Computer Science 75
Spring 2009
Scribe Notes

Lecture 10: April 20, 2009
Andrew Sellergren

Contents

Computer Science 75 — Lecture 10: April 20, 2009
Spring 2009 — Andrew Sellergren
Scribe Notes

1 Welcome (0:00–2:00)

- A quick welcome to Dan Armendariz!
- Dan co-lectures the summer version of E-75 and also teaches Computer Science E-7: Exposing Digital Photography.

2 jQuery (2:00–50:00)

- You're familiar by now with Google's use of Ajax to improve Google Maps. Gmail and Google Calendar also rely heavily on Ajax for their core functionality. Outside of Google, Apple makes great use of Ajax, particularly in their MobileMe service.
- The much-exalted list of JavaScript libraries (which is by no means comprehensive):
  - Dojo
  - Ext JS
  - jQuery
  - MooTools
  - Prototype
  - script.aculo.us
  - YUI
- With any of these libraries, typical use involves downloading a large JavaScript file which you link to in your `head` tag.
- If you take a look at the course website for Computer Science E-7, you'll see that you can click on the main image and dynamically change the theme of the whole homepage. How might we accomplish this using JavaScript?
- Take a look at the Color Test page. Enter in four colors in hexadecimal format and watch the CSS properties of the website change before your very eyes. Are we using Ajax for this? Not exactly. Just plain old JavaScript:

  ```
  if(!testColor(page)) {
      alert("The page color is not a valid hexadecimal color.");
      return false;
  }
  ```

  Within the function `changeTheme`, the above snippet of code is repeated several times. As you might've guessed, we're just testing to see if the color provided by the user is a valid hexadecimal number. Next, the real substance:

```
var headID = document.getElementsByTagName("head")[0];
var newNode = document.createElement("link");
newNode.type = "text/css";
newNode.rel = "stylesheet";
newNode.href = "/styles/colors.php?p=" + page + "&c=" + content +
               "&h=" + highlight + "&t=" + text;
newNode.media = "screen";
headID.appendChild(newNode);
```

Here, we're finding the `head` element of the page and creating a new `link` tag to place within it. This `link` tag will contain a new stylesheet, generated dynamically by our `colors.php` file. Because stylesheets are *cascading*, this stylesheet will override the others above it when we place it below the other `link` tags. If you have Firebug, you can see it will be highlighted once it's been added to the DOM.

- In `jquery1.html`, you can see that we have three buttons, two of which purportedly enable and disable all of the buttons. This is actually quite easy to accomplish using jQuery:

```
function disableAllButtons() {

    // add the attribute "disabled" with value "disabled"
    // to all submit buttons
    $(":submit").attr("disabled","disabled");

}
```

Although it might look quite cryptic at first, the syntax for jQuery is quite simple. Here, we're simply searching for all elements in the DOM of type `submit`. Then we're adding the `disabled` attribute to them and setting its value to `disabled`.

- Note that nearly all of jQuery's functionality, minus a few plugins here and there, is packaged into a single JavaScript file. This is in contrast to the YUI library, for example, which has been splintered into multiple files along the way. As with anything, both have their advantages and disadvantages.

- The dollar sign notation is common to multiple different JavaScript libraries. Here, it references the jQuery object. Note that if you use multiple libraries on a single page, all of which use this dollar sign notation, at least one of the libraries will have to default to alternate notation.

- Of course, we'll need to take steps forward, now, because when we click Disable All, we have no way of re-enabling the buttons, since the Enable All button is disabled along with the others.

- In `jquery2.html`, we disable all the buttons except the Enable All button like so:

  ```
  function disableAllButtons() {

      // add the attribute "disabled" with value" disabled"
      // to all submit buttons
      $(":submit").attr("disabled","disabled");

      // that might have been premature, let's re-enable
      // the "enable all" button.
      $(":submit:eq(1)").attr("disabled", "");

  }
  ```

  Of course, this is a somewhat naive approach since it would make more sense to avoid disabling the Enable All button in the first place instead of disabling it and then re-enabling it.

- The `enableAllButtons()` function exhibits another way of searching through the DOM:

  ```
  function enableAllButtons() {

      // from the example above, we could have done:
      // $(":submit").attr("disabled", "");
      // But we can use css-style selectors to
      // find the ones we want also:

      $("input[type='submit']").attr("disabled", "");

  }
  ```

  As you can see, this type of query is more XPath-like than the first.

- Note that JavaScript doesn't really have classes or constructors, per se, but we can mimic that behavior quite well.

- In `jquery3.html`, we've pared down the code a little bit using this XPath-like search. Now, we're disabling all but the Enable All button from the start:

  ```
  function disableAllButtons() {

      // unlike xpath, there is no '@' character before attributes

      $("input[name!='submit1']").attr("disabled", "disabled");
  ```

```
}

function enableAllButtons() {

    $(":submit").attr("disabled", "");

}
```

- We can take a quick look back at the alternative to using JavaScript libraries, which is to handle the XMLHttpRequest object ourselves. This quickly becomes cumbersome, as you've already seen:

```
// an XMLHttpRequest
var xhr = null;

function doAjax(id) {

    // 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 = "json.php?id=" + id;

    // get quote
    xhr.onreadystatechange =
    function()
    {
        // only handle loaded requests
        if (xhr.readyState == 4)
        {
            if (xhr.status == 200)
```

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

                    // show JSON in textarea
                    document.getElementById("bar").innerHTML = data.servertime;

                }
                else
                    alert("Error with Ajax call!");
            }
        }
        xhr.open("GET", url, true);
        xhr.send(null);

    }
```

Most of this syntax will be familiar to you from previous lectures, but just for practice, try working through it again and making sure you understand it in this context in which we're asking for the server time.

- What's the equivalent if we use jQuery? A whopping 4 lines of code:

```
function doAjax(id) {

    $.getJSON("json.php?id=" + id,
            function(data) {
                $('#bar').html(data.servertime);
            });

}
```

- Note that the `id` argument passed to `doAjax()` is all but meaningless. It's merely meant to show you that you *can* pass arguments as necessary and concatenate them to your Ajax requests using jQuery. Here, we're querying the same PHP file as before, which, just for reference, has the following lines of code:

```
<?

$myArray = array(

            'servertime' => date("l \\t\h\e jS @ h:i:s A"),
            'random' => rand(0,100)

        );
```

```
echo json_encode($myArray);

?>
```

Going back to the `doAjax()` function, we can see that we're again searching through the DOM for a particular element, this time one with the `id` of `bar`. The data that's returned from our Ajax request is being passed to an anonymous function (the second argument of the `getJSON()` method) as the variable `data`. Within that function, we access the `servertime` property of the `data` object.

- One last example, `jquery5.html`, demonstrates how we can modify multiple DOM elements concurrently:

```
function doAjax(id) {

    $.getJSON("json.php?id=" + id,
            function(data) {
                $('.foo').each(
                    function(i) {
                        $(this).html(data.servertime);
                    })
            });

}
```

3 YUI and More (50:00–85:00)

- Now that we have the foundation for Ajax, what can we do with it? One well-known example is autocomplete functionality. Check out YUI's example here. If you take a look at the snippets of code posted below the example, you'll see that what it boils down to is sending a `GET` string to a PHP file. This `GET` string is updated as the user types in the search box. The hard part, really, is writing the back-end PHP search which will look up results in a database.
- Realize, of course, that in the YUI example, the URL they were requesting was on their own servers. This goes back to the same origin policy we discussed before. If you implement the YUI AutoComplete widget on your own site, you'll be responsible for writing the PHP file that is requested. Yahoo! probably doesn't have much in the way of examples for how to perform the back-end search.
- Let's take a look at an example which uses YUI DataTable. First, the back-end:

```
<?
/*
 *
 * marathon.php
 *
 * Returns details from the 2009 Boston Marathon results
 * in JSON format.
 *
 * Dan Armendariz
 * Computer Science E-75
 * Harvard Extension School
 *
 */

// set the content type
header("Content-type: application/json");

// define the data's structure
$desc =
    array('race','pos','name','country','time');

// 20 Apr 2009 results from http://www.baa.org/
$results = array( places => array(
    array('Women Open','1','Salina Kosgei','Kenya','2:32:16'),
    array('Women Open','2','Dire Tune','Ethiopia','2:32:17'),
    array('Women Open','3','Kara Goucher','USA','2:32:25'),
    array('Women Open','4','Bezunesh Bekele','Ethiopia','2:33:08'),
    array('Women Open','5','Helena Kirop','Kenya','2:33:24'),
    array('Women Open','6','Lidiya Grigoryeva','Russia','2:34:20'),
    array('Women Open','7','Atsede Habtamu','Ethiopia','2:35:34'),
    array('Women Open','8','Colleen S. De Reuck','USA','2:35:37'),
    array('Women Open','9','Alice Timbilili','Kenya','2:36:25'),
    array('Women Open','10','Alina Ivanova','Russia','2:36:50'),
    array('Men Open','1','Deriba Merga','Ethiopia','2:08:42'),
    array('Men Open','2','Daniel Rono','Kenya','2:09:32'),
    array('Men Open','3','Ryan Hall','USA','2:09:40'),
    array('Men Open','4','Tekeste Kebede','Ethiopia','2:09:49'),
    array('Men Open','5','Robert Cheruiyot','Kenya','2:10:06'),
    array('Men Open','6','Gashaw Asfaw','Ethiopia','2:10:44'),
    array('Men Open','7','Solomon Molla','Ethiopia','2:12:02'),
    array('Men Open','8','Evans Cheruiyot','Kenya','2:12:45'),
    array('Men Open','9','Stephen Kiogora','Kenya','2:13:00'),
    array('Men Open','10','Timothy Cherigat','Kenya','2:13:04'),
    array('Women Wheelchair','1','Wakako Tsuchida','Japan','1:54:37'),
    array('Women Wheelchair','2','Diane Roy','Canada','2:01:27'),
    array('Women Wheelchair','3','Shirley S. Reilly','USA','2:04:54'),
    array('Men Wheelchair','1','Ernst Van Dyk','South Africa','1:33:29'),
    array('Men Wheelchair','2','Masazumi Soejima','Japan','1:36:57'),
    array('Men Wheelchair','3','Roger Puigbo Verdaguer, Sr.','Spain','1:37:47')
));

foreach ($results['places'] as &$result) {
    $result = array_combine($desc,$result);

}

print(json_encode($results));

?>
```

Notice we've taken the brute-force method of just hardcoding in the results of this year's Boston Marathon. More sophisticated implementations would obviously rely on well-structured data. The `array_combine()` function is called so that `race`, `pos`, `name`, `country`, and `time` become the keys for each of the rows of our table.

- Now check out the front end here. Notice that the YUI DataTable makes certain things very easy, including sorting and pagination. Click on any of the columns to sort by it and click on any of the names of the runners to open a Google search for him or her.

- How do we implement this? First, by linking to several YUI libraries, both JavaScript and CSS. Second, with the following lines of JavaScript:

```
// on load
YAHOO.util.Event.addListener(window, "load", function() {

    populateTable = new function() {

        // modify country cells to provide a link to the Wikipedia page.
        var countryUrl = function(cell, record, column, data) {
            cell.innerHTML = "<a href='http://en.wikipedia.org/wiki/"
            + data + "' target='_blank'>" + data + "</a>";
        };

        // modify name cells to provide a link to Google search results.
        var googleUrl = function(cell, record, column, data) {
            cell.innerHTML = "<a href='http://www.google.com/search?q="
            + data + "' target='_blank'>" + data + "</a>";
        };


        // define the columns in the table
```

```
            var columns = [
                {key:"race",   label:"Race",     sortable:true},
                {key:"pos",    label:"Position", sortable:true,
                                                 formatter:"number"},
                {key:"name",   label:"Name",     sortable:true,
                                                 formatter:googleUrl},
                {key:"country",label:"Country",  sortable:true,
                                                 formatter:countryUrl},
                {key:"time",   label:"Time",     sortable:true}
            ];


            // create a new data source object whose data comes from marathon.php
            // and is returned from the server with type JSON
            var dataSource = new YAHOO.util.DataSource("marathon.php?");
            dataSource.responseType = YAHOO.util.DataSource.TYPE_JSON;

            // if a request is already in progress, wait until it's done before
            // performing another request.
            dataSource.connXhrMode = "queueRequests";

            // define the schema of the data as returned from the server
            dataSource.responseSchema = {
                resultsList : "places",
                fields: ["race","pos","name","country","time"]
            };

            // define the table as living within a div with id results with
            // columns and a datasource we have just defined
            dataTable = new YAHOO.widget.DataTable("results", columns,
                    dataSource);

            // finally, perform the Ajax request to populate the table with data
            dataSource.sendRequest();

        };
    });
```

The first important chunk we'll focus on is the definition of the variable `columns`. Notice that each of the columns in our DataTable corresponds to an index into the array in our PHP file. Here's where we tell the YUI library about that association. The `columns` variable is actually an array of JavaScript objects with predefined keys, namely `key`, `label`, `sortable`, and `formatter`. Pretty self-explanatory.

- The `formatter` key is where we can tell YUI to use a function of our own definition, namely `googleURL` and `countryURL`, which make the contents of the column's cells into links to Google and Wikipedia, respectively.

- Below the `columns` definition, we have several other important snippets of code we should examine. First, we're instantiating a new YUI DataSource and telling it that the response we'll be getting will be of type JSON. Then, we're also telling YUI to queue our Ajax requests if several are made close together. Below that, we have to define the response schema of our DataSource. This may sound a little cryptic, but know that it's an extension of our earlier definition of the `columns` array. And trust that it's all in the YUI documentation somwhere! Finally, we tell YUI that the DataTable will live in the `div` that has `id="results"`. That's it!

- Recall that the data in our PHP file was confined within an inner array named `places`. This corresponds to the `resultsList` property of our response schema.

- Wanna see something else cool? I know you do. Here's jQuery Cycle. It has some pretty cool transition effects for displaying photos. Using jQuery Cycle along with zenphoto, we can create some pretty cool projects. Take a look at this slideshow for E-7. But, shh, don't tell the students about it, it's a surprise!

- One technique that this slideshow makes use of is called *lazy loading*. This is a method whereby content is only loaded as needed in order to reduce download times. Not all of the photos in this slideshow are loaded right away, but only right before they are to be displayed.

- jQuery Cycle uses a larger `div` that contains multiple `div` elements within it. The bottom `div` elements remain hidden until the transition is ready to be made. Then the topmost `div` disappears and the one below it takes its place. Pretty neat, huh?

- The innerworkings of this slideshow can be summarized as follows: each time we change slides, we fire an event that calls `fetchSlide` to get the next. Once this next slide is fetched, its properties aren't yet defined the way we want them. So we search for the slides that don't have these properties defined and then we define them. A little vague, I know, but take a look at the source yourself and see if you can figure out what's going on!