Computer Science 75
Fall 2009
Scribe Notes

Lecture 9: November 16, 2009
Andrew Sellergren

Contents

1 Announcements (0:00–2:00) 2

2 Ajax (2:00–105:00) 2
2.1 Introduction . . . 2
2.2 Data-driven Websites . . . 2
2.3 Getting Started . . . 3
2.4 Examples . . . 5
2.4.1 `ajax1.html` . . . 5
2.4.2 `ajax2.html` . . . 10
2.4.3 `ajax3.html` . . . 11
2.4.4 `ajax4.html` . . . 11
2.4.5 `ajax5.html` . . . 12
2.4.6 `ajax6.html` . . . 13
2.4.7 `ajax7.html` . . . 13
2.4.8 `ajax8.html` . . . 14
2.4.9 `ajax9.html` . . . 16
2.4.10 `ajax10.html` . . . 16
2.4.11 `ajax11.html` . . . 17
2.4.12 `ajax12.html` . . . 17

Computer Science 75
Fall 2009
Scribe Notes

Lecture 9: November 16, 2009
Andrew Sellergren

1 Announcements (0:00–2:00)

- Check out The CS 50 Store! Sure, why not?

2 Ajax (2:00–105:00)

2.1 Introduction

- Notice the lower-case letters in "Ajax." Formerly, it was an acronym that stood for "asynchronous JavaScript and XML," but these days, since data can be returned from requests in many more formats than just XML, its meaning as an acronym has lost significance.
- Ajax is a technology by which HTTP requests can be made after a page has already loaded in order to seamlessly retrieve additional data without the need for a page refresh. Sometimes a page refresh will do the job just fine—for example, in David's Shuttleboy app, the page refreshes every 60 seconds or so rather than retrieving data via Ajax. Soon, however, David will update Shuttleboy to use Ajax.
- As we mentioned last time, Kayak is a an excellent example of how to use Ajax to improve a user interface. Google Maps one-upped MapQuest when it first came out by implementing its map with a smooth drag feature. The map tiles surrounding a location are downloaded on demand to create this smooth effect; we can verify this using Live HTTP Headers.
- Luckily, for Project 3, Google will be handling one of the Ajax requests for us. Using its GClientGeocoder object, you'll be able to geocode the address a user inputs. Google will send the request and return you a response; all you need to do is handle that response appropriately when it comes.

2.2 Data-driven Websites

- One reason that Ajax is in vogue these days is that data-driven websites are very popular. Data is becoming increasingly easy to gather from many different sources. For his own part, David has endeavored to make data easier to gather data from some of his pet projects like HarvardMaps and HarvardEvents. One of the recent additions to this list is HarvardFood which gathers data from the Harvard University Dining Services (HUDS) menus by screen-scraping their website. Screen-scraping can be accomplished by actually traversing the DOM (although this becomes more difficult if the site isn't valid XHTML) or by using regular expressions. If we view the source of the HUDS menu, we see that two CSS classes might be of use to us in trying to grab item names: `item_wrap` and `menu_item`. Recalling our discussion of XPath, we can search for nodes with these attributes, the former inside of the latter, with a query that might look something like this:

```
xpath("//div[@class='menu_item']/div[@class='item_wrap']/span/a");
```

Great, that didn't take long at all. But what's the big danger here? If HUDS decides to change how they display their data, our screen-scraper is probably going to break. What's our big assumption? We're assuming that the page is valid XHTML. When we plug it in to the W3C validator, we quickly see that it's not. If we skim the list of reported errors, we see that their JavaScript hasn't been placed in a `CDATA` section and their ampersands haven't been escaped, among other errors. Ideally, all these errors will be small ones we can fix. In fact, there exists a program called Tidy that will assist us in doing so. Formerly, it was a command-line utility, but now there also exists a PHP version. What this tool will do is fix small errors like unclosed tags and unescaped ampersands so as to output a valid XHTML document. Once we've passed the HUDS menu to Tidy, we can create a new SimpleXML DOM and perform our original XPath query on it.

- Once we've written our script to grab the menu data, we can automate the process by creating a `cron` job. `cron` is a Linux utility which schedules tasks to be performed at designated times. The `cron` job to grab the menu data runs nightly, for example.
- In fact, by studying the site a little more, we can figure out that it supports a GET string that includes a `date` parameter and a `meal` parameter. So we can grab breakfast, lunch, and dinner even for future dates if we form our request properly.
- The HarvardFood API is what's called a RESTful interface, which basically means that it responds to queries that are formed as URLs. So to retrieve the breakfast menu data for 11/11/09 in CSV format, simply make a request to a URL like so:

  ```
  http://food.cs50.net/api/1.0/items?date=2009-11-11&meal=Breakfast&output=csv
  ```

  Recall that Twitter has a RESTful interface for its API. We've implemented similar APIs for HarvardEvents, HarvardMaps, etc. Check out the full list here.
- Question: are there other types of API interfaces out there? Yes, SOAP and XML RPC are two. Although they're more heavy-handed than REST, they do allow more specificity in terms of data types and there are many libraries and toolkits available to make the job easier.

2.3 Getting Started

- Using Ajax is all about manipulating the XHTML DOM. You might be adding nodes, deleting nodes, or updating nodes, but ultimately you'll be affecting the overall structure of the document. Take a look at these references below for more about the DOM:

  - HTML DOM Reference
  - DOM (Document Object Model) Reference
  - DOM objects and methods

- The object which actually implements Ajax in JavaScript is the `XMLHttpRequest` object. It is implemented slightly differently across different browsers, but there is some core functionality which is common to all. Take a look at this list of common methods:
  - `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)`
- 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.

- The basic flow for an Ajax request is as follows: instantiate an XMLHttpRequest object, open a request, send a request, wait for the response, then pass the response to a handler function.
- 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)

2.4 Examples

2.4.1 `ajax1.html`

- 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. If we access this URL directly, we see that it outputs to the browser a string representing the stock price. Even `ajax1.html`, however, isn't much fancier than this:

```
<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.

- 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.

- To construct our URL, we use the `+` sign to concatenate whatever the user inputted into the `symbol` form 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.

- Recall that because of the Same Origin Policy (SOP), the URL we request must be on our own server. This is a security measure to prevent

Distributed Denial of Service (DDOS) attacks. If not for the SOP, a malicious developer would be able to create a site which, when visited, would spawn thousands of Ajax requests to other web servers and thereby potentially crash them. In the case of Project 3, Ajax requests for more map tiles and for geocoding are allowed because the code which makes the requests is hosted on Google's servers. If you try to request via Ajax a URL on a different server, you'll get a browser error of some kind. For Project 3, we'll "circumvent" the SOP by establishing a PHP proxy. This isn't really thwarting it, though, since all of the requests for Google News will then originate from our server, which Google can easily blacklist.[1] If the requests were created via JavaScript, they would appear to come from multiple clients' IP addresses, not the server's single IP address. It would be much more difficult, if not impossible, to blacklist all these IP addresses to defend against the DDOS attack.

- One advantage of JavaScript is *asynchronicity*. With JavaScript, we can call a function and have it call another function—a handler—whenever a response is returned. That way, we can continue executing the rest of our code without having to wait for this response. 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.

- As a sidenote, pay attention to capitalization. JavaScript methods are case-sensitive!

- Question: how can JavaScript and PHP communicate with one another? Well, we could've just as easily assigned our Ajax `responseText` to a JavaScript variable instead of passing it directly to the `alert` function. If

[1] As a sidenote, they do, in fact, blacklist us quite frequently which is why you'll occasionally see dummy results returned by your requests for Google News. We've set up a proxy for your proxy!

we wanted to manipulate the stock price as a number, we would have to pass it to the `parseFloat` function first.

- Question: what is `cron`? To execute commands on a specified schedule, start by typing `crontab -e` at the command line. This will open up a configuration file in which you write lines like the following:

  ```
  */5 * * * *  /home/user/test.pl
  ```

  This will cause the `test.pl` script to be executed every 5 minutes, every hour, every day of the month, every month, every day of the week (represented from left to right by the `*`'s). We use `cron` jobs, for example, to send SMS reminders for office hours to our teaching fellows and to update our MySQL database containing the HUDS menus.

- Question: how can Firebug can help us debug? Let's introduce an error into our JavaScript: we'll make a mistake in capitalization. When we try to look up a stock quote now, we see that the symbol field actually gets cleared when we click Get Quote. This means that the form is actually submitting despite our `return false` in the `onsubmit` attribute. This is a good indication that we have a JavaScript syntax error somewhere.

  If we open up the Firebug window, we see that it actually doesn't immediately catch our errors. So let's put a breakpoint next to our `try` statement on line 34 by viewing it in the Script tab and then clicking the space to the left of it which adds a red dot. Now when we click Get Quote, execution of our JavaScript will halt at this line (as denoted by a yellow arrow inside the red dot). On the righthand side of the Firebug window, we can click the Watch tab to see the values of variables that are currently in scope. We see that `xhr` is undefined and `url` is undefined.

  On the upper right of the lefthand side of the Firebug window, there are four buttons: one blue arrow followed by three yellow arrows. The blue arrow will cause our script to Continue, meaning execution will proceed until the next breakpoint. This is probably too fast for us. Next to is is Step Into followed by Step Over and Step Out. If the current line is a function call, Step Into will walk us inside the function and continue execution. Step Over will walk us past the function call to the next line. And if we're currently inside a function, Step Out will jump us out of it to the caller function. Here, it doesn't so much matter, but we'll use Step Over.

  When we click Step Over, we jump all the way to the line that begins `if (xhr == null)`. That's because the `catch` statement didn't need to execute since the first instantiation attempt succeeded. On the righthand side, we can see that `xhr` is no longer undefined.

  When we Step Over the line that defines `url`, we see that something's wrong because it jumps back to the beginning of our function and our

page refreshes. Unfortunately, this is our only take-away from Firebug (although usually it's more helpful). At least we figured out that the problem lies on line 51 because of our incorrect capitalization of the `getElementById` method.

2.4.2 `ajax2.html`

- 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` of `price`.

- 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. Nothing really all that special.

- So this approach is more appealing than the alert window since it's persistent, but it also allows a user to edit what appears in the text box, which is not ideal.

2.4.3 `ajax3.html`

- 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 high-performing) technique. Frankly, the standards-compliant way of editing the DOM by calling cross-browser methods to add and update nodes is cumbersome and slow. Generally, it's much faster to generate XHTML on the server-side and cramming it into the DOM.

- If we open up Firebug, we can watch the previous content of this `span` be entirely clobbered in realtime. Conveniently, Firebug shows the current state of the DOM. If we go to View Source, we won't see the stock price anywhere in our XHTML.

- Question: is it possible to send multiple Ajax requests? Yes, but you need to make sure to avoid race conditions. We'll come back to that.

- Question: could we use the `responseXML` property instead of `responseText`? Yes, but we'd have to make sure to send back valid XHTML from our proxy. This will automatically be parsed by our browser to populate the `responseXML` property. But in this case we don't so much care about that extra step, so we prefer to use `responseText`.

2.4.4 `ajax4.html`

- For `ajax4.html`, we'll be making use of a different proxy. Let's take a look at `quote2.php`, which will return a little more data than before:[2].

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

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

```
     * 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.

- In `ajax4.html`, we've taken a bit of a step back in that we're inserting into a text box again, but at least we see that Ajax can return more than just a single stock price with one call.

2.4.5 `ajax5.html`

- If you take a look at `ajax5.html`, you'll see that it prints extra data, as before, in the `innerHTML` of a `span`. Take a look at how we're returning this data from a new proxy, `quote3.php`:

```
<?
    /**
     * quote3.php
     *
     * Outputs price, low, and high of given symbol as text/html.
     *
     * David J. Malan
     * Computer Science E-75
     * Harvard Extension School
     */

    // 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]}");
            print("<br />");
            print("High: {$data[2]}");
            print("<br />");
            print("Low: {$data[3]}");
        }
        fclose($handle);
    }
?>
```

Now we're returning XHTML instead of plaintext, so we use `<br />` tags to create newlines rather than `\n` characters.

- As a sidenote, Shuttleboy achieves page refreshes with the following line:

  ```
  <meta http-equiv="refresh" content="23" />
  ```

  The value of `content` is actually changed dynamically—we check the value of the server time and calculate the number of seconds until the next minute.

2.4.6 `ajax6.html`

- Again, we'll be using a new proxy, `quote4.php`, but the only real difference is that we added a call to `sleep()` to simulate a slow server response.
- In `ajax6.html`, we'll be providing some feedback to the user to let him know that the response is on its way. While the request has not yet returned, we write "Looking up symbol..." We accomplish this with the following line:

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

2.4.7 `ajax7.html`

- In `ajax7.html`, we actually implement 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 difference between the `display` and `visibility` CSS properties is that the former takes up no space on the page when it is set to `none` but the latter reserves a chunk of space for the element when it is set to `hidden`.
- Incidentally, you can create your own progress bar here.

2.4.8 `ajax8.html`

- 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. If we hit `quote5.php` directly in our browser, we see that we get output that looks something like this:

```
<quote symbol="goog">
    <price>576.28</price>
    <high>576.99</high>
    <low>572.78</low>
</quote>
```

- Now let's take a look at the new code 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.

2.4.9 `ajax9.html`

- 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.

2.4.10 `ajax10.html`

- 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 commas. But how do we go about outputting the JSON? First, we'll need to change the MIME type of our proxy:

```
// 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 }");
```

- In order to convert our JSON to an actual JavaScript object in `ajax10.html`, we use the following line of code:

```
// 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);
```

Note we're taking a more standards-compliant approach by calling `createTextNode` rather than clobbering the `innerHTML` property.

2.4.11 `ajax11.html`

- One word: debuggin'.[3]

2.4.12 `ajax12.html`

- 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 });
  ```

- Know that outputting JSON can be much simpler than printing it manually. In PHP, the `json_encode` function, for example, will convert an associative array (or almost anything) to JSON. There are other libraries to do the same in other languages.

[3] Technically that was three, I guess. Now nine. Dammit.