web222

WEB222 - Week 6

Suggested Readings

- HTML Tables (MDN)
- Images in HTML (MDN)
- Video and Audio Content (MDN)
- HTML Reference

HTML Element Types: Block vs. Inline

Visual HTML elements are categorized into one of two groups:

1. Block-level elements: create a "block" of content in a page, with an empty line before and after them. Block elements fill the width of their parent element. Block elements can contain other block elements, inline elements, or text.
2. Inline elements: creates "inline" content, which is part of the containing block. Inline elements can contain other inline elements or text.

Consider the following HTML content:

```
<body>
    <p>The <em>cow</em> jumped over the <b>moon</b>.</p>
</body>
```

Here we have a `<p>` paragraph element. Because it is a block-level element, this paragraph will fill its container (in this case the `<body>` element). It will also have empty space added above and below it.

Within this block, we also encounter a number of other inline elements. First, we have simple text. However, we also see the `<em>` and `<b>` elements being used. These will affect their content, but not create a new block; rather, they will continue to flow inline in their container (the `<p>` element).

Empty Elements

Many of the elements we've seen so far begin with an opening tag, and end with a closing tag: `<body></body>`. However, not all elements need to be closed. Some elements have no *content*, and therefore don't need to have a closing tag. We call these empty elements.

An example is the `<br>` line break element. We use a `<br>` when we want to tell the browser to insert a newline (similar to using `\n` in C):

```
<p>Knock, Knock<br>Who's there?</p>
```

Other examples of empty elements include `<hr>` (for a horizontal line), `<meta>` for including metadata in the `<head>`, and a dozen others.

Grouping Elements

Often we need to group elements in our page together. We have a number of pre-defined element container options for how to achieve this, depending on what kind of content we are creating, and where it is in the document.

Using this so-called semantic markup helps the browser and other tools (e.g., accessibility) determine important structural information about the document (see this post for a great discussion):

- `<header>` - introductory material at the top of a
- `<nav>` - content related to navigation (a menu, index, links, etc)
- `<main>` - the main content of the document.
- `<article>` - a self-contained composition, such as a blog post, article, etc.
- `<section>` - a group of related elements in a document representing one section of a whole
- `<footer>` - end material (author, copyright, links)

Sometimes there is no appropriate semantic container element for our content, and we need something more generic. In such cases we have two options:

- `<div>` - a generic block-level container
- `<span>` - a generic inline container

```
<div>
    <p>This is an example of a using a div element. It also includes this <span><em>span</
    <p>Later we'll use a div or span like this to target content in our page with JavaScri
</div>
```

Tables

Sometimes our data is tabular in nature, and we need to present it in a grid. A number of elements are used to create them:

- `<table>` - the root of a table in HTML
- `<caption>` - the optional title (or caption) of the table
- `<thead>` - row(s) at the top of the table (header row or rows)
- `<tbody>` - rows that form the main body of the table (the table's content rows)
- `<tfoot>` - row(s) at the bottom of the table (footer row or rows)

We define rows and columns of data within the above using the following:

- `<tr>` - a single row in a table
- `<td>` - a single cell (row/column intersection) that contains table data
- `<th>` - a header (e.g., a title for a column)

We can use the `rowspan` and `colspan` attributes to extend table elements beyond their usual bounds, for example: have an element span two columns (`colspan="2"`) or have a heading span 3 rows (`rowspan="3")` .

```
<table>
    <caption>Order Information</caption>

    <thead>
        <tr>
            <th>Quantity</th>
            <th>Colour</th>
            <th>Price (CAD)</th>
        </tr>
    </thead>

    <tbody>
        <tr>
            <td>1</td>
            <td>Red</td>
            <td>$5.60</td>
        </tr>
        <tr>
            <td>3</td>
            <td>Blue</td>
            <td>$3.00</td>
        </tr>
        <tr>
            <td>8</td>
            <td>Blue</td>
            <td>$1.50</td>
        </tr>
    </tbody>

    <tfoot>
        <tr>
```

```
            <th colspan="2">Total</th>
            <th>$26.60</th>
        </tr>
    </tfoot>
</table>
```

Multimedia: `<img>` , `<audio>` , `<video>`

HTML5 has built in support for including images, videos, and audio along with text. We specify the media source we want to use, and also how to present it to the user via different elements and attributes

```
<!-- External image URL, use full width of browser window -->
<img src="https://images.unsplash.com/photo-1502720433255-614171a1835e?ixlib=rb-0.3.5&ixi

<!-- Local file cat.jpg, limit to 400 pixels wide -->
<img src="cat.jpg" alt="Picture of a cat" width="400">
```

HTML5 has also recently added the `<picture>` element, to allow for an optimal image type to be chosen from amongst a list of several options.

We can also include sounds, music, or other audio:

```
<!-- No controls, music will just auto-play in the background. Only MP3 source provided --
<audio src="https://ia800607.us.archive.org/15/items/music_for_programming/music_for_progr

<!-- Audio with controls showing, multiple formats available -->
<audio controls>
    <source src="song.mp3" type="audio/mp3">
    <source src="song.ogg" type="audio/ogg">
    <p>Sorry, your browser doesn't support HTML5 audio. Here is a <a href="song.mp3">link
</audio>
```

Including video is very similar to audio:

```
<!-- External Video File, MP4 file format, show controls -->
<video src="http://commondatastorage.googleapis.com/gtv-videos-bucket/sample/BigBuckBunny.

<!-- Local video file in various formats, show with controls -->
<video width="320" height="240" controls>
    <source src="video.mp4" type="video/mp4">
    <source src="video.ogg" type="video/ogg">
    <source src="video.webm" type="video/webm">
    <p>Sorry, your browser doesn't support HTML5 video</p>
</video>
```

NOTE: the `<audio>` and `<video>` elements must use source URLs that point to actual audio or video files and not to a YouTube URL or some other source that is actually an HTML page.

Including Scripts

We've spent a good portion of the course learning about JavaScript. So far, all of our code has been written in a stand-alone form, executed in the Firefox Scratchpad, or by using node.js.

Our ultimate goal is to be able to run our JavaScript programs within web pages and applications. To do that, we need a way to include JavaScript code in an HTML file. Obviously HTML isn't anything like JavaScript, so we can't simply type our code in the middle of an HTML file and expect the browser to understand it.

Instead, we need an HTML element that can be used to contain (or link to) our JavaScript code. HTML provides such an element in the form of the `<script>` element.

We can use `<script>` in one of two ways.

Inline Scripts

First, we can embed our JavaScript program directly within the content area of a `<script>` element:

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Web Page with Script</title>
    </head>

    <body>
        <script>
            console.log('Hello World!');
        </script>
    </body>
</html>
```

Such `<script>` elements can occur anywhere in your HTML, though it is common to put them at the end of the `<body>`. We can also include more than one, and each shares a common global environment, which is useful for combining scripts:

```
<script>
    // Define a global variable `msg` with a String
    var msg = "Hello World!";
```

```
</script>

<script>
    // Access the global variable `msg`, defined in another <script>, but within the same
    console.log(msg);
</script>
```

External Scripts Linked via URL

As our JavaScript programs get larger, embedding them directly within the HTML file via an inline `<script>` starts to become unwieldy. For very small scripts, and debugging or experimentation, inline scripts are fine. However, HTML and JavaScript aren't the same thing, and it's useful to separate them into their own files for a number of reasons.

First, browsers can cache files to improve load times on a web site. If you embed a large JavaScript file in the HTML, it can't be cached.

Second, your HTML becomes harder to read. Instead of looking at semantic content about the structure of your page, now you have script mixed in too. This can make it harder to understand what you're looking at while debugging.

Third, there are lots of tools for HTML, and even more for JavaScript, that only work when fed the proper file type. For example, we often use linters or bundling tools in JavaScript. We can't do that if our JavaScript is combined with HTML markup.

For these and other reasons, it's common to move your JavaScript programs to separate files with a `.js` file extension. We then tell the browser to load and run these files as needed via our `<script>` tag like so:

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Web Page with Script</title>
    </head>

    <body>
        <script src="script.js"></script>
    </body>
</html>
```

In this case, we have no content within our `<script>` element, and instead include a `src="script.js"` attribute. Much like the `<img>` element, a `<script>` can include a `src` URL to load at runtime. The browser will begin by loading your `.html` file, and when it encounters the `<script src="script.js">` element, it will begin to download `script.js` from the web server, and then run the program it contains.

We can combine both of these methods, and include as many scripts as we need. The scripts we include can be:

- embedded inline in the HTML
- a relative URL to the same web server that served the HTML file
- an absolute URL to another web server somewhere else on the web

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Web Page with Scripts</title>
    </head>

    <body>
        <script src="https://scripts.com/some/other/external/script/file.js"></script>
        <script src="local-script.js"></script>
        <script>
            // Use functions and Objects defined in the previous two files
            doSomethingAmazing();
        </script>
    </body>
</html>
```

Validating HTML

It's clear that learning to write proper and correct HTML is going to take practice. There are lots of elements to get used to, and learn to use in conjunction. Also each has various attributes that have to be taken into account.

Browsers are fairly liberal in what they will accept in the way of HTML. Even if an HTML file isn't 100% perfect, a browser can often still render something. That said, it's best if we do our best to provide valid HTML.

In order to make sure that your HTML is valid, you can use an HTML Validator. There are a few available online:

- https://html5.validator.nu/
- https://validator.w3.org/

Both allow you to enter a URL to an existing web page, or enter HTML directly in a text field. They will then attempt to parse your HTML and report back on any errors or warnings, for example: an element missing a closing tag.