Computer Science 50
Fall 2010
Scribe Notes

Week 9 Wednesday: November 3, 2010
Andrew Sellergren

**Contents**

Computer Science 50
Fall 2010
Scribe Notes

Week 9 Wednesday: November 3, 2010
Andrew Sellergren

## 1 Announcements and Demos (0:00–10:00)

- This is CS50.
- 0 new handouts.
- Check out a fellow classmate's homepage which puts up a good fight for the most annoying website in the world!
- According to David McCandless, who recently gave a TED talk on data visualization, most Facebook breakups take place after Valentine's Day and during Spring Break. We mention this because data visualization will be an important part of Problem Set 8 as we make use of the Google Charts API. Also check out HarvardEnergy which dynamically generates charts using JavaScript after syncing its data with a database using PHP and MySQL. JavaScript's model is based on events (e.g. clicking and dragging) triggered by the user and listener functions which respond to them.
- Realize that for Problem Set 7 when we say that your code must be valid, we're referring to your HTML, not your PHP, passing the W3C standards test. To test it out, copy and paste the HTML output of your application into the W3C Validator. Because your pages require validation, you won't be able to simply point the W3C Validator to your URLs. You can also avail yourself of the Web Developer Firefox plugin which will allow you to locally validate your HTML.

## 2 FAQs (10:00–20:00)

### 2.1 HTTP Errors

- If you were unlucky enough to visit Google Calendar this morning during its unplanned downtime, you might've seen a page that said "404 Not Found." The 404 is an HTTP error code which generally indicates that the file that was requested can't be found on the server. In this case, it was probably some other configuration error that caused Google Calendar to go down, but in the context of your own website, it might mean that you misspelled the name of the page you were hoping to visit.
- Here's a short, non-exhaustive list of HTTP errors:
  - 401 - unauthorized
  - 403 - forbidden
  - 404 - not found
  - 500 - internal server error

If a site requires a username and password and you've provided the wrong ones, you might be met with a 401 error. 403 usually indicates an error with file permissions, i.e. the file has not been `chmod`'ed properly. 500 is usually indicative of one of a variety of more serious server configuration errors.

### 2.2 Sessions

- HTTP sessions are a handy tool for persisting data across a user's time while logged in to your website. You can store whatever data you want in the `$_SESSION` superglobal variable and that data will be available on the server side as long as the session has not expired, for example when the browser window is closed.
- The distribution code for Problem Set 8 makes use of the `$_SESSION` variable to store the user ID of the person who just logged in.
- How does the web server know what data to populate `$_SESSION` with? Generally, it relies on the user's cookie and then looks up the corresponding temporary session data stored on the server. This is exactly the information that is being stolen when a session is hijacked by Firesheep. You might have noticed that CS50 is now using SSL exclusively for logged-in pages which will thwart this kind of attack.

### 2.3 Interpolation

- One gotcha with interpolation in PHP (i.e. variables enclosed in double quotes will automatically be substituted with their values) pertains to arrays. To use interpolation with arrays, you must enclose the array name in curly braces like so:

```
$sql = "SELECT * FROM users WHERE uid = {$_SESSION["uid"]}";
```

  You could also use the concatenation operator to get around this interpolation problem:

```
$sql = "SELECT * FROM users WHERE uid = " . $_SESSION["uid"];
```

### 2.4 PHP Documentation

- PHP's documentation available at php.net will be invaluable to you as you learn the language. In particular, the function reference pages are extremely useful. They will provide you with function prototypes that specify return and argument types (even though PHP is not strictly typed) as well as example code and snippets from users.

## 3 Cool Demos (20:00–34:00)

### 3.1 HTML5

- One of the intents of HTML5 is to preempt the need for browser plugins like Flash in order to play movies and games. Check out these HTML5 demos to see what is possible.

### 3.2 JavaScript

- Also check out Rumpetroll, a chat room implemented entirely in JavaScript![1]
- Finally, as a teaser for where we're going, check out the WebGL Aquarium, also implemented entirely in JavaScript.

### 3.3 PHP

- This demo is perhaps only impressive to us because we just finished writing dozens of lines of code in order to implement a spellchecker in C. Take a look at how easy it is to implement the `load` function in PHP:

```
$hashtable = array();

function load($dict)
{
    for (file($dict) as $word)
        $hashtable[$word] = TRUE;
}
```

  The `array` function in PHP declares a so-called associative array which is really just an implementation of a hash table.

- `check` is just as easy to implement:

```
function check($word)
{
    if ($hashtable[$word])
        return TRUE;
    else
        return FALSE;
}
```

- The downside of using PHP for our spellchecker is that because PHP is an interpreted language rather than a compiled language like C (i.e. source code is translated into binary at runtime for PHP as opposed to before

[1] The choice of avatars is somewhat unfortunate. Or hilarious, depending on how you look at it.

runtime for C), the runtime will be much longer. However, the tradeoff is that development time is much shorter. If we run the staff solution for Problem Set 6, implemented in C, against a PHP version of speller while checking `kjv.txt`, we get runtimes of about 0.5 seconds and 2 seconds, respectively.

- For your own edification, check out the entirety of the PHP implementation of speller here. Note that at the top of `speller` there is the following line of code:

  ```
  #!/usr/bin/php
  ```

  This line of code allows us to drop the `.php` extension from the file and make the file itself executable. Whereas normally, we'd have to run a command like `/usr/bin/php speller.php` so that the operating system knows what program to execute our file with, this line of code called a shebang shortcuts this process so that we can simply run `speller` from the command line. In the context of web development, the web server makes the assumption that `.php` files should be passed to the PHP interpreter before being presented to the user.

## 4 JavaScript (34:00–74:00)

- The `script` HTML tag is not one we've had occasion to examine so far. Generally speaking, a *script* refers to an program coded in an interpreted language. Thus, the `script` tag tells the HTML interpreter that a program is being introduced. Although programs in other languages are possible, the `script` tag usually introduces a JavaScript program.

- In years gone by, JavaScript was really only capable of annoying things like popping open alert windows. Nowadays, it has become incredibly powerful as a programming language in its own right. On HarvardEvents, for example, JavaScript is used to display a course information dropdown when the plus icon is clicked next to the course name. This drop down is almost instantaneous because the data has already been retrieved from the server and thus there is no need for a new HTTP request to be made.

### 4.1 `form1.html`

- `form1.html` demonstrates a standard HTML login form that has no client-side validation:

  ```
  <!--

  form1.html
  ```

```
A form without client-side validation.

Computer Science 50
David J. Malan

-->

<!DOCTYPE html>

<html>
  <head>
    <title>form1</title>
  </head>
  <body>
    <form action="dump.php" method="get">
      Email: <input name="email" type="text">
      <br>
      Password: <input name="password1" type="password">
      <br>
      Password (again): <input name="password2" type="password">
      <br>
      I agree to the terms and conditions: <input name="agreement" type="checkbox">
      <br><br>
      <input type="submit" value="Submit">
    </form>
  </body>
</html>
```

That is, a user can input almost anything as his e-mail and password and the form will submit to the back end. Of course, the back end might have some validation built in so that if a malformed e-mail address is passed, the form submission will fail and the user will be bounced back to the input page. It would be nice, however, if instead of going through the trouble of clicking Submit and waiting for the HTTP response indicating a submission error, the user could be warned that his inputs are wrong. This is where JavaScript comes in.

### 4.2 dump.php

- Incidentally, `dump.php`, the back end for this form, does nothing but spit out the contents of the `$_GET` variable so that we can do some debugging:

```
<!DOCTYPE html>
<html>
  <head>
    <title>dump</title>
```

```
  </head>
  <body>
    <pre><? print_r($_GET); ?></pre>
  </body>
</html>
```

The `pre` HTML tag tells the browser to print everything inside it as is, i.e. without formatting. The `print_r` function stands for "print recursive" and will run through hierarchical objects like arrays and dump all their contents.

### 4.3 `form2.html`

- `form2.html` introduces the client-side validation that we just discussed:

```
<!--

form2.html

A form with client-side validation.

Computer Science 50
David J. Malan

-->

<!DOCTYPE html>

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

        function validate()
        {
            if (document.forms.registration.email.value == "")
            {
                alert("You must provide an email adddress.");
                return false;
            }
            else if (document.forms.registration.password1.value == "")
            {
                alert("You must provide a password.");
                return false;
            }
            else if (document.forms.registration.password1.value !=
```

```
                    document.forms.registration.password2.value)
            {
                alert("You must provide the same password twice.");
                return false;
            }
            else if (!document.forms.registration.agreement.checked)
            {
                alert("You must agree to our terms and conditions.");
                return false;
            }
            return true;
        }

    // ]]>
    </script>
    <title>form2</title>
  </head>
  <body>
    <form action="dump.php" method="get" name="registration"
          onsubmit="return validate();">
      Email: <input name="email" type="text">
      <br>
      Password: <input name="password1" type="password">
      <br>
      Password (again): <input name="password2" type="password">
      <br>
      I agree to the terms and conditions:
      <input name="agreement" type="checkbox">
      <br><br>
      <input type="submit" value="Submit">
    </form>
  </body>
</html>
```

Here we see that a new attribute of the `form` tag is defined: `onsubmit`. As you might guess, it controls what happens when the form is actually submitted. In this case, we're calling a JavaScript function named `validate`. The `onsubmit` attribute, according to the W3C's specification of HTML, can take actual JavaScript code as its value. If this code evaluates to true, then the form will be submitted. Otherwise, the form will not submit. In this case, we're relying on the `validate` function to evaluate to true if the form inputs are valid and to evaluate to false otherwise.

- At the top, in the `head` element, we have our `script` tag.[2] Right after

[2]Incidentally, the `script` tag can be placed elsewhere on the page other than in the `head` element. There are good reasons one might do so and we'll see examples of it later in the

the open tag, we have the following sequence of characters:

```
// <![CDATA[
```

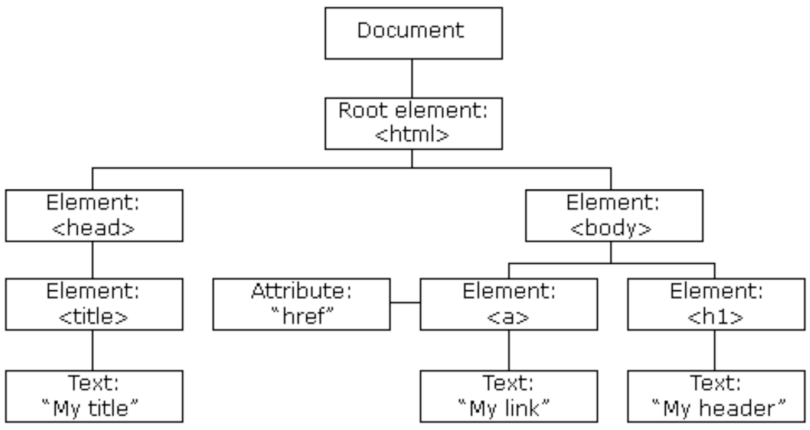
The `<![CDATA[` tells the browser is that the lines which follow are *character data* and shouldn't be parsed as HTML. In this way, characters like `<` which are used in the JavaScript won't prevent our webpage from validating. The double slash in front tells the JavaScript interpreter not to interpret that line as JavaScript.

- The first if condition in `validate` checks for a blank e-mail field. We do this by accessing a global object named `document` that is provided to us natively in JavaScript. `document` is actually a kind of object or struct which contains the entire hierarchy of the page's content.

### 4.4 The DOM

- This hierarchy of a page's content encapsulated in the `document` object is called the DOM, or document object model, and can be visualized like so:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My title</title>
  </head>
  <body>
    <a href="">My link</a>
    <h1>My header</h1>
  </body>
</html>
```




course.

As you can see, because we've been careful to close tags in the order that we open them, certain tags will become the children of others. For example, the `title` element is a child of the `head` element.

### 4.5 `form2.html` (cont'd)

- Within `document`, we access `forms` followed by the `registration` form in particular. Notice that `registration` corresponds to the value we gave to the `name` attribute of our form. Finally we access the `value` attribute of the `email` field. If this value is the empty string, then we call a built-in function called `alert` which pops up a window. After this window pops up, we return false, which is important to ensure that the form doesn't actually submit.
- In the next conditions, we check for a blank password field and for non-matching password fields. Then we access the `checked` property of the checkbox field to make sure that it has been clicked.

### 4.6 `form3.html`

- `form3.html` improves upon `form2.html` by simplifying the validation code:

```
<!--

form3.html

A form with client-side validation demonstrating "this" keyword.

Computer Science 50
David J. Malan

-->

<!DOCTYPE html>

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

        function validate(f)
        {
            if (f.email.value == "")
            {
                alert("You must provide an email adddress.");
                return false;
```

```
            }
            else if (f.password1.value == "")
            {
                alert("You must provide a password.");
                return false;
            }
            else if (f.password1.value != f.password2.value)
            {
                alert("You must provide the same password twice.");
                return false;
            }
            else if (!f.agreement.checked)
            {
                alert("You must agree to our terms and conditions.");
                return false;
            }
            return true;
        }

    // ]]>
    </script>
    <title>form3</title>
  </head>
  <body>
    <form action="dump.php" method="get"
          onsubmit="return validate(this);">
      Email: <input name="email" type="text">
      <br>
      Password: <input name="password1" type="password">
      <br>
      Password (again): <input name="password2" type="password">
      <br>
      I agree to the terms and conditions:
      <input name="agreement" type="checkbox">
      <br><br>
      <input type="submit" value="Submit">
    </form>
  </body>
</html>
```

This incarnation of `validate` takes a single argument instead of none. The argument that is passed to `validate` is the actual form object itself. We achieve this by writing `validate(this)` in the `onsubmit` attribute. The value of `this` will change from context to context. Here, it stands for the form object.

### 4.7 form4.html

- form4.html demonstrates a clever use of the disabled property:

```
<!--

form4.html

A form with client-side validation demonstrating disabled property.

Computer Science 50
David J. Malan

-->

<!DOCTYPE html>

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

        function toggle()
        {
            if (document.forms.registration.button.disabled)
                document.forms.registration.button.disabled = false;
            else
                document.forms.registration.button.disabled = true;
        }

        function validate()
        {
            if (document.forms.registration.email.value == "")
            {
                alert("You must provide an email adddress.");
                return false;
            }
            else if (document.forms.registration.password1.value == "")
            {
                alert("You must provide a password.");
                return false;
            }
            else if (document.forms.registration.password1.value !=
                     document.forms.registration.password2.value)
            {
```

```
                alert("You must provide the same password twice.");
                return false;
            }
            else if (!document.forms.registration.agreement.checked)
            {
                alert("You must agree to our terms and conditions.");
                return false;
            }
            return true;
        }

    // ]]>
    </script>
    <title>form4</title>
  </head>
  <body>
    <form action="dump.php" method="get"
          name="registration" onsubmit="return validate();">
      Email: <input name="email" type="text">
      <br>
      Password: <input name="password1" type="password">
      <br>
      Password (again): <input name="password2" type="password">
      <br>
      I agree to the terms and conditions:
      <input name="agreement" onclick="toggle();" type="checkbox">
      <br><br>
      <input disabled="disabled" name="button" type="submit" value="Submit">
    </form>
  </body>
</html>
```

In this version of our login form, the Submit button isn't clickable until the terms and conditions checkbox has been checked. We achieve this by initially setting the `disabled` attribute of the checkbox to the value `disabled`.[3] Then, we assign a JavaScript function `toggle` to be the listener for the click event (via the `onclick` attribute) on the terms and conditions checkbox. Whenever the checkbox is checked, the `toggle` function will be called. If the checkbox is checked, the `disabled` attribute of the Submit button will be set to false.

- So why bother with server-side validation if client-side validation is so simple and elegant? As it turns out, users can disable JavaScript in almost every major browser with a few clicks of the mouse. If a malicious user were to do this, he could get past all your client-side validation. Thus if

[3] Yes, it's a stupid convention.

you have no server-side validation, he could have a field day with your form. Just as importantly, not every browser fully supports JavaScript—the BlackBerry browser being a good example. When you're developing a website, then, you need to consider what users you might be alienating if you choose to implement functionality which absolutely requires JavaScript.

### 4.8 Ajax

- As we said before, event listeners are at the heart of JavaScript's power. Take Google Maps, for example. The click and drag functionality of the map itself was revolutionary only a few years ago. What makes this functionality possible is listeners for the drag event which then spawn HTTP requests to grab more tiles, the small squares that piece together to make the whole map. If you drag fast enough, you can see that before these tiles are downloaded, the map is at least partly gray. And if you inspect the map in Firebug, you can see that dozens of HTTP requests are made each time you click and drag. Spawning HTTP requests from JavaScript events is the core of Ajax technology.

#### 4.8.1 `ajax1.html`

- In `ajax1.html`, we leverage Ajax in order to create a stock lookup page that never actually refreshes:

```
<!--

ajax1.html

Gets stock quote from quote1.php via Ajax,
displaying result with alert().

Computer Science 50
David J. Malan

-->

<!DOCTYPE html>

<html>
  <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)
            {
                // display response if possible
                if (xhr.status == 200)
                    alert(xhr.responseText);
                else
                    alert("Error with Ajax call!");
            }
        }

    // ]]>
    </script>
    <title>ajax1</title>
  </head>
  <body>
    <form onsubmit="quote(); return false;">
      Symbol: <input id="symbol" type="text">
      <br><br>
      <input type="submit" value="Get Quote">
    </form>
  </body>
</html>
```

Down at the bottom, we see that there's very little HTML that's needed to implement the actual form. One thing to notice is that we've given the symbol input an `id` attribute to uniquely identify it. In order for our page to validate, we need to make sure this attribute actually exists, but we don't have to put a real filename there. Clearly if this attribute is blank, the form isn't actually going to submit anywhere. Instead, we have a JavaScript function called `quote`, specified in the `onsubmit` attribute, which is going to look up the stock price. After this function executes, we're going to return false so that the form doesn't actually submit.

- Ajax is a technology which allows browsers to make additional requests to the server after the web page has already loaded. Unfortunately, browsers never agreed upon how to implement Ajax, so we have to use some muddy syntax in order to ensure cross-browser compatibility. First, we're initializing a global variable named `xhr` by trying to create a new XMLHttpRequest object, the object which implements the magic of Ajax. Unfortunately, this won't work in Internet Explorer because Microsoft decided that their particular flavor of this object would be called an ActiveXObject. For that reason, we use the try-catch syntax, which attempts to execute the try block and only executes the catch block if the try block fails for some reason.
- After we've initialized `xhr`, we check for null just in case the user is run-

ning a browser that doesn't support Ajax. Next we're dynamically creating a URL[4] which we're going to request from the server. In a `GET` variable named `symbol`, we're appending the value the user has entered into the text box. We are accessing this value by invoking a method called `getElementById`, which, as you might've guessed, searches for an HTML element whose `id` attribute we specify. In this case, we've given the symbol input an `id` of `symbol`, so that's what we're searching for.

- The three lines at the bottom of `quote` are the ones which actually retrieve the stock quote. The last two lines actually open a connection to the server and send the data. If you wanted to use the `POST` method, you would specify `POST` as the first argument to `open` and you would pass the actual data as the argument to `send`, rather than null. By passing true as the third argument to `open`, we specify that the request will be asynchronous; that is, our JavaScript program won't wait for the request to return before it continues executing the rest of our code. In computer-science speak, this is a *non-blocking call*.

- The line of code that accesses the `onreadystatechange` property of `xhr` tells it to call a function named `handler` when it receives a response for its HTTP request. This demonstrates that functions in JavaScript, because they are implemented as pointers, can be passed around just like any other object or variable.

- Within the `handler` function, we are checking two properties of the `xhr` object: `readyState` and `status`. First, we check `readyState` to find if the request has been sent successfully (i.e. 4); second, we check `status`, to see if the server has returned a response of OK (i.e. 200). If both of those checks are passed, then we access the `responseText` of the object and display it via an alert window.

- So let's actually see what this URL will return if we access it directly. If we navigate to `quote1.php?symbol=GOOG`, we get back nothing but a stock quote—no HTML markup, even. It is our `handler` function which will be manipulating this directly.

#### 4.8.2 `ajax3.html`

- Slightly more sophisticated than an alert window would be to embed the response in the actual HTML of the webpage, as we do in `ajax3.html`:

```
<!--

ajax3.html

Gets stock quote (plus day's low and high) from quote2.php via Ajax,
```

[4]The URL is actually relative, so the prefix `https://cloud.cs50.net/` will be assumed.

```
embedding result in page itself after indicating progress with an
animated GIF.

Computer Science 50
David J. Malan

-->

<!DOCTYPE html>

<html>
  <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 = "quote2.php?symbol=" +
                      document.getElementById("symbol").value;
```

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

            // get quote
            xhr.onreadystatechange = handler;
            xhr.open("GET", url, true);
            xhr.send(null);
        }


        /*
         * void
         * handler()
         *
         * Handles the Ajax response.
         */
        function handler()
        {
            // only handle requests in "loaded" state
            if (xhr.readyState == 4)
            {
                // hide progress
                document.getElementById("progress").style.display = "none";

                // embed response in page if possible
                if (xhr.status == 200)
                    document.getElementById("quote").innerHTML =
                       xhr.responseText;
                else
                    alert("Error with Ajax call!");
            }
        }

  // ]]>
  </script>
  <title>ajax3</title>
</head>
<body>
  <form onsubmit="quote(); return false;">
    Symbol: <input id="symbol" type="text">
    <br><br>
    <div id="progress" style="display: none">
      <img alt="Please Wait" src="pleasewait.gif">
      <br><br>
    </div>
```

```
      <div id="quote"></div>
      <br><br>
      <input type="submit" value="Get Quote">
    </form>
  </body>
</html>
```

In the actual HTML source, we see that the the progress bar GIF is actually already embedded. But because the `div` which contains it has its CSS property `display` set to `none`, it won't actually be visible when the page is first loaded. If we examine the JavaScript, we see that it's almost identical to `ajax1.html`, except for a few lines, one in the `quote` function which sets the `display` property to `block`, and one in the `handler` function which sets this `display` property back to `none`.

- Realize that the GIF animation is not beginning when we click Get Quote. The animation is actually built into the GIF, which has been in the background the whole time. We simply make it visible when we click Get Quote.