Computer Science 75
Spring 2009
Scribe Notes

Lecture 9: April 13, 2009
Andrew Sellergren

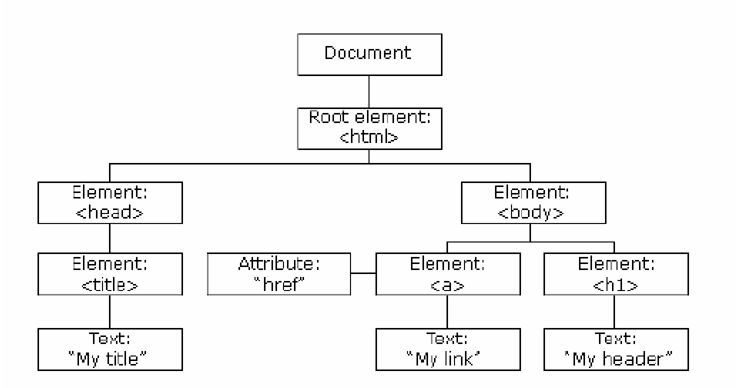
Contents

Computer Science 75 Lecture 9: April 13, 2009
Spring 2009 Andrew Sellergren
Scribe Notes

1 The DOM and Ajax (0:00–48:00)

- We first took a look at the DOM in the context of XML and Project 1. We used the query language XPath to look up information in the DOM.
- Here we have a small snippet of well-formed XHTML followed by a diagram of it using the DOM:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>My title</title>
</head>
<body>
    <a href="">My link</a>
    <h1>My header</h1>
</body>
</html>
```



JavaScript and AJAX are all about the ability to dynamically generate boxes like this on the fly. Excited yet?

- The power of Ajax, as we've already mentioned, is the ability to make HTTP requests in the background, unbeknownst to the user and without the necessity of a page refresh. The tool which makes this possible is the XMLHttpRequest object, which you can read about here:
  - Microsoft Developer Network
  - Mozilla Developer Center
  - World Wide Web Consortium

The annoying thing is that when Ajax was first rolled out, there was no agreement among the major browsers in terms of this object. So when we begin with our examples, we'll be taking a look at a few hacks that achieve cross-browser compatibility. Know that this is one of the advantages of JavaScript libraries like jQuery, which abstract away the need for hacks like this.

- The XMLHttpRequest object is a black box that takes care of both sending and receiving the data behind the scenes. It has the following methods, most of which are fairly self-explanatory:
  - `abort()`
  - `getAllResponseHeaders()`
  - `getResponseHeader(header)`
  - `open(method, url)`
  - `open(method, url, async)`
  - `open(method, url, async, user)`
  - `open(method, url, async, user, password)`
  - `send()`
  - `send(data)`
  - `setRequestHeader(header, value)`
- Because of the *Same Origin Policy*, Ajax requests can only be made for files on the same server. This is to prevent malicious JavaScript from being loaded without the user being aware. We'll come back to this in a few weeks when we talk about security, but for now, realize that it's the reason that you must write `cities.php`, which acts as a proxy for gathering the RSS information from Google News.
- In addition to its methods, the XMLHttpRequest object has the following properties:
  - `onreadystatechange`
  - `readyState`
    - `0 (unitialized)`
    - `1 (open)`
    - `2 (sent)`
    - `3 (receiving)`
    - `4 (loaded)`
  - `responseBody`
  - `responseText`
  - `responseXML`

  - `status`
    * `200 (OK)`
    * `404 (Not Found)`
    * `500 (Internal Server Error)`
  - `statusText`

  As you might've guessed, `onreadystatechange` is effectively an event handler which allows you to react to any change in the state of your request. Status code 4 is really the only important one–it means that a response has come back and been loaded. The properties with prefix `response` are exactly what you would expect. This is how you access the data that you asked for. The `status` property, of course, corresponds to the HTTP status code that is returned.

- Since the file you're requesting via Ajax is ultimately on your own server, you have a choice to make: what's the format of the data that it will return? The following are a few examples, all of which are specified by the `Content-type` header:
  - XHTML (text/html)
  - XML (text/xml)
  - JSON (application/json)

  XHTML is a quick and dirty way of plopping some content out on your website. If you want a little more structure, try XML or JSON, which have a bit of metadata bundled along with them. JSON allows you to serialize data from an array or an object and then pass it over the wire where it can be unserialized and read back into memory.

- Take a look at `ajax1.html`. Notice that aesthetically, it's a very simple form, but it does manage to accomplish something we haven't yet accomplished: delivering content without a page refresh. If you input a stock symbol and click Get Quote, an alert pops up to display the stock price. If we look at this using Live HTTP Headers, we see that an HTTP request for the file `quote1.php` was made with the stock symbol passed as a `GET` parameter. Take a look at how we achieve this in the body of `ajax1.html`:

```
<body>
  <form onsubmit="quote(); return false;">
      Symbol: <input id="symbol" type="text" />
    <br /><br />
    <input type="submit" value="Get Quote" />
  </form>
</body>
```

Here, we're deliberately telling the form not to submit itself and reload (`return false;`) but rather to make a call to our own function `quote()`. What's the substance of `quote()`? We need only to look to the `<head>` tag to find out:

```
<head>
    <script type="text/javascript">
    // <![CDATA[

        // an XMLHttpRequest
        var xhr = null;

        /*
         * void
         * quote()
         *
         * Gets a quote.
         */
        function quote()
        {
            // instantiate XMLHttpRequest object
            try
            {
                xhr = new XMLHttpRequest();
            }
            catch (e)
            {
                xhr = new ActiveXObject("Microsoft.XMLHTTP");
            }

            // handle old browsers
            if (xhr == null)
            {
                alert("Ajax not supported by your browser!");
                return;
            }

            // construct URL
            var url = "quote1.php?symbol=" +
                document.getElementById("symbol").value;

            // get quote
            xhr.onreadystatechange = handler;
            xhr.open("GET", url, true);
            xhr.send(null);
        }
        /*
         * void
         * handler()
         *
         * Handles the Ajax response.
         */
        function handler()
        {
            // only handle loaded requests
            if (xhr.readyState == 4)
            {
                if (xhr.status == 200)
                    alert(xhr.responseText);
                else
                    alert("Error with Ajax call!");
            }
        }

    // ]]>
    </script>
    <title></title>
</head>
```

Notice we declare a global variable `xhr` before trying to define it as a new instance of an XMLHttpRequest object. This variable is *global* by virtue of being declared outside the context of any function. If you want to declare a global variable within a function, then leave off the *var* at the beginning of the declaration. We're also using the `try` and `catch` syntax, which is a way of achieving error-handling. The `try` and `catch` syntax allows you to sandwich together several lines of code and to deal with any number of different *exceptions* or errors that result.

- Note that JavaScript doesn't really have classes or constructors, per se, but we can mimic that behavior quite well.
- What's the deal with two different assignments to `xhr`? Microsoft, liking to be different, has its own version of the XMLHttpRequest object called ActiveXObject. We're kind of abusing the `try` and `catch` syntax here, but effectively we're saying "If instantiating an XMLHttpRequest object fails, assume the user has a Microsoft browser and try to instantiate an ActiveXObject instead." There are other ways to do this, but this one is tried and true.
- Finally, if both those instantiation attempts fail, we're assuming that the

user's browser doesn't support Ajax and we're providing him with a message accordingly.

- JavaScript, like PHP, doesn't have strict data typing, so we can simply declare a `var` to hold our URL. To construct our URL, we use the + sign to concatenate whatever the user inputted into the `symbol` form (which we retrieve by calling `getElementById()`) onto the string `quote1.php?symbol=`. There are different ways to get at the DOM nodes that we want, but one good method is to give them unique IDs and then find them using the `getElementById()` method.

- One advantage of JavaScript is *asynchronicity.* With JavaScript, we can call a function and have it return immediately, calling another function as soon as it returns. This is great for UI considerations in which we want to appear like nothing has stalled. If you do, however, want to turn off the asynchronicity of XMLHttpRequest, you have that ability by simply specifying `false` as the third argument to the `open()` method. Take a look at these lines:

```
xhr.onreadystatechange = handler;
xhr.open("GET", url, true);
xhr.send(null);
```

  What we're doing here is telling the operating environment to call the `handler()` function whenever it's ready to change its state. Then we're telling it to send the `GET` request in an asychronous fashion (using `true` as the third argument to the `open()` method). What does `handler()` do? Notice it makes a call to the `alert()` method (which accomplishes the pop-up) only if the `readyState` property is equal to 4 (which means that the page is loaded) and the overall status is 200 (meaning OK). The argument to `alert()` is the `responseText` property of the XMLHttpRequest object, which, in this case, is the stock price. w00t.

- Question: will the browser handle redirects for you (i.e. `status` of 301 or 302)? David thinks yes, but isn't quite sure.

- Question: why do we assign `handler` without parentheses? We're actually passing a pointer to the function, telling it to be called only for our specific event. If we had the parentheses on there, the function would be called as soon as that line of code executed. Not what we want.

- Question: what if you have multiple functions with the same name but different signatures? Which one gets called after an assignment like this? Not sure, actually!

- What about `quote1.php`? Take a look at its code:

```
<?php

    /**
     * quote1.php
     *
     * Outputs price of given symbol as text/html.
     *
     * Computer Science 50
     * David J. Malan
     */

    // get quote
    $handle = @fopen("http://download.finance.yahoo.com/d/"
              . "quotes.csv?s={$_GET['symbol']}&f=e1l1", "r");
    if ($handle !== FALSE)
    {
        $data = fgetcsv($handle);
        if ($data !== FALSE && $data[0] == "N/A")
            print($data[1]);
        fclose($handle);
    }
?>
```

Here we're calling `fopen()`, which, as you're probably familiar with by now, allows you to open URLs like files and start reading from them. If this call returns true, then we make a call to `fgetcsv()`, which should give us only the first row. Then we're doing a sanity check and making sure that the symbol is valid before we simply print out its price.

- If we access quote1.php?symbol=msft directly from our browser, we get the stock price printed directly to our browser window. However, because we're normally making an HTTP request for this page from within `ajax1.html`, this value won't be printed to our browser window but rather incorporated into an JavaScript alert pop-up thanks to our AJAX call.
- Truthfully, this webpage isn't even a webpage, since there's no markup associated with it. To be more explicit, then, we might've specified an appropriate header before spitting out the stock price:

  ```
  header("Content-type: text/plain");
  ```

- Let's take a moment to answer the question from before about whether Ajax will handle redirects or not. We'll start by rewriting the code of `quote1.php` to be the following:

  ```
  header("Location:
   http://www.cs75.net/lectures/9/src/quote1.5.php?symbol={$_GET["symbol"]}");
  ```

Then, as you might've guessed, we copy `quote1.php` to `quote1.5.php`. Tada! It works. Note that we had to manually add the `GET` arguments back onto the end of the URL. Turns out it works in Firefox, IE, as well as Chrome. Don't try to redirect to a different server and then embed the content in your own webpage, because it most likely won't work.

- Now, a shameless, pointless break in the action so that this one section doesn't overrun all the scribe notes!

2 More Fun with AJAX (48:00–107:00)

- No one likes pop-ups, so how can we achieve this content update in a more elegant fashion? Take a look at `ajax2.html` where it differs from `ajax1.html`:

  ```
  document.getElementById("price").value = xhr.responseText;
  ```

  Notice this change to the line in `handler()`. Instead of printing the `responseText` property to a JavaScript pop-up, we're going to write it to a form input with `id="price"`.

- What's new with `ajax3.html`? Nothing much, except that now instead of writing the stock price to a `form`, we're going to write it to a `span`. To do this, we change the line of code like so:

  ```
  document.getElementById("price").innerHTML = xhr.responseText;
  ```

  Note that writing to `innerHTML` isn't technically standards-compliant, but it's a fairly widespread (and useful) technique.

- If we open up Firebug, we can watch the previous content of this `span` be entirely clobbered in realtime.

- A quick sidenote about Project 3: as noted on the Bulletin Board, the so-called "proper" way of handling duplicate zipcodes is to only import the one with "P" for the primary record column. However, as Chris Power suggested, you could also define a joint primary key using city name and zipcode such that only a single record will be inserted for each unique pair of those. Frankly, either way is fine, and it's also acceptable for you to do some interpreting of the data on your own and decide what's best.

- Let's take a look, now, at `quote2.php`, which will return a little more data than before:[1].

[1]Sorry for the broken URL, but I needed it to wrap across two lines to fully display

```
<?
    /**
     * quote2.php
     *
     * Outputs price, low, and high of given symbol as plain/text.
     *
     * David J. Malan
     * Computer Science E-75
     * Harvard Extension School
     */

    // send MIME type
    header("Content-type: text/plain");

    // try to get quote
    $handle = @fopen("http://download.finance.yahoo.com/d/"
    . "quotes.csv?s={$_GET['symbol']}&f=e1l1hg", "r");
    if ($handle !== FALSE)
    {
        $data = fgetcsv($handle);
        if ($data !== FALSE && $data[0] == "N/A")
        {
            print("Price: {$data[1]}\n");
            print("High: {$data[2]}\n");
            print("Low: {$data[3]}");
        }
        fclose($handle);
    }
?>
```

Here, we're doing the same as before, except we're printing multiple lines of stock data instead of just one. If you take a look at `ajax5.html`, you'll see that it prints this extra data, as before, in the `innerHTML` of a `span`.

- Note that it's a little lazy and sloppy of us to only be checking for a `readyState` of 4. We could certainly check for other status changes along the way.
- A quick note: why is it okay that we don't have any of the usual tags, e.g. `head`, `body`, or even `html`, in our PHP quote files? Well, we're simply inserting it into another document which presumably is valid XHTML.
- In `ajax6.html`, we have a poor man's progress bar:

  ```
  document.getElementById("quote").innerHTML = "Looking up symbol...";
  ```

  Before we make our Ajax call, we're just manually changing the content so that there's the appearance of progress. If we wanted to create animation, we could put a GIF in there using an `img` tag—perhaps one that's already embedded whose `display` property we could dynamically change.

- In `ajax7.html`, we actually implement this animation using only a few extra lines of code, a few of them being the below:

```
// show progress
document.getElementById("progress").style.display = "block";
```

  Here, we're dynamically accessing the `style` property of a `div` that's already in our DOM by virtue of the following XHTML:

```
<div id="progress" style="display: none;">
  <img alt="Please Wait" src="19-0.gif" />
  <br /><br />
</div>
```

  As you can see, it's just a GIF whose `display` CSS property is set to `none`. If we want to toggle it to be visible, we just change this to the standard `block`. Then, in our handler, once our request has returned successfully, we reset the GIF's `display` to `none`.

- The next step we're going to take is a little bigger. We'll start by examining the relevant changes that have been made in `quote5.php`:

```
// set MIME type
header("Content-type: text/xml");

// output root element's start tag
print("<quote symbol='{$_GET['symbol']}'>");

// try to get quote
$handle = @fopen("http://download.finance.yahoo.com/d/"
 . "quotes.csv?s={$_GET['symbol']}&f=e1l1hg", "r");
if ($handle !== FALSE)
{
    $data = fgetcsv($handle);
    if ($data !== FALSE && $data[0] == "N/A")
    {
        print("<price>{$data[1]}</price>");
        print("<high>{$data[2]}</high>");
        print("<low>{$data[3]}</low>");
    }
    fclose($handle);
}

// output root element's end tag
print("</quote>");
```

Notice that now we're declaring a MIME type of XML and we're spitting out a few nodes of XML data, though not a complete, well-formed document. That's fine, as long as we know what we're looking for in `ajax8.html`:

```
// get XML
var xml = xhr.responseXML;

// update price
var prices = xml.getElementsByTagName("price");
if (prices.length == 1)
{
    var price = prices[0].firstChild.nodeValue;
    document.getElementById("price").innerHTML = price;
}

// update low
var lows = xml.getElementsByTagName("low");
if (lows.length == 1)
{
    var low = lows[0].firstChild.nodeValue;
    document.getElementById("low").innerHTML = low;
}

// update high
var highs = xml.getElementsByTagName("high");
if (highs.length == 1)
{
    var high = highs[0].firstChild.nodeValue;
    document.getElementById("high").innerHTML = high;
}
```

Here, as you can see, we're accessing `responseXML` as a DOM document itself, which has been automatically loaded up into memory by our XMLHttpRequest object. Then we're going to search it for the tags we know are there and, since the `getElementsByTagName()` method returns an array, we're going to ask for the first such tag and access the `firstChild` thereof.

- You can verify for yourself that if you change the MIME type to plain text, you can't parse it like an XML file and our `ajax8.html` won't function properly.
- In `ajax9.html`, we're going to revert back to accessing `quote1.php` and proceed to build our own DOM nodes using the following code:

```
// insert quote into DOM
```

```
var div = document.createElement("div");
var text = document.createTextNode(symbol + ": " + xhr.responseText);
div.appendChild(text);
document.getElementById("quotes").appendChild(div);
```

We're just creating a new DOM element of type `div` and then appending, as a child, the `responseText` as a new text node. Finally, we append the `div` as a child of a `div` in our whole DOM.

- So we were discussing earlier the fact that using the `innerHTML` property is not technically standards-compliant. Technically, what we should be doing is simply grabbing the data from the server and then rendering it client-side using code like the above. The problem with that approach is that it might end up being much slower. The speed with which the user's browser interprets JavaScript is variable, whereas the speed with which our server can format the data is something more or less constant—or at least known to us. These days, the arguments over browser speed don't pertain only to the speed with which they load pages, but also the speed with which they interpret JavaScript.

- As an interesting sidebar, the YUI library has a widget called a DataTable which automatically implements pagination and sorting of a table. However, the YUI developers have found that creating DOM nodes on the fly tends to be pretty slow, so they actually generated code that would allow nodes to be reused in the interest of performance optimization.

- As we mentioned before, we have another format for passing data across the wire known as JSON, or *JavaScript Object Notation*. Basically, we're taking a JavaScript object, serializing it—or converting it to a string—and then sending it to be unserialized by the file that requested it. Our serialized object looks something like this, as displayed in `ajax10.html`:

  ```
  { price: 379.30, high: 390.65, low: 375.89 }
  ```

  Not too difficult, it's just a series of key-value pairs separated by colons. How do we parse this? Here's one way:

  ```
  // evaluate JSON
  var quote = eval("(" + xhr.responseText + ")");
  ```

  With this call to `eval()`, we're evaluating the `responseText` as JavaScript, so it's being loaded up into memory as an object again. Note that we have to put parentheses around it. In other contexts, `eval()` can be a security concern, but since we're passing it our own code, presumably this is a safe context. Now, when we want to access the price that's returned, we do so with our dot notation:

  ```
  var text = document.createTextNode(symbol + ": " + quote.price);
  ```

- All of this is pretty simple. But how do we go about outputting the JSON? First, we'll need to change the MIME type:

  ```
  // set MIME type
  header("Content-type: application/json");
  ```

  Of course, we could simply print out the JSON we need manually, as we do in `quote6.php`:

  ```
  // output JSON
  print("{ price: $price, high: $high, low: $low }");
  ```

  But a sleeker way of doing this would be to leverage PHP's built-in function called `json_encode()`. First, we'll need to define a class for our stock:

  ```
  // defines a stock
  class Stock
  {
  public $price;
  public $high;
  public $low;
  }
  ```

  If you're not familiar with object-oriented programming, don't be alarmed. It's not too difficult, especially in this context, where our class is effectively an array. If we call `print_r()` on it, we get the following:

  ```
  Stock Object
  (
      [price] => 378.11
      [high] => 379.10
      [low] => 370.30
  )
  ```

  Not too scary, right? Looks just like an array. Once we've created our object, we only need to pass it to `json_encode()` to have it properly formatted.

- As we've said before, the great thing about JavaScript is the multitude of libraries that are available. Using the YUI library, we can whittle our Ajax request down to the following:

  ```
  // make call
  YAHOO.util.Connect.asyncRequest("GET", url, { success: handler });
  ```

  Magic! And next time, more magic!