Computer Science 75
Spring 2009
Scribe Notes

Lecture 3: February 23, 2009
Andrew Sellergren

Contents

1 Announcements (0:00–5:00)

- Project 1 was released today. (And Project 0 was due today, in case you didn't know. . .) Unfortunately, Three Aces Pizza actually closed for good. Sigh. Thus, the problem set has been reorganized a bit, but the theme is still the same.
- Alex Chang has decided to focus on his own studies, so Kent Rakip, a sophomore CS concentrator, will be replacing him as a TF for the semester. Welcome, Kent!
- Tonight's section will be led by David (ooh, special treat) and will focus on getting your feet wet for Problem Set 1.
- Know that Wednesday's section will be recorded and posted online. You can also view it live online (the link is now on the website unlike last week!).
- 2 handouts this week!
- Check out the Bulletin Board post here which solicits those interested in getting a study group together. We definitely encourage this, so hop on board!
- Avail yourself of the scribe notes[1] so that you don't have to suffer the tedium of scribbling down every little tidbit during lecture.[2]

2 Cascading Style Sheets (5:00–12:00)

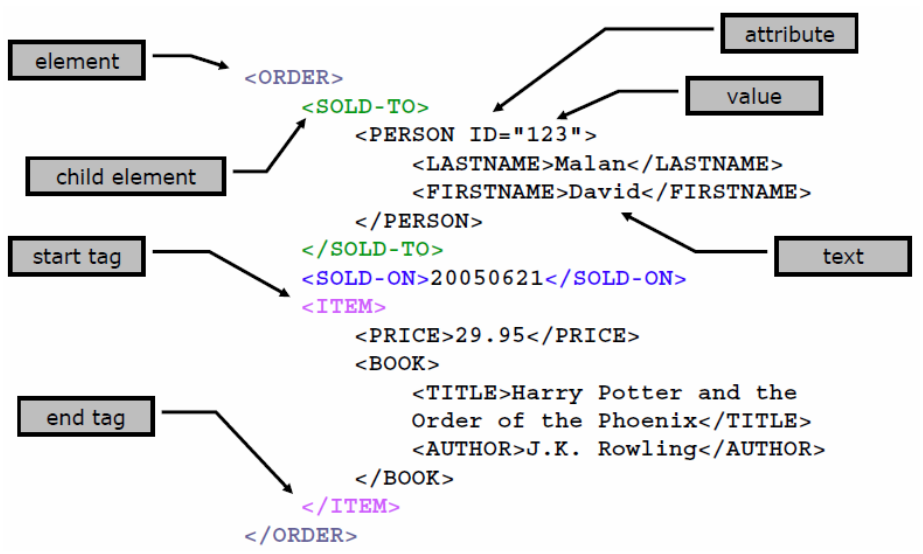
- Not something we usually emphasize in the course, but we required you to include some from the YUI library merely to illustrate the importance of cross-browser compliances (which the YUI library helps achieve).
- As a good example, David whipped up some simple HTML code which was meant to create a `div` with a green background. Notice that in Firefox, Safari, and Chrome, there is a relatively small white border around the `div` even though no padding was explicitly set. In IE, this border is even larger. Check it out here. Once we include the proper YUI CSS, however, this border disappears in all browsers. They achieved this by specifying a `margin` of 0 on the `body` element. Pretty handy!
- YUI is one of many libraries (e.g. MooTools, jQuery) which prevent you from having to reinvent the wheel. When you want tabs, for example, you can simply instantiate an object of that type. No need to struggle with the JavaScript yourself!

[1]Wow, so meta.
[2]Leave that tedium to me.

- For the most part, the Reset and the Font library from YUI are really about leveling the playing field, allowing you to start from a clean state. From Project 1 forward, you're welcome to not use them at all if you so desire!
- It's helpful that Yahoo and Google host libraries like this so that you don't have to pay for bandwidth (and if other websites are using them too, then they might be cached on your user's browsers).
- What exactly does the Reset library do? If you open it up for yourself, you'll notice that the first thing it does is to set the margin for the `html` and `body` and `div` elements, etc, to 0. Can you simply copy and paste this into your own CSS file? Certainly. The YUI license allows this—be careful to read the fine print for other libraries before you do this.
- Note that the course does not expect your CSS to validate, only your XHTML.
- Do we discourage the use of tables in your HTML? Nope! Feel free to use them if they get the job done. There *are* issues you need to be aware of, however, when you do this. For example, interfering with screen readers for your users who are blind.

3 XML (12:00–65:00)

- We being our exploration of XML with a look at the course's own website, which itself uses XML files as a way of separating data from aesthetics.
- Basically, what we found is that writing a few lines of code to parse these XML files was easier than manually editing the website's XHTML every time we needed to update it.
- Notice that we have XML files for sections, lectures, resources, software. These are all stored in the `/etc/xml/` directory, although that's not really important.
- In the `lectures.xml` file, we see that there is a `<lectures>` tag at the top, within which there are individual `<lecture>` tags. Each of these lecture tags has several elements of its own, namely `<title>`, `<dates>`, and `<resources>`. Basically, long story short, this is a way of representing the data in an organized, human-readable, and machine-readable way.
- Think of XML as being much like XHTML, except there's no fixed set of tags which you can use. The number and types of tags are up to you!
- There are languages like XML Schema and DTD which allow you to define what your XML has to look like in order to establish consistency. But we're not going to worry about that in this course.



```
<ORDER>
    <SOLD-TO>
        <PERSON ID="123">
            <LASTNAME>Malan</LASTNAME>
            <FIRSTNAME>David</FIRSTNAME>
        </PERSON>
    </SOLD-TO>
    <SOLD-ON>20050621</SOLD-ON>
    <ITEM>
        <PRICE>29.95</PRICE>
        <BOOK>
            <TITLE>Harry Potter and the
            Order of the Phoenix</TITLE>
            <AUTHOR>J.K. Rowling</AUTHOR>
        </BOOK>
    </ITEM>
</ORDER>
```

- This snippet of XML seems to represent some kind of purchase order, perhaps from Amazon. Amazon, as you've probably noticed, sells a lot of merchandise on behalf of third-party vendors. How might it transmit data to them about purchases? Well, one way it might do so is using XML, which is a great language for standardizing data since it's so easy to parse.

- A few pieces of jargon: open tags mark the beginning of *elements* and close tags mark the end of the same. Along with *attributes*, this should be familiar to you from XHTML. So what about *text elements*, the stuff that comes between the tags? This is the actual data, not the metadata. Why do you think it's useful to group the two together? Well, for the simple reason that the presentation is self-evident. You don't need to be told in advance what the file's going to look like or how the data is going to be organized in order to properly parse it. All you need to do is look for the open and close tags for each bit of data. What's more, this format is extensible. For example, we could add INITIAL in addition to FIRSTNAME and LASTNAME to describe a person.

- Let's take a look at a slightly more complicated XML file:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- This is an XML document that describes students -->
<students>
    <student id="0001">
        <name>Jim Bob</name>
        <status>graduate</status>
```

```
            <dorm/>
            <major>Computer Science &amp; Music</major>
            <description>
                    <![CDATA[ <h1>Jim Bob!</h1>
                    Hi my name is jim. I look like
                    <img src="jim.jpg"> ]]>
            </description>
        </student>
        <student id="0002">
        ...
        </student>
    </students>
```

- Notice this first line of code:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  ```

  This is the XML declaration. It simply says "Hi, I'm an XML document!" Note that it needs to be at the very top and the very left of a document in order to be treated properly by an XML parser. It *can*, however, be left off in your own config files, for example, because your own parser will be aware of the format of the file to come.

- The `encoding` attribute simply specifies a standard for representing letters and other special characters with numbers. `UTF-8` is a superset of ASCII.

- What about the `<!--`? It denotes a comment.

- The convention of XML is for everything to be surrounded by the open and close tags of a single root element. In this case, it's the `<students>` tag. Beneath that root element, you can have as many children as you want.

- What about the `<dorm />` tag? This is a so-called empty element, much like the `<br />` tag in XHTML. This is useful in the case that you want to convey that you didn't forget the dorm data, but merely that it was absent or missing in this particular case.

- The `<![CDATA[` and `]]>` tags denote *character data*. This tells the parser to gobble up everything between these tags but *not* to parse it. This is useful so that you can embed XHTML within XML without confusing the parser.

- One little gotcha: you can't begin an element name with a number.

- Question: when should something be a child element and when should it be an attribute? There isn't necessarily a clearcut answer to this question, actually. A few things to consider, though, are: *a*) attributes are often more quickly and easily accessible than child elements *b*) attributes tend to be data that do not need to be extensible and *c*) attributes are short and sweet—not whole sentences, for example.

- Below are several content models for XML you should be aware of:
  - Element Content

    ```
    <student>
        <status>...</status>
    </student>
    ```

  - Parsed Character Data (a.k.a. PCDATA, Test)

    ```
    <name>Jim Bob</name>
    ```

  - Mixed Content

    ```
    <name>Jim <initial>J</initial>Bob</name>
    ```

  - No Content

    ```
    <dorm />
    ```

- In XML, there exists the notion of *entities*, which are ways of escaping certain special characters. For example, you can't simply insert an & into XML, but must instead write `&amp;`. This entity is pre-defined, along with four others. Any other entities must be declare legal, much like macros. Google is your friend when it comes to entities!

- Let's take a stab at creating our own XML file. In the spirit of the Oscars last night, we'll name it `movies.xml`:

  ```
  <movies>
    <movie id="1">
      <name>Slumdog Millionaire</name>
      <genre>Romance</genre>
    </movie>
    <movie id="2">
      <name>Milk</name>
      <genre>Drama</genre>
    </movie>
    <movie id="3">
      <name>Curious Case of Benjamin Button</name>
      <genre>Drama</genre>
    </movie>
  </movies>
  ```

  Is this XML well-formed? You might think it isn't because it doesn't have an XML declaration, but you should know that this is *optional*. As it turns out, yes it's well-formed. That is, it's syntactically valid. XML is case-sensitive, but there's no strict requirement on whether tag names be uppercase or lowercase.

- If you open this file in Firefox, you'll notice that it actually has some style to it. The tags are collapsible. This is purely Firefox's doing. If you right click and View Source, you'll get the plain text.

- So how do we interact with this XML, particularly using PHP? Turns out that recent versions of PHP come with an API called SimpleXML which allows just that.

- To begin using SimpleXML, we're going to instantiate an object of type `SimpleXMLElement` like so:

  ```
  $xml = new SimpleXMLElement(file_get_contents("movies.xml"));
  ```

  What are we passing to this object as an argument? We're actually calling a separate function `file_get_contents` which fetches a file or web page that we specify and returns it as a string.

- Now, say we want to iterate over each of the movies that we wrote into our XML file and print out some data pertaining to them. What construct will we use for iterating over them? The `foreach` loop is especially useful for this:

  ```
  foreach($xml->movie as $movie)
  {

  }
  ```

  Basically, this syntax says, "Walk through each of the child elements of the root element `movie` and assign them to a temporary variable named `$movie`.

- Even as we assign each `movie` child element to a temporary variable, that temporary variable has a structure which we can access. Let's try to access the `id` attribute:

  ```
  foreach($xml->movie as $movie)
  {
      print("<li>" . $movie["id"] . "</li>");
  }
  ```

  As you can see here, SimpleXML allows us to access the attributes of an element simply as indices to an array. As we mentioned before, this instant access might play into your design decisions (i.e. whether to write something as an attribute or a child element).

- Still, accessing child elements is also rather easy:

  ```
  foreach($xml->movie as $movie)
  {
      print("<li>" . $movie->name . "</li>");
  }
  ```

  Because each movie has only one `name` element, we need only write an arrow to it

- Let's examine a more concrete example of the concept of extensibility. What if we wanted to add information to our XML file about the awards that each movie won at the Oscars? We might do that like so:

  ```
  <movies>
    <movie id="1">
      <name>Slumdog Millionaire</name>
      <genre>Romance</genre>
      <awards>
        <award>Best Picture</award>
        <award>Original Score</award>
      </awards>
    </movie>
    <movie id="2">
      <name>Milk</name>
      <genre>Drama</genre>
    </movie>
    <movie id="3">
      <name>Curious Case of Benjamin Button</name>
      <genre>Drama</genre>
    </movie>
  </movies>
  ```

  Now, we must ask ourselves the question: did we break our PHP code by adding to our XML file? No, in fact. This is because we made no prior assumptions about what other information might be added to each element—nor did we need to!

- Let's add a little bit of nesting to the equation:

  ```
  foreach ($xml->movie as $movie)
  {
      print("<li>");
      print($movie->name);
      print("<ol>");
      foreach ($movie->awards->award as $award)
          print("<li>" . $award . "</li>");
  ```

```
    print("</ol>");
  print("</li>");
}
```

Notice that, although it gets a little ugly, we can access the `award` child elements just as we did the `name` child elements even though they're one more level down the hierarchy.

- What happens when we try to execute this code, though? We get a few errors spit out to the browser window. Know that these errors you see are not actually the typical ones generated by PHP, but rather a marked up version of them generated by Xdebug. Xdebug gives us a stack trace as well as an error message.

- So what's the problem? Well, as you might've guessed, we don't have an `award` element for all of the movies, so when we try to access it using the arrow construct for these movies, we're trying to access something that doesn't exist.

- The actual error is "invalid `foreach` argument." How do solve this? We could first check that what we're accessing is actually an array using the following code:

  ```
  if (is_array($movie->awards->award))
  ```

- Another approach we could take is to simply turn off the reporting of warnings. We can do this in two ways:

  - `ini_set("display_errors", false);`
  - `error_reporting(E_ALL ^ E_WARNING ^ E_NOTICE);`

  Both of these methods are changing the configuration of PHP at runtime. The first simply turns all errors off while the second specifies that only errors, not warnings or notices, are displayed. The ^ is the XOR operator, so we're saying "Start with all errors, then take out the warnings and the notices."

- Of course, this isn't really fixing the problem, just hiding it, so it's not in our best interest.

- A more advanced sidenote to the above: `SimpleXMLElement` is not really returning an array, per se, but rather a construct which allows us to use syntactic sugar like the arrow operator.

- What's a better solution to our problem? How about if we use the following condition:

  ```
  if ($movie->awards->award)
  ```

Now we're checking not if the `award` element is an array, but rather if it exists at all. This is probably the best approach we've examined so far.

- The power of combining PHP and XML is in our ability to choose what information we display. If we have an XML file containing all types of movies, but we only want to display information pertaining to the Dramas, we might write the following:

```
foreach ($xml->movie as $movie)
{
    if ($movie->genre == "Drama")
    {
        print("<li>");
        print($movie->name);
        print("</li>");
    }
}
```

You'll be making lots of choices like this as you prepare your online menu for Three Aces in Project 1. Clearly, you won't want to display all the information at once, but rather smaller chunks of it in different sections.

- Let's take a stab at making this page a little more dynamic:

```
foreach ($xml->movie as $movie)
{
    if ($movie->genre == $_GET["genre"])
    {
        print("<li>");
        print($movie->name);
        print("</li>");
    }
}
```

Now we're going to be printing out movies based on user input. At this stage, however, we have no front-end interface. In order to generate content, we must actually add input to the URL (which is how we pass data by the `GET` method, as you'll recall).

- We can implement the user interface as a simple form:

```
    <form action="genres.php" method="get">
        <select name="genre">
        <option value=""></option>
        <?
            $xml = new SimpleXMLElement(file_get_contents("movies.xml"));
```

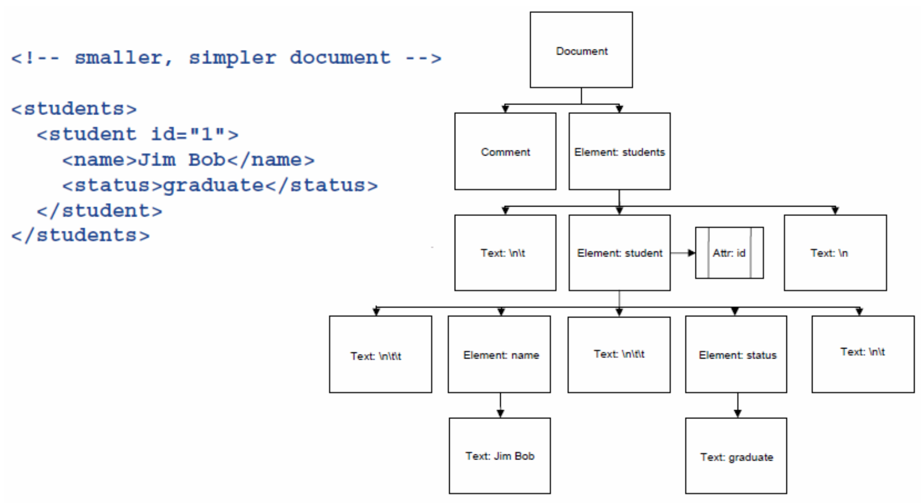
```
                foreach ($xml->movie as $movie)
                {
                    print("<option value='{$movie->genre}'>"
                      . $movie->genre . "</option>");
                }

            ?>
            </select>
            <input type="submit" value="Search" />
        </form>
```

Here, we're simply iterating over each of the movies in our file and spitting out its genre as an option in a dropdown menu. We'll then submit this file using the submit button and it will be sent to the file we coded above, `genres.php`, as specified by the `action` attribute of the form. Note that the curly braces around `$movie->genre` tell the PHP interpreter that the expression needs to be evaluated before it's printed.

- What's the problem with this approach, though? Well, one of the problems is that we're printing out the genre for every movie in our file, so if there are two movies of the same genre, their genre will be listed twice as an option.

4 DOM, XPath, and RSS (65:00–97:00)

- Thankfully, if we want to get at the metadata within an XML file, there exists a query language called XPath. Much as you use directory paths to access the contents of your hard drive (what you might call a database of files), you can use XPath to access the contents of an XML database.
- Up until now, we've neglected to consider what the `SimpleXMLElement` class actually encompasses. Clearly, it is some representation of an XML document using a hierarchical tree structure. How might we describe this tree structure?
- The world of XML effectively revolves around something called the *Document Object Model* (DOM). This is the very tree structure we alluded to earlier:



```
<!-- smaller, simpler document -->

<students>
  <student id="1">
    <name>Jim Bob</name>
    <status>graduate</status>
  </student>
</students>
```

In this diagram, each of the child elements is shown hierarchically below its parent. At the very top is the element from which all others stem—the `document` element. This is to be distinguished from the `root` element, however, which is `students` in this case.

- What's the deal with the other elements besides `students` and `student`? Technically, although whitespace is ignored by us as humans and by the parser as well, we should include it in our DOM to be precise.

- Why is the `id` attribute on the same level as the node it describes? Conceptually, since it's not a child, it doesn't really make sense to show it beneath its element.

- So how can we use XPath to access the genres of movies? We might write something like the following:

  ```
  $genres = $xml->xpath("/movies/movie/genre");

  foreach ($genres as $g)
  {
      print("<li>{$g}</li>");
  }
  ```

  As you might've guessed, the path that we're providing is simply telling the parser that it should only look within the `movies` element and, even deeper than that, within each `movie` element for the `genre` child elements. In general, the more specific you can be when providing a path, the less work the parser is going to have to do to satisfy your query.

- We could change this up and go back to our original example of printing out the movie names:

```
$movies = $xml->xpath("/movies/movie[genre='Drama']");

foreach ($movies as $m)
{
    print("<li>{$m->name}</li>");
}
```

Using this syntax, we can tell the parser to filter the results set so that only Dramas are returned. For each of those Dramas, we print out the name.

- We can also use XPath to access the attributes of elements using the `@` symbol like so:

```
$movies = $xml->xpath("/movies/movie[@id=2]");

foreach ($movies as $m)
{
    print("<li>{$m->name}</li>");
}
```

You should know that the paths we're specifying here are actually shortcuts to a certain extent. To fully comply with the XPath specification, we would need to indicate which *axis* is to be searched on each level. For example, we should really write this:

```
$movies = $xml->xpath("/child::movies/child::movie[attribute::id=2]");
```

This tells the parser explicitly that we want to search on the child axis for the first two levels followed by the attribute axis for the last. There are plenty more axes besides these two, but we'll leave it to you to read up on them in the recommended tutorial for Project 1.

- A few more advanced questions and answers:
  - We won't be using XSLT templates in this course, but it *is* installed on our servers if you'd like to play around with it. Take a look at PHP's documentation to see how much of that library is implemented in PHP.
  - For very large amounts of data, of course, you're better off using databases which support indexing. This will allow for much faster searching than XPath and XML can provide.
  - When the XML is read into PHP, it is slurped up into a large tree structure in memory which is garbage-collected after the file is done executing.

  – There is no need for schemas in XML because it's self-evident from the language itself (as long as it's well-formed). One of the reasons why PHP and XPath don't work well for large datasets is that the entire XML file is parsed as it is read into memory. Searching for elements within the file, then, is done by recursive search. Java, on the other hand, implements certain optimizations like lazy parsing.

- RSS feeds are all the rage these days. Our own bulletin board allows you to subscribe to the RSS feed so you can be notified of new posts. The official RSS specification actually lives here on Harvard's website, but suffice it to say that RSS is really just a glorified XML file. Take a look at the example below:

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
    <channel>
       <title></title>
       <description></description>
       <link></link>
       <item>
          <guid></guid>
          <title></title>
          <link></link>
          <description></description>
          <category></category>
         <pubDate></pubDate>
       </item>
      [...]
    </channel>
</rss>
```

  RSS readers are really just XML parsers when you consider the actual structure of an RSS feed.

- What can we do with this kind of data? Notice on the course's homepage, there is a list of the five most recent bulletin board posts. We can accomplish this with a simple snippet of code that parses the RSS feed:

```
$xml = new SimpleXMLElement(file_get_contents("https://
www.cs75.net/bb/index.php?action=.xml;board=2;type=rss");

foreach ($xml->channel->item as $item)
{
    print("<li><a href='{$item->link}'>{$item->title}</a></li>");
}
```

- In the fall, we leveraged this RSS functionality for the much sillier[3] application of presenting the lolcat of the day.

[3]Read: much awesomer.