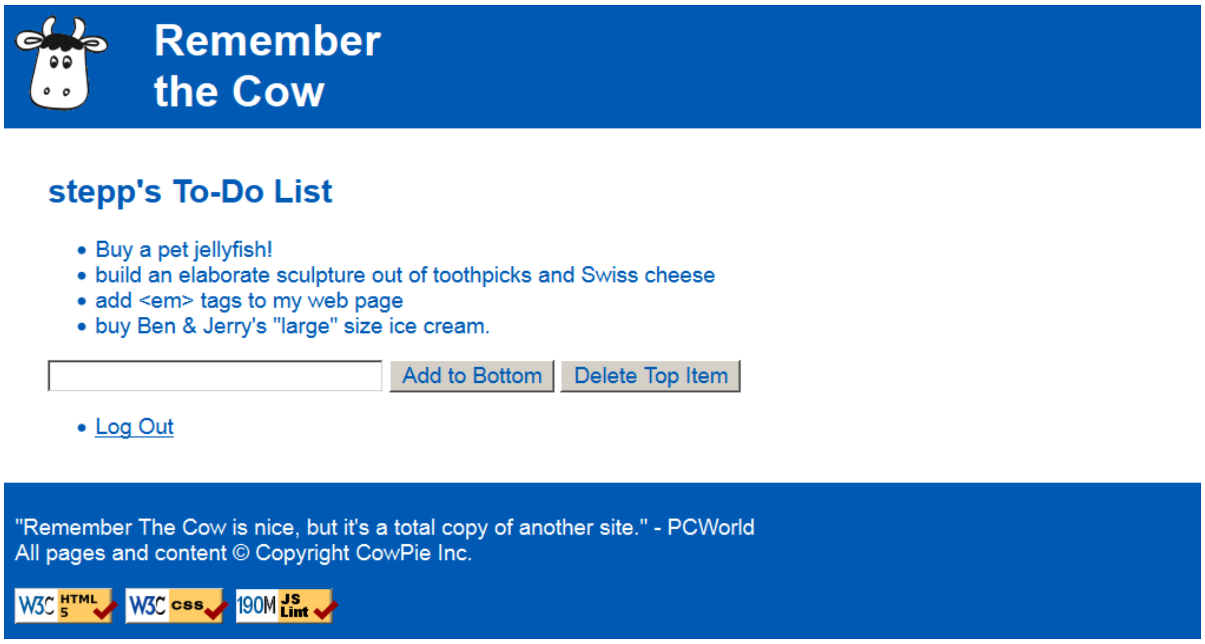
University of Washington, CSE 190 M
Homework Assignment 9: Remember the Cow (To-Do List)

In this assignment you will write a small yet complete "Web 2.0" application that includes user login sessions, a PHP web service with Ajax, and graphical effects from the Scriptaculous JavaScript library.

You will create a web page for a fictional online to-do list site called "Remember the Cow", which is a parody of the real to-do list web site "Remember the Milk". Your site requires the user to log in first. After logging in, the user can manipulate his/her to-do list. The user can add an item to the end of the list, delete the first item from the list, or drag the items into a different order. Any of these changes made to the list will be saved to the web server using Ajax, so that if the user leaves the page and returns later, the current state of the list will be remembered.


Remember the Cow

stepp's To-Do List

- Buy a pet jellyfish!
- build an elaborate sculpture out of toothpicks and Swiss cheese
- add <em> tags to my web page
- buy Ben & Jerry's "large" size ice cream.

"Remember The Cow is nice, but it's a total copy of another site." - PCWorld
All pages and content © Copyright CowPie Inc.

You will write and turn in the following files *(see the next page for more details about each file)*:

- **top.html** and **bottom.html**, files holding HTML content shared by several of your web pages
- **cow.css**, the style sheet describing the appearance of all of your web pages
- **index.php**, the front web page describing the site and containing a form for the user to log in
- **login.php**, the target where **index.php** submits its login form data to log the user in
- **logout.php**, logs the user out and returns them to the front page **index.php**
- **todolist.php**, the PHP web service for remembering the state of the to-do list
- **todolist.js**, the JavaScript code that manages the state of the to-do list
- **webservice.php**, a PHP web service for saving/loading the current state of the user's to-do list
- **shared.php**, a PHP file containing any shared PHP code or functions used by multiple other files

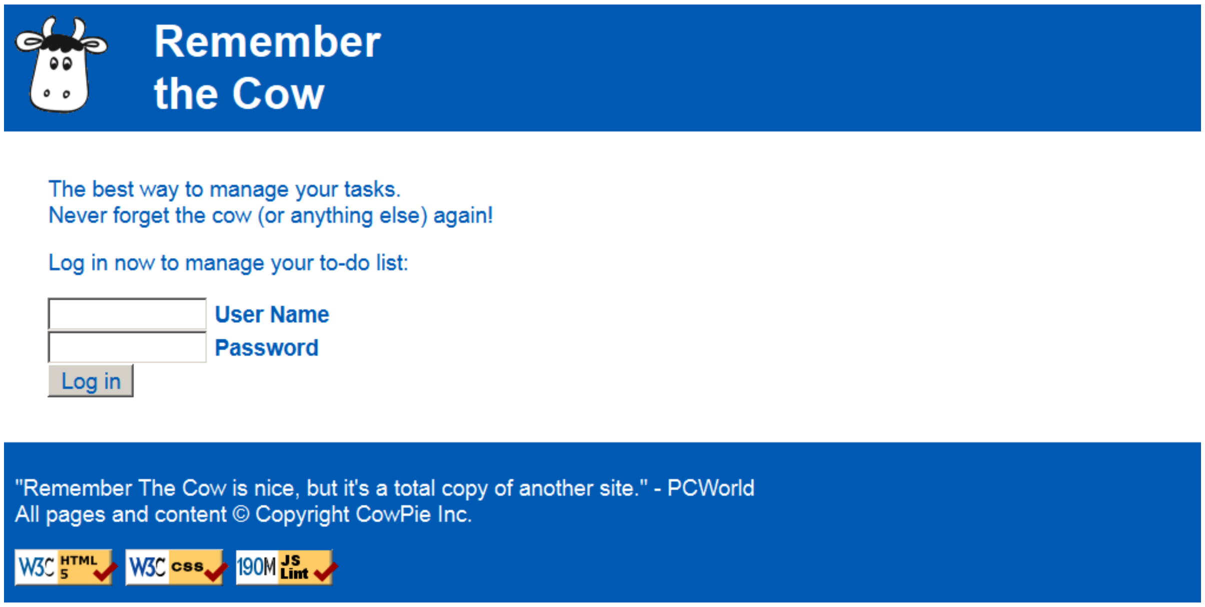
On the course web site we have provided a **skeleton of the HTML output** from **index.php** and **todolist.php**, to help you get started. The HTML given should not necessarily stay in those files exactly as written (e.g., it might go into one of the "common" files), but it is an indication of what the output should be. You don't have to produce exactly the same HTML output as in these skeletons, but they can serve as a starting point.

This assignment uses Ajax to communicate with a PHP web service. In order for Ajax and PHP to work correctly, you must **upload your files to the Webster server** and test them there.

Pages' Appearance:

The site consists of two pages that actually output HTML content for the user to see: **index.php** and **todolist.php**. Both pages share a common appearance and theme. Your pages' appearance must match the following specification; but several aspects of the page appearance are intentionally not mentioned in this spec and are left up to you. The intent is for you to **be creative and make your own unique page appearance**, so long as you satisfy the requirements in this spec. If you want to exactly match our expected output images, that's fine, but we'd rather see you make something unique and creative. *(We require you to match our top/bottom blue bars and some page behavior, but the exact appearance of the content in the main section between the blue bars is basically up to you. Through the rest of this spec, we will mention aspects of our own page's appearance in italic for reference, but you are not required to match those italic aspects.)*

Here is the **index.php** page's initial appearance:



Remember
the Cow

The best way to manage your tasks.
Never forget the cow (or anything else) again!

Log in now to manage your to-do list:

"Remember The Cow is nice, but it's a total copy of another site." - PCWorld
All pages and content © Copyright CowPie Inc.

The **overall page body** has no margin or padding, so that the page content is able to stretch to the very edges of the browser window. Text in the page body uses font Arial, falling back to Helvetica, then to the default sans-serif font available on the system. The default body font size is 14pt and all other text (headings, etc.) are relative to this. All form controls (text boxes, buttons, etc.) on both pages should also use these fonts and sizes.

A **blue bar** appears along the top and bottom of each page; the bars use a blue background color of #005AB4 and white text. Both bars have 0.5em of space between the edge of the content and the edge of the blue background.

The **top bar** contains the site's "cow" logo image, and 1em to the right of the image, the site's name, "Remember the Cow", as a level-1 heading split across two lines. The heading has no vertical margin. The logo image is found at:

- **http://www.cs.washington.edu/education/courses/cse190m/12sp/homework/9/logo.gif**

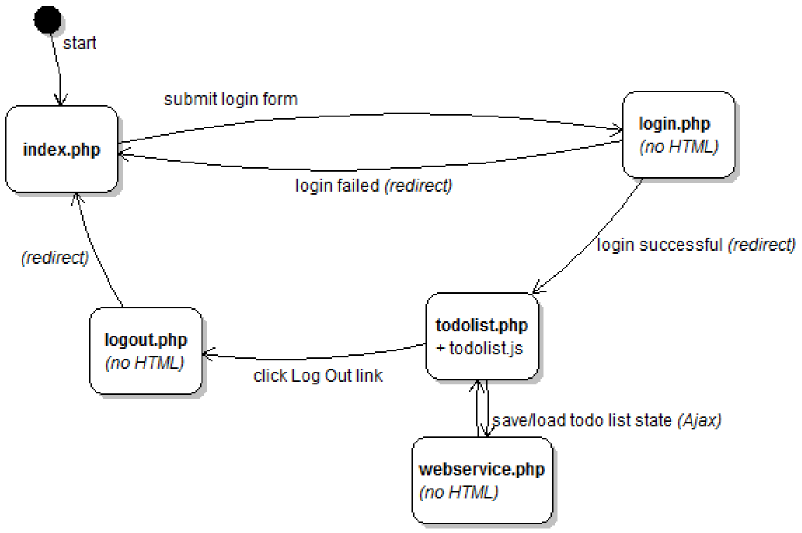
The **bottom blue bar** on each page displays a quote about the web site and a copyright notice, along with the standard three W3C and JSLint images. The URLs of these images and their link targets are the same as in the past:

Image:	**Links to:**
https://webster.cs.washington.edu/w3c-html.png	https://webster.cs.washington.edu/validate-html.php
https://webster.cs.washington.edu/w3c-css.png	https://webster.cs.washington.edu/validate-css.php
https://webster.cs.washington.edu/w3c-js.png	https://webster.cs.washington.edu/jslint/?referer

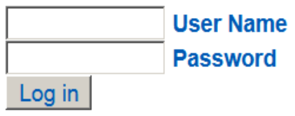
Between the top/bottom blue bars is a **main section of content** on both pages. This main section's appearance is up to you. Be creative! *(Our version has a white background and blue text in the color of #005AB4. The form controls also display their text in this color. Our main area has 2em of padding on all four sides around the edge of its content.)*

Page Relationships and Behavior:

Various actions cause the user to transition from one page to another, as summarized by the following diagram and described in more detail in the following sections.



index.php: This is the initial page that the user visits upon entering the site. The page displays a login form with text boxes for a user name and password *(ours are each 12 characters in size)*. The user types a name and password and presses a submit button to log in, which causes the form to submit to **login.php** as a POST request.

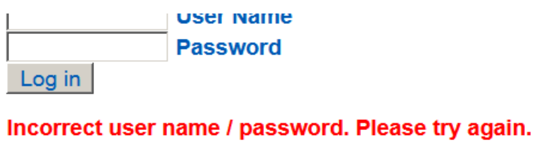


login.php: This file never directly produces any HTML output. Instead, it accepts the user name and password parameters from **index.php** and checks if they are correct, and based on this, redirects the user to another page. If either of the required parameters are not passed, issue an **HTTP 400 error** (Invalid Request) and exit.

In a real site, we would maintain a list of many user names and passwords and allow users to sign up for new accounts; but for this assignment, we will have a single known correct user name and password. Use **your UW NetID** as the expected correct user name, and use **"12345"** as the correct password.

If they are correct, a new user **login session** is started so that the site will remember the user's information, and the user is immediately redirected to **todolist.php**. Store session data in PHP's `$_SESSION` global array. *(See textbook Chapter 14 for help on implementing user login sessions, particularly the case study.)*

If the user name or **password is incorrect**, the user is instead redirected back to **index.php**, which should now display some kind of formatted error message paragraph underneath the login form, indicating that the login failed: *(Ours displays a red bold message saying, "Incorrect user name / password. Please try again.")*



Incorrect user name / password. Please try again.

You can use the PHP `header` function discussed in class to redirect from one page to another, such as:

```
header("Location: foo.php");
```

(Hint: You could implement such an error message by redirecting the user back to ***index.php*** *with a particular GET query parameter indicating a failed login, and checking for this parameter in* ***index.php****.)*

(From the user's perspective it may look as though ***index.php*** *submits directly to* ***todolist.php*** *on success or submits back to itself on failure, since the redirect will happen quickly.)*

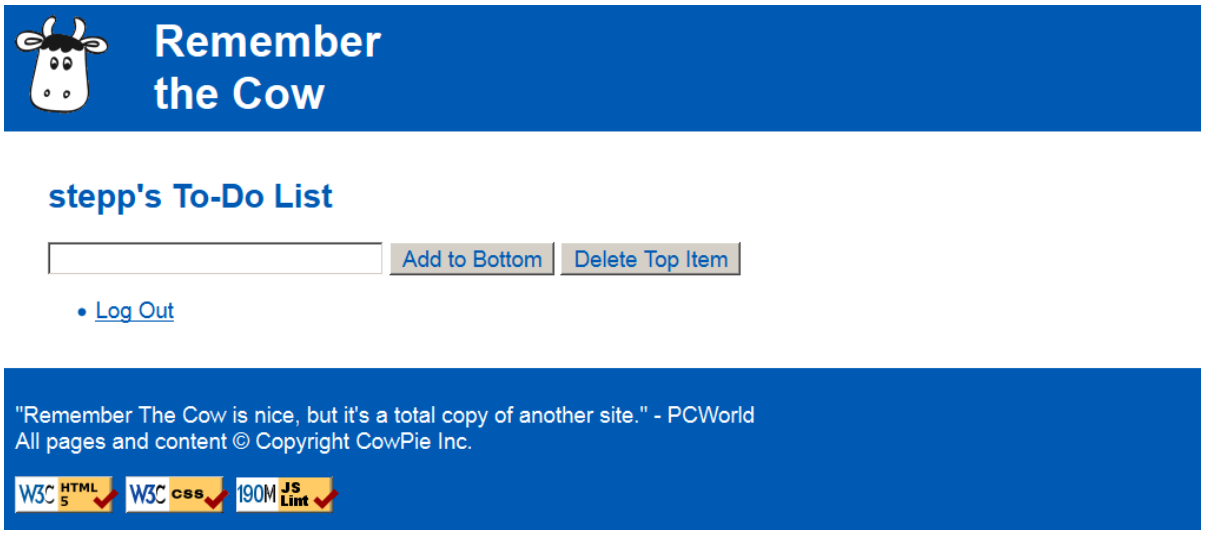
Page Relationships and Behavior (continued):

todolist.php: This is the page to which the user is sent after logging in. If the user tries to visit **todolist.php** without being logged in, they are immediately redirected to **index.php**. (Similarly, if the user tries to visit **index.php** when they are already logged in, they should be immediately redirected to **todolist.php**.)

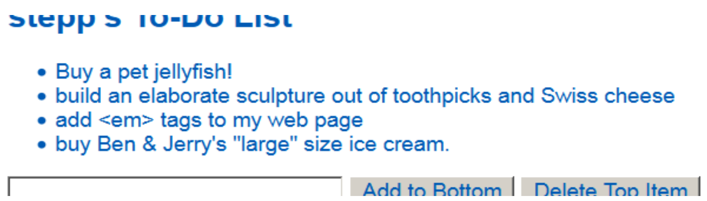
The page displays the user's current to-do list and allows the user to manipulate that list in various ways. The page should have a descriptive heading *(ours says, "<username>'s To-Do List")*. After the heading is an (initially empty) bulleted list of to-do items, followed by a text box to allow the user to type a new to-do item to add to the bottom of the list *(ours is 30 characters in size)*. The page should also have options for the user to add an item to the list and delete an item(s) from the list *(our page has two buttons right of the text box labeled "Add to Bottom" and "Delete Top Item")*.

The to-do list is initially empty, if the user has never visited the site before. If the user has visited the site previously and added items to his/her to-do list, those items are fetched from the server using Ajax and shown (described later).

Your **Add** button adds a new item to the end of the to-do list using the text currently in the text box (if this text is non-empty). The **Delete** button erases an item from the current to-do list; if the to-do list is empty, clicking the Delete button does nothing. *(Our Delete button deletes the first (top) item from the list. This is the minimum delete functionality you must provide. If you want, you can make it possible to delete other items, but this is not required.)*


Remember
the Cow
stepp's To-Do List
Add to Bottom
Delete Top Item
Log Out
"Remember The Cow is nice, but it's a total copy of another site." - PCWorld
All pages and content © Copyright CowPie Inc.

For example, after adding several items the to-do list might look like the following. Notice that to-do items can contain characters like `<` or `&`; your JS code should **HTML-encode them** using the string's `escapeHTML` method.


stepp's To-Do List
Buy a pet jellyfish!
build an elaborate sculpture out of toothpicks and Swiss cheese
add <em> tags to my web page
buy Ben & Jerry's "large" size ice cream.
Add to Bottom
Delete Top Item

In addition to being able to add/delete items, the user should also be able to drag an item up and down in the list to **reorder** it to a new position relative to the other items. Use the Scriptaculous `Sortable` feature to achieve this.

The to-do list page must also contain a "log out" link. *(Ours appears below the to-do list as another one-item bulleted list with a "Log Out" link.)* When the user clicks this link, it goes to **logout.php**. As with the login file, **logout.php** is not a file that produces any HTML output. Instead, it immediately ends the user's login session and redirects to **index.php**, which will show the login form.

webservice.php: Each time the user makes a change to the to-do list, immediately save the to-do list's new state on the server by sending it in an **Ajax POST request** to a file you will write named **webservice.php**. You will send the to-do list as a POST query parameter named `todolist` whose value is a JSON object representing the list. (If a POST is made to your web service without this required parameters, issue an **HTTP 400 error** and exit.)

The JSON object to submit should contain a single field named `items` which is an array of strings, where each string is one to-do item. You'll have to build this object yourself; construct an empty JavaScript `{}` object and put an array inside it that you have built from of the contents of the bulleted list DOM elements on the page. For example:

```
{
    "items": [
        "Buy a pet jellyfish!",
        "build an elaborate sculpture out of toothpicks and Swiss cheese",
        "add &lt;em&gt; tags to my web page",
        "buy Ben &amp; Jerry's \"large\" size ice cream."
    ]
}
```

Your JSON data does not need to match the exact whitespace shown above, but the object's fields should match. Recall that you can use the `JSON.stringify` function in JavaScript to encode an object into JSON format. The web service should save the JSON data it receives into a file called **list.json**. If no such file exists, your web service should create the file; if the file already exists, its contents should be overwritten.

Similarly, when the **todolist.php** page first loads, it should show any prior contents of the to-do list by sending an Ajax GET request to **webservice.php**. When the web service receives a GET request, it should retrieve the contents of **list.json** (if any) and output them using the content type of **application/json**. If the file does not exist when the web service is contacted (for example, if the user is visiting the site for the first time), you should produce an empty output and not produce any errors. On a POST request, your service doesn't need to produce any output.

Since the items are retrieved from the server, the page will not show your to-do list items until the server has been contacted. Once the items arrive, they should be added to your page using the DOM. If your JS code receives an error from any Ajax request (`onFailure`), show a brief error message in the page including the HTTP error code.

While writing the file **list.json** you may see an error, *"Permission denied"*. If so, use your SFTP software (e.g. FileZilla) to give "write" permissions on the **list.json** file, and "write/execute" permissions on your overall **hw9/** folder.

We recommend that you debug your queries in Firebug or Chrome. You can see each Ajax query request in the Network or Net tab. Expand it with the + sign to view the query parameters passed and the web service's response.

Visual Effects with Scriptaculous:

For full credit, your page should apply visual effects to actions done on the **todolist.php** page. Specifically, apply a **Scriptaculous appearing effect** to items when they first arrive from the server, such as making them "fade in" or "shake". You can make items initially invisible by calling the `hide` method on their DOM objects or by setting `display` to `none`. An invisible element can be shown using Scriptaculous effects such as `appear` or `grow`. To put an effect on an element being deleted, consult textbook Ch. 10 or the Scriptaculous slides on the `afterFinish` event.

Any change the user makes to the to-do list should be accompanied by a Scriptaculous effect of your choice, along with any other visual cues you want. For example, an added item could highlight or fade into view.

Use Scriptaculous to make the list items reorderable. Give the list an `id` and make it into a `Sortable` list. Note that `id`s that must be present on elements. Also note that the sortability of a list breaks if you add or remove elements after you've made it sortable. To fix such a case, re-specify the list to be `Sortable` after each modification.

When any of the three preceding actions (add, delete, reorder) has been made on the page, the page should immediately send an Ajax request to your PHP web service to inform the server of the change. If you've done this properly, at any point the user should be able to refresh the browser and still see the to-do list in its current state.

Any updates to the to-do list should appear instantly on the page. The instant that the user adds, deletes, or reorders items, the page should update to reflect this action. In the background, the page may be contacting the server to inform it of the change, but the page UI should not be out of date or locked up during this. Specifically, you should not need to do a GET on the web service after every POST just to be able to see the to-do list's new state. You do not need to worry about multiple rapid updates overloading the server or arriving to the server out of order.

Development Strategy:

There is a lot of code to be written, and none of it is being provided to you. It can be challenging to know where to start or how to make the various pieces fit together. We suggest roughly the following development strategy:

- Write the **index.php** page and the **cow.css** for its basic appearance, based on our provided skeleton.
- Write the **todolist.php** page's initial appearance based on our skeleton, ignoring the issue of logging in. Think about factoring out common HTML and/or PHP code shared between multiple pages.
- Write **todolist.js** to enable your to-do list to add and delete from the list, without actually using any Ajax to contact the server. The list will be forgotten when the user refreshes or leaves the page.
- Make the list rearrangeable using Scriptaculous' `Sortable` functionality, again without contacting the server.
- Add Scriptaculous effects to your page.
- Write your **webservice.php** and make the page send/receive the to-do list on each change and on initially loading. Remember to debug your query requests and responses in Firebug / Chrome.
- Write the log in/out logic, sessions, and redirects to manage the overall flow between pages. (See Ch. 14.)

For reference, our solution has roughly 40 (35) lines in ***.html**, 35 (20) lines in ***.css**, 140 (100) lines in ***.php**, and 80 (50) lines in ***.js** (total / substantive). But you do not need to match these totals exactly.

Implementation and Grading:

For full credit, your page must pass the W3C HTML and CSS **validators**. Express your CSS concisely and without unnecessary or **redundant** styles. If a color or font or other important property is used in multiple places, declare it in a common shared rule if possible so that it could be changed by modifying the CSS file in just one place.

Make effort to avoid redundancy in your HTML, CSS, PHP, and JavaScript. For example, common HTML that appears on all pages should go in **top**/**bottom.html** and be included by the other pages. Any shared PHP code you want to execute from more than one place should be placed into **shared.php**, probably into functions.

For full credit, your JavaScript code should pass the provided **JSLint** tool with no errors reported and should be run in strict mode. Use the HTML DOM appropriately. Follow proper style in your Ajax requests, including obeying the proper query request type (GET vs. POST). You should also follow reasonable style guidelines similar to those of a CSE 14x programming assignment. In particular, minimize global variables, avoid **redundant code**, and use parameters and return values properly. You should not use any other libraries besides Prototype and Scriptaculous.

You are not allowed to have any **global variables** on this assignment; values should be declared in the most local scope possible. If a particular constant value is used frequently throughout your code, you may declare it as a global "constant" variable named `IN_UPPER_CASE` and used throughout your code.

You should separate content (HTML), presentation (CSS), and behavior (JS). Your JS code should generally not set styles of elements manually if the same effect could be achieved by setting classes in JS to target your CSS file.

For full credit, you must write your code using **unobtrusive JavaScript**, so that no JavaScript code, `onclick` handlers, etc. are embedded into the HTML code.

All of the files you submit should have adequate **commenting**. The top of every file should have a descriptive comment header describing yourself, the assignment, and that file's purpose. PHP and JavaScript files should also have descriptive comment headers on each function and on each complex section of code.

Your **PHP/JS code** should generate no error or warning messages when run using reasonable sets of parameters.

Format your code similarly to the examples from class. Properly use whitespace and indentation. Use good variable and method names. Avoid lines of PHP/JS code more than 100 characters wide. In your HTML, do not place more than one block element on the same line or begin any block element past the 100th character on a line.

Do not place a solution to this assignment on a public web site. Upload your files to the **Webster** server at:

https://webster.cs.washington.edu/*your_uwnetid*/hw9/