
```
JS

The keyword `this` (8.1.3)

```
window.onload = pageLoad;

function pageLoad() {
  var okButton = document.getElementById("ok");
  okButton.onclick = okayClick;  // bound to okButton here
}

function okayClick() {           // okayClick knows what DOM object
  this.innerHTML = "booyah";     // it was called on
}
```
JS

- event handlers attached unobtrusively are **bound** to the element
- inside the handler, the element can refer to itself as `this`
  - also useful when the same handler is shared on multiple elements

Fixing redundant code with `this`

```
<fieldset>
  <label><input id="Huey"  type="radio" name="ducks" /> Huey</label>
  <label><input id="Dewey" type="radio" name="ducks" /> Dewey</label>
  <label><input id="Louie" type="radio" name="ducks" /> Louie</label>
</fieldset>
```
HTML

```
function processDucks() {
  if (document.getElementById("huey").checked) {
    alert("Huey is checked!");
  } else if (document.getElementById("dewey").checked) {
    alert("Dewey is checked!");
  } else {
    alert("Louie is checked!");
  }
  alert(this.id + " is checked!");
}
```
JS

8.2: DOM Element Objects

Modifying text inside an element

```
var paragraph = document.getElementById("welcome");
paragraph.innerHTML = "Welcome to our site!";  // change text on page
```
JS

DOM element objects have the following properties:

- `innerHTML` : text and/or HTML tags inside a node
- `textContent` : text (no HTML tags) inside a node
  - simpler than `innerHTML`, but not supported in IE6
- `value` : the value inside a form control

Abuse of `innerHTML`

```
// bad style!
var paragraph = document.getElementById("welcome");
paragraph.innerHTML = "<p>text and <a href="page.html">link</a>";
```
JS

- `innerHTML` can inject arbitrary HTML content into the page
- however, this is prone to bugs and errors and is considered poor style
- we forbid using `innerHTML` to inject HTML tags; inject plain text only
    - so how do we add content with HTML tags in it to the page?

Adjusting styles with the DOM (8.2.2)

```
<button id="clickme">Color Me</button>
```
HTML

```
window.onload = function() {
  document.getElementById("clickme").onclick = changeColor;
};
function changeColor() {
  var clickMe = document.getElementById("clickme");
  clickMe.style.color = "red";
}
```
JS

Color Me

output

- `style` property lets you set any CSS style for an element
- contains same properties as in CSS, but with `camelCasedNames`
    - examples: `backgroundColor`, `borderLeftWidth`, `fontFamily`

Common DOM styling errors

- many students forget to write `.style` when setting styles

```
var clickMe = document.getElementById("clickme");
clickMe.color = "red";
clickMe.style.color = "red";
```
JS

- style properties are capitalized `likeThis`, not `like-this`

```
clickMe.style.font-size = "14pt";
clickMe.style.fontSize = "14pt";
```
JS

- style properties must be set as strings, often with units at the end

```
clickMe.style.width = 200;
clickMe.style.width = "200px";
clickMe.style.padding = "0.5em";
```
JS

  - write the value you would have written in the CSS, but in quotes

Unobtrusive styling (8.2.3)

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

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

- 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

8.3: The DOM Tree

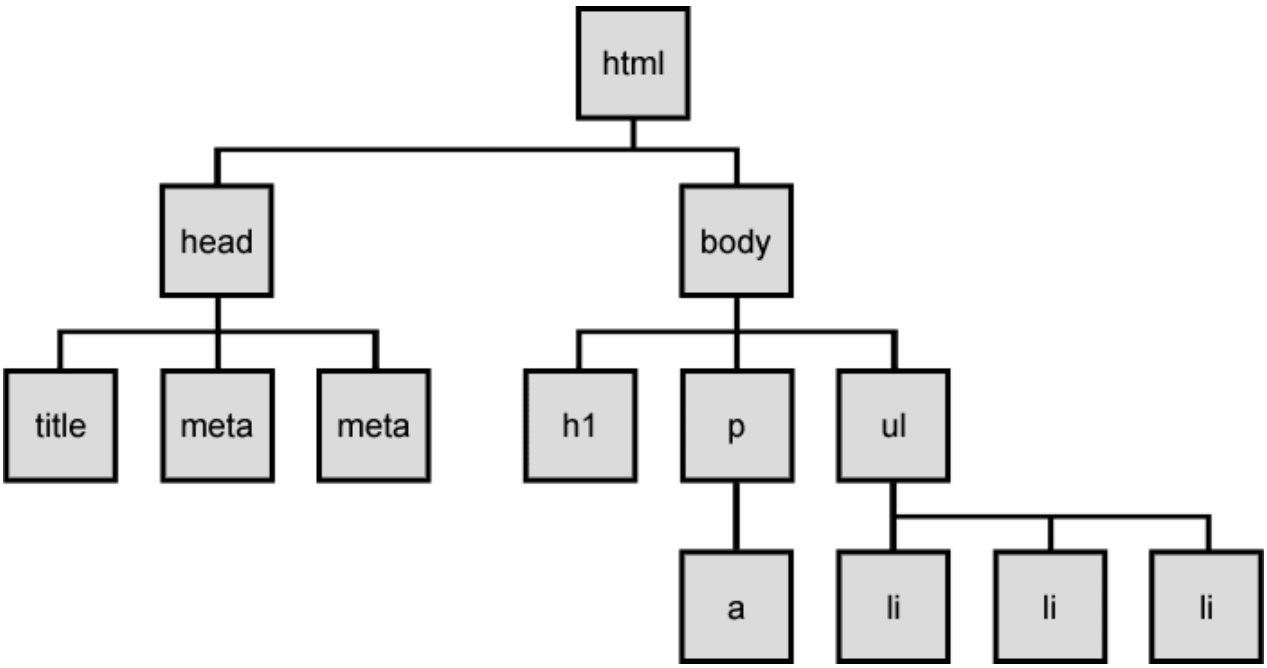
Complex DOM manipulation problems

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 `divs` 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 tree of DOM objects

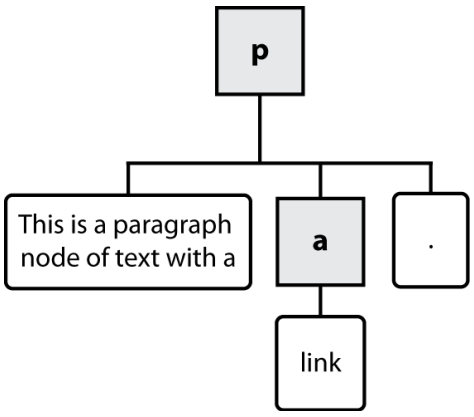
- The elements of a page are nested into a tree-like structure of objects
  - the DOM has properties and methods for traversing this tree



Types of DOM nodes (8.3.1)

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

- **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
  - they cannot have children or attributes



Traversing the DOM tree (8.3.2 - 8.3.3)

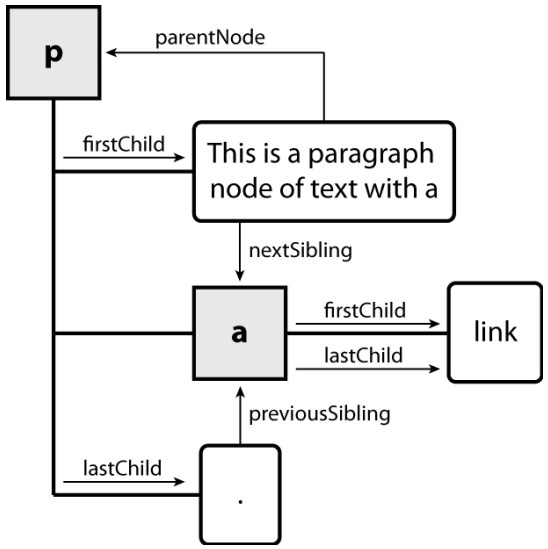
every node's DOM object has the following properties:

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

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



Element vs. text nodes

```HTML
<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 (8.3.5)

- methods in `document` and other DOM objects for accessing descendents:

name	description
`getElementsByTagName`	returns array of descendents that have the given HTML tag, such as `"div"`
`getElementsByName`	returns array of descendents that have the given `name` attribute (mostly useful for accessing form controls)

Getting all elements of a certain type

highlight all paragraphs in the document:

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

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

Combining with `getElementById`

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

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

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

Creating new nodes (8.3.5)

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

- `document.createElement("tag")` : creates and returns a new empty DOM node representing an element of that type
  - this node's properties can be set just like any other DOM node's
- `document.createTextNode("text")` : creates and returns a new text node containing the given text

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

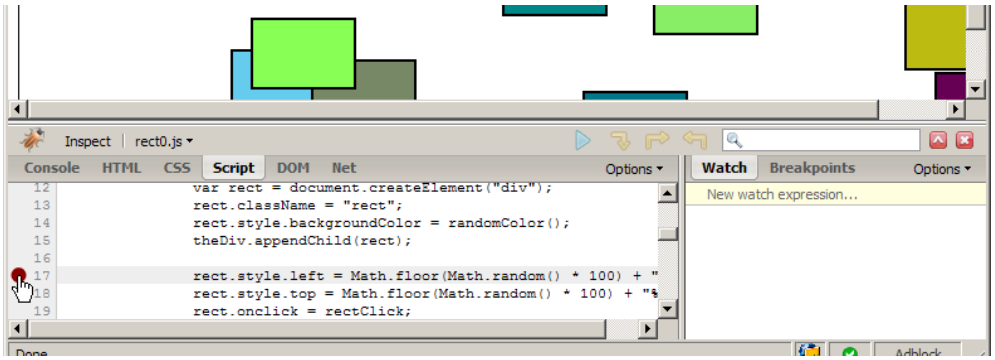
Adding a node to the page

```JS
window.onload = function() {
  var thisSlide = document.getElementById("slide38");
  thisSlide.onclick = slideClick;
}

function slideClick() {
  var p = document.createElement("p");
  p.innerHTML = "A paragraph!";
  this.appendChild(p);
}
```

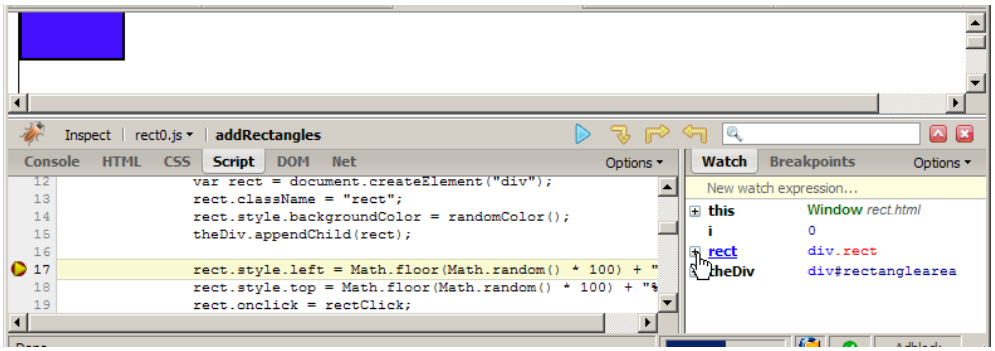
- merely creating a node does not add it to the page
- you must add the new node as a child of an existing element on the page

Firebug's debugger



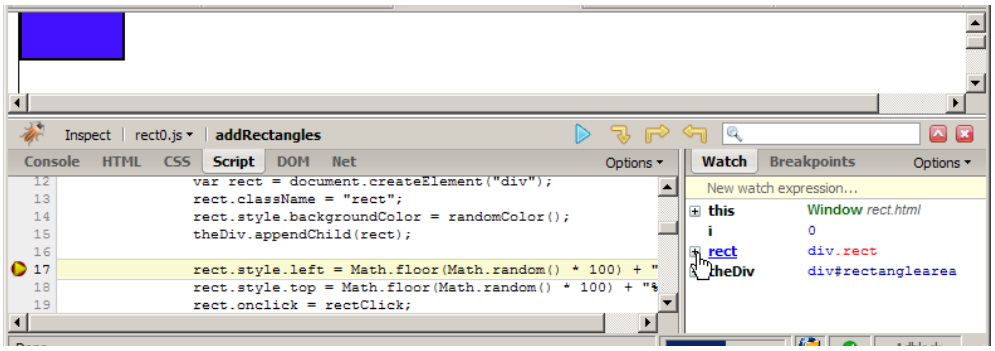
- open Firebug, click **Script** tab
- click to the left of a line to set a **breakpoint**
- **refresh** page; when script gets to that line, program will halt

Breakpoints



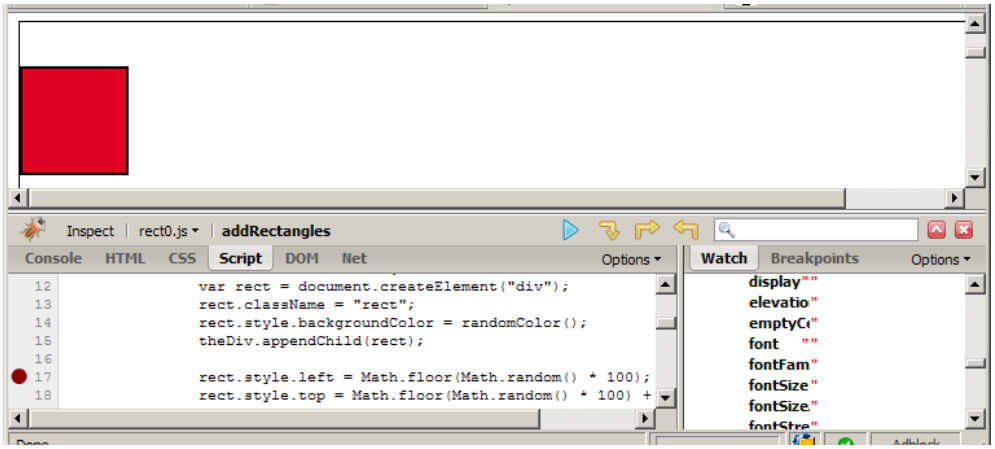
- once stopped at a breakpoint, you can examine variables in the **Watch** tab at right
  - can click + to see properties/methods inside any object
  - **this** variable holds data about current object, or global data
  - if the object is global or not listed, type its name in the "New watch expression..." box

Stepping through code



- once stopped at a breakpoint, you can continue execution:
  - **continue** (F8): start program running again
  - **step over** (F10): run current line of code completely, then stop again
  - **step into** (F11): run current line of code; if it contains a call to another function, go into it
  - **step out** (Shift-F11): run the current function to completion and return, then stop

Debugging CSS property code



- expand DOM object with +, and expand its `style` property to see all styles
- also look at HTML (left) tab, Style (right) tab to see styles

General good coding practices

- ALWAYS code with Firebug installed
- incremental development: code a little, test a little
- follow good general coding principles
  - remove redundant code
  - make each line short and simple
- use lines and variables liberally
  - it's good to save parts of a complex computation as variables
  - helps see what part of a big expression was bad/undefined/etc.
  - blank lines and profuse whitespace make code easier to read
- don't fear the Firebug debugger