# Lecture 2: JavaScript and the HTML DOM

Nick Hynes

January 14, 2014

## Topics



## 1 HTML

<rhetorical-question>So, what is HTML?</rhetorical-question> HTML is a **M**arkup **L**anguage designed for use with **H**yper**T**ext, or text that contains links to other (hyper)text. Practically, HTML is a language used to create web pages. This also means that HTML something of a GUI language for JavaScript!

Markup languages are designed to allow syntactic description of data to change its display or semantics (or both). As a markup language, HTML is very similar to other such languages with which you may be familiar; XML (eXtensible Markup Language), Markdown, and LaTeX are notable examples. It is important to note that, unlike LaTeX and Markdown, HTML, especially HTML5, the latest version, focuses more on the *semantics* of a document than its presentation.

Great! Now that you know what HTML does and why it's used, let's look at how you actually use it!

## 1.1 Syntax

An HTML page is constructed by nesting HTML tags or text inside of other HTML tags. A general HTML tag takes the following format:

```
<tagname attribute="value" data-customattribute="value">
  More HTML tags or text
</tagname>
```

Although the attributes are optional, the start and end tags are required. Here's an example of a page div[ision] containing a paragraph on HTML escape characters:

```
<div>
  <p>
    You can make the literal greater than and less than symbols
    using &amp;gt; and &amp;lt;, respectively. The ampersand is
    the escape character, so printing a literal ampersand
    (as we are doing here) requires &amp;. Remember to include
    the semicolon!
  </p>
</div>
```

Tags that may not contain HTML, namely `meta`, `img`, `br`, and `hr`, are specified using just the start tag.

```
<tagname attribute="value" data-customattribute="value">
```

For instance,

```
<img src="http://web.mit.edu/img/MIT_logo.gif" alt="MIT logo">
```

creates an image element with the MIT logo as the source and "MIT logo" as the alt (mouseover/screen reader) text.

With tags in mind, let's look at a basic HTML page and go two for two on "hello world" examples:

**Example 1 – HTML says "Hello!"**

```
1  <!DOCTYPE html> <!-- specifies HTML5 standards mode -->
2  <html>
3    <head>
4      <!-- Page metadata (e.g. title, styles) lives here -->
5    </head>
6    <body>
7      <!-- Display elements and document content goes here -->
8      <h1>Hello, world!</h1> <!-- A top-level heading -->
9    </body>
10 </html>
```

The most important part of this example, by far, is the first line: the *doctype*. This signals to the browser that the document contains HTML5 and should be rendered according to the HTML5 specification. The page will, thus, be rendered in the same way by all compliant browsers (in a perfect world). If this is omitted, the document will be rendered in "quirks mode," in which the browser tries its best to display what it thinks the author meant, which may or may not be correct. The next most important part is on line 2 and 9: this is the that that defines the HTML root element. Everything inside `<html>` and `</html>` will be treated as—you guessed it—HTML. The two elements that `html` may contain are `head` and `body`. The former contains information about the page itself and its display properties (styling) and the latter contains the actual elements to display.

At this point, you may be thinking: "Okay, that's nice, but how am I supposed to run my scripts?" If this is, in fact, the case, then you raise a very important question. However, you may put yourself at ease since you will now be introduced to the almighty `<script>` tag!

## 1.2 The `<script>` tag

The `<script>` tag allows scripts (JavaScripts, to be exact) to be embedded directly into the page or loaded from an external file. The `<script>` tag is similar to other content-containing tags in that it must possess both a start and end tag. However, in the case of inline (i.e. non-external) scripts,

the inner HTML of the tag is parsed and executed as JavaScript. Here's an example:

**Example 2 – Scriptacular scripts!**

```
<script type='application/javascript'>
  console.log('I am a script running directly in the page.');
</script>

<script type='application/javascript' src='external_script.js'>
//Anything here is ignored
</script>


/* external_script.js */
//Note that this file does not contain <script> tags
console.log("I am a script loaded from an external file!");
```

Let's look at these two tags more closely. Notice how each has its `type` attribute set to `text/javascript`. This signals to the browser that the script is JavaScript and should be run as such. Additionally, the `src`, or source, attribute of the second tag specifies the location of the JavaScript file. While, in this case, `src` is pointing to a file named "external_script.js" located in the same directory (folder) as the loading page, `src` may be an absolute location specified by a URL using `http://` or a relative path using any number of `../`, meaning "go up one directory."

**Example 3 – Relative paths are absolutely amazing!**

```
<!--
HTML loaded from http://introjsiap.com/examples/js/index.html
-->

<script type='application/javascript' src='../../myfile.js'>
//Points to http://introjsiap.com/myfile.js
</script>

<script type='application/javascript'
src='http://introjsiap.com/scripts/myfile.js'>
//Points to http://introjsiap.com/scripts/myfile.js
</script>
```

As far as the placement of the `<script>` tags in the HTML document is concerned, they are valid anywhere in the `<head>` and `<body>` tags. However, there are a few choice locations to use that minimize page load times. In particular, `<script>`s should go towards the bottom of the the `<head>` tag if the script is involved in the rendering of the page or towards the bottom of the `body` tag, otherwise. This is due to HTML pages being loaded from the "top-down" and scripts blocking the loading of other resources.

**Example 4 – We want to know: where does the `<script>` go?**

```
<!DOCTYPE html>
<html>
  <head>
    <!-- styles, metadata, title, etc. -->
    <!-- scripts that draw/render/create the page go here -->
    <script type='application/javascript'></script>
  </head>
  <body>
    <p>Other HTML elements and text</p>
    <!-- other scripts go here -->
    <script type='application/javascript'></script>
  </body>
</html>
```

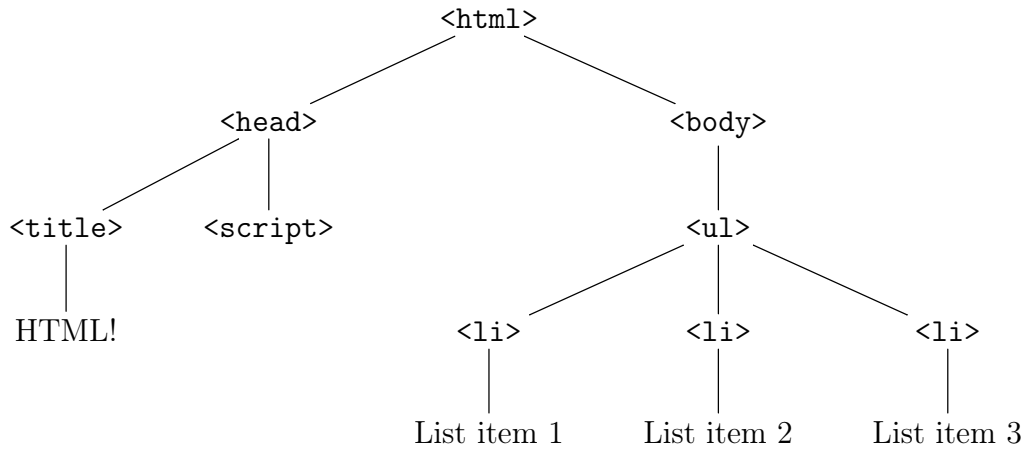
## 1.3 The HTML DOM

The HTML Document Object Model is an abstract, tree-like representation of the marked-up text. The nesting of HTML elements produces a hierarchy that describes the structure of the document. Intuitively, the relationships between elements, also called nodes, are described using the terms *parent*, *child*, and *sibling*.

In the following example, we will create an HTML document titled "HTML!" that contains a bulleted list containing three items and examine the resulting DOM.

## Example 5 – In the domain of the DOM

```
<html>
  <head>
    <title>HTML!</title>
    <script type='application/javascript' src='draw_page.js'>
    </script>
  </head>
  <body>
    <ul> <!-- an unordered list -->
      <li>List item 1</li>
      <li>List item 2</li>
      <li>List item 3</li>
    </ul>
  </body>
</html>
```




This is about as complicated as the structure of an HTML document is going to get. Of course, the trick is in knowing where to put the elements in order to make the page display properly. There are two important things to observe about this structure, however. The first is that a node contains

references to its parent node and all of its children, but not its siblings. The other is that the text "leaves" of the DOM tree—"HTML", and "List item *n*" in the example—are known as *text nodes*, but it's rare that they're ever created or referred to as such since it is more convenient to treat them as their parent's *inner HTMl*.

### A template, for your convenience

**Example 6 – The HTML page to begin all HTML pages**

```
<!DOCTYPE html>
<html>
  <head>
    <title>The title of the page</title>
    <meta charset='utf-8'> <!-- the text encoding -->
  </head>
  <body>
  </body>
</html>
```

This template contains the essential basics of any HTML document. The new `meta charset` tag allows the browser to correctly translate the received binary data into characters; this becomes especially important when using characters from other languages (i.e. non-ASCII or non-Western European).

## 2 Interacting with the HTML DOM

Hooray! Now for the really fun part: manipulating the DOM! It will be useful to start with the understanding that JavaScript gives you complete and absolute control over every aspect of your page's DOM **cue maniacal laughter**. Namely, JavaScript can be used to add and delete DOM nodes as well as modify their attributes.

**jQuery**

In the following sections and for the rest of this course, examples of new techniques will be presented using both the DOM API method and its corresponding jQuery equivalent. jQuery is an oft-used JS library that handles a lot of the boilerplate code required for making JS applications including convenience functions and abstractions for browser inconsistencies (resolving these is a major part of JS "debugging"). In the "Real World," you should mostly be using jQuery, but it's always nice to understand what's going on under the hood. Keep in mind that jQuery methods, unless otherwise specified, return DOM nodes wrapped as jQuery objects that contain jQuery functions; plain DOM nodes do *not* have any of the jQuery functions. Variables that contain jQuery objects will be preceded by the $ character.

## 2.1 The DOM in JavaScript World

In order to be able to modify the DOM, it needs to be accessible from JavaScript World, the abstract realm where JavaScript syntax is all-powerful. Fortunately, one of the objects that the browser extends into JavaScript World is the `window` object. The `window` object represents the browser and the HTML document. For our current purposes, we'll be focusing on the `window.document` object, which represents, none other than, the HTML document and provides a number of fun methods for accessing and changing its elements, all of which are represented as JavaScript objects.
Note: since `window` is the root object of all JavaScripts running in the page (including global variables), the `window` object specifier can be omitted.

## 2.2 Selecting HTML elements

Creating a page using only JavaScript is not particularly convenient, so interacting with the existing DOM is borderline essential. To do this, we must first obtain a reference to the desired node(s). There are a few ways of doing this:

- Recursively check every element for a specific property
- Select an element by its `id` attribute
- Select a group of elements by their `name` attribute

- Select a group of elements by their tag name
- Select a group of elements by their class name
- Select a group of elements using a CSS selector

The first four methods, and, to some degree, the fifth, are straightforward and only rely on the value(s) of the HTML element's attributes.

### Recursively checking every DOM node

The first method, recursively checking every node for a specific condition, is effectively what the browser does internally when it selects a group of elements by `name`, tag name, `class`, or CSS selector.

## Example 7 – Meet the HTML family

```
<html>
<!-- <head> omitted for brevity -->
<body>
  <div> <!-- a page div[ision] -->
    <span id='someText'>This is some text.</span>
  </div>
  <script type='application/javascript' src='DOMRecursion.js'>
  </script>
</body>
</html>
```

```
/* recursiveDOM.js */
//Functions similarly to document.getElementById()
function reimplementTheWheelById(id, parentNode) {
    //If parentNode is undefined, set it to the root node
    parentNode = parentNode || document.documentElement;

    if(parentNode.id === id) {
        return parentNode;
    }

    var children = parentNode.children;
    for(var i=0; i < children.length; i++) {
        var found = reimplementTheWheelById(id, children[i]);
        if(found !== null) {
            return found;
        }
    }
    return null;
}

var someText = reimplementTheWheelById("someText");
console.log(someText.innerHTML); //"This is some text."
```

Although the browser uses a hash map of `id`→element, this example demonstrates a JavaScript implementation of the second selection method, `document.getElementById()`, which selects an element based on its unique `id` attribute. The basic idea is to perform a depth-first search of the DOM tree until a node with an `id` matching the first argument is found. The interesting parts of this example are on the $18^{th}$, $20^{th}$, and $29^{th}$ lines. The first of which is the `children` property of a DOM node which returns an HTMLCollection object which is actually just a fancy Array of DOM node objects.

```
DOMNode.children; //returns an HTMLCollection of child nodes
                  //HTMLCollection is just a fancy Array

$jqObj.children(); //returns the children as a jQuery object
                   //jQuery objects are really fancy Arrays

//Also,
DOMNode.parentNode; //returns the parent DOM node

$jqObj.parent();    //returns the parent jQuery object
```

The next is the setting of the optional argument `parentNode` that keeps track of the iteration through the tree. Finally, on line 29 is the `innerHTML` property of the DOM node. This returns a String containing the contents of the DOM node, including any HTML tags.

```
DOMNode.innerHTML; //returns the String'd content of the node
                   //including any HTML tags
```

### Selecting an element by its id

Okay, now let's look at how to select an element by its `id` using a single line!

```
/* DOM method */
document.getElementById("theID"); //returns a DOM node

/* jQuery method */
$("#theID"); //returns a jQuery object wrapping a DOM node
```

Since the select by `id` methods return the first matched element, assigned `id`s should be unique.

The # in the "#theID" is the CSS `id` selector; there will be more on this, and more, in just a bit.

### Selecting a group of elements by their `name` attribute

The following methods select every HTML element that has its `name` attribute set to the specified name.
Note that the jQuery syntax is the same as the CSS selector for a generic attribute.

```
/* DOM method */
document.getElementsByName("name");
//returns an HTMLCollection of elements with the given name

/* jQuery method */
$('[name="name"]'); //returns the same as a jQuery object
```

### Selecting a group of elements by their tag name

The following methods select every HTML element defined using a specified tag. For instance, selecting by the `div` tag name will return all `<div>` elements in the page.

```
/* DOM method */
document.getElementsByTagName("tagname");
element.getElementsByTagName("tagname");
//returns an HTMLCollection (Array) of <tagname> elements

/* jQuery method */
$("tagname"); //returns the same elements as a jQuery object
```

### Selecting a group of elements by their class name

The class of an element is a space-delimited String of names that the browser uses to determine which CSS rules to apply to the element. The following methods select every HTML element that contains a certain class name. For

instance, selecting by the `div` tag name will return all `<div>` elements in the page.

```
/* DOM method */
document.getElementsByClassName("class1 optionalClass2");
element.getElementsByClassName("class1 optionalClass2");

/* jQuery method */
$(".class1 .optionalClass2");
```

## Example 8 – Keeping it classy

```
<html>
  <body>
    <p class='bodyText flavorText'>
      <span class='fourthWall'>
        Hmm. I seem to be in an example.
      </span>
    </p>
    <span class='bodyText'>Text, everywhere!</span>
  </body>
  <script type='application/javascript'
          src='jquery.min.js'></script>
  <script type='application/javascript'>
    //selects both the <p> and the second <span>
    var bodyText = document.getElementsByClassName("bodyText");
    //selects just the span
    var sr = bodyText[0].getElementsByClassName("fourthWall");

    var $bodyText = $(".bodyText");
    var $sr = $bodyText.find(".fourthWall");
  </script>
</html>
```

### Selecting a group of elements using a CSS selector

The selectors used by the following functions are the same as those used to apply CSS rules to HTML elements. You've already seen some examples of selectors. #, blank, and . are the selector tokens for `id`, tag name, and `class`, respectively.

```
/* DOM method */
document.querySelector(selectors);

/* jQuery method */
$(selectors);

//selectors is a space-delimited String of CSS selectors
```

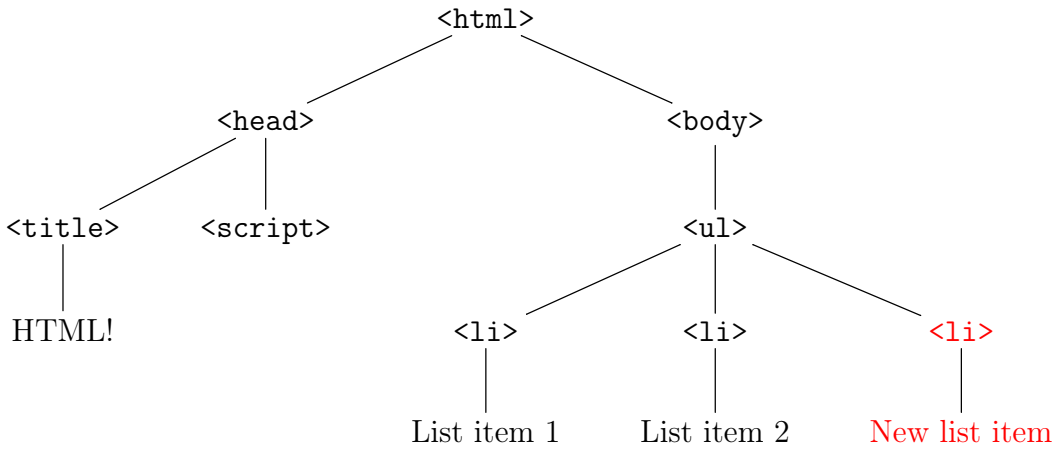
## 2.3 Adding and removing HTML elements

### Adding elements

```
/* DOM method */
var newElement = document.createElement("tagname");
otherElement.appendChild(newElement); //insert at the end
otherElement.insertBefore(newElement, beforeElement);

/* jQuery method */
var $newElement = $("<newElementTagName>");
$otherElem.append($newElement);
$newElement.appendTo(other);
//other can be a selector, jQuery object or DOM node
//$otherElem is a jQuery object
//$newElement is appended to every element in the selection
```

If you think of the DOM as a tree, appending a node adds it as the rightmost child of its new parent node. In the HTML document, it would be equivalent to adding a new element right before the end tag.




### Removing elements

```
/* DOM method */
var removeMe = document.getElementById("removeMe");
var removed = removeMe.parentNode.removeChild(removeMe);
//returns the removed element

/* jQuery method */
var $removeThese = $("someSelector");
$removeThese.remove();
$removeThese.detach(); //removes but preserves jQuery data
```

jQuery provides a much richer set of manipulation functions than the DOM API. I recommend checking out the docs! You can find a link to them in the *Additional Resources* section.

Since the different append and insert methods might get confusing, a helpful way to remember the syntax is by thinking about it as a sentence. For instance, if I want to append my new element and insert it before another element, the sentence would be "Take this element and insert it before this other element" and the function would be `insertBefore`.

## 2.4 Modifying HTML elements

### Inner HTML

```
/* DOM method */
element.innerHTML;                    //getter
element.innerHTML = "some String";    //setter

/* jQuery method */
$element.html();                      //getter
$element.html("some string");         //setter
```

While it is possible to add elements using this method, it is significantly less performant than the DOM methods. However, this is a great way to remove all child nodes (by setting the inner HTML to "") or add a text node.

### Attributes

```
/* DOM method */
element.getAttribute("attribute");
element.setAttribute("attribute", "value");

/* jQuery method */
$element.attr("attribute");           //getter
$element.attr("attribute", "value");  //setter
```

**Style/CSS**

```
/* DOM method */
//To set a single property
element.style.cssProp;                 //getter
element.style.cssProp = "CSS String"; //setter

//To set many properties using a  class
element.className += " newClassName";
//removing a class requires string manipulation

/* jQuery method */
//To set a single property
$element.css("prop");                  //getter
$element.css("prop", "CSS String");   //setter

//To set many properties using a class
$element.addClass("className");
$element.removeClass("className"); //automagical
```

**Important:** When accessing CSS properties using the DOM API, make sure that you're referring to them using the `style` object of the DOM node. Simply setting `cssProp` of the node itself will result both in the page not changing, and you wondering why it's not working since the syntax looks correct. It gets even better (worse) when you try logging `node.cssProp` and it prints out what you expect.

# 3 CSS

After all of this talk of CSS, you're surely wondering what it is. The name CSS stands for Cascading Style Sheets, but this isn't very informative. Essentially CSS is a set of styles to apply to matched elements and their children (hence the cascading part). If HTML is the JS GUI language, then CSS is the GUI "skinning" language. Let's look at a simple example:

## Example 9 – Sí, es CSS

```
1  <html>
2  <head>
3    <style type='text/css'>
4      div { /* selects all divs */
5        width: 100px;
6        height: 100px;
7        background-color: #00ff00;
8      }
9      #divToo { background-color: #ff0000; }
10   </style>
11 </head>
12 <body>
13   <div>
14     I will look like a green square.
15   </div>
16   <div id='divToo'>
17     I will look like a red square.
18   </div>
19 </body>
20 </html>
```

On line 3, we create a `style` tag of the type `text/css` and fill it with a two CSS rules. The first rule selects all `div` elements and makes their width and height 100px and their background color green. The second selects the element with id `"divToo"` and makes its background color red. This is a good example of how rules applied by more specific CSS selectors take precedence over those that cascade. CSS syntax looks like this:

```
selector { /* This is a comment */
  property-name: value;
}
/* in JavaScript, property-name becomes propertyName */
```

In general, the way to include stylesheets in a page is by placing one of the following in the `head` section of the page.

```
<style type='text/css'>
  /* styles go here */
</style>

<link rel='stylesheet' type='text/css'
      href='path/to/stylesheet.css'>
<!-- basically the same as including JS -->
```

## 3.1 Selectors

This section is not meant to be a comprehensive list of selectors, but rather a list that captures the capabilities of CSS selectors. For a complete list, see the link in *Additional Resources*.

*****

selects all elements

**tagname**

selects all elements of type `tagname`

**#elementId**

selects the element that has the `id` of `elementId`

**.class-name**

selects an element that has the class `class-name`

**first-selector second-selector**

selects the descendants of the elements selected by `first-selector` that match `second-selector`

**selector[attr]**

selects an element that matches the basic selector, `selector`, and has `attr` as an attribute

**selector[attr="value"]**

same as the previous selector but `attr` must equal `value`

**selector:first-child**

selects the first child of the element selected by `selector`

**selector:hover**

pseudo-class that applies the style when the user is hovering over the selected element

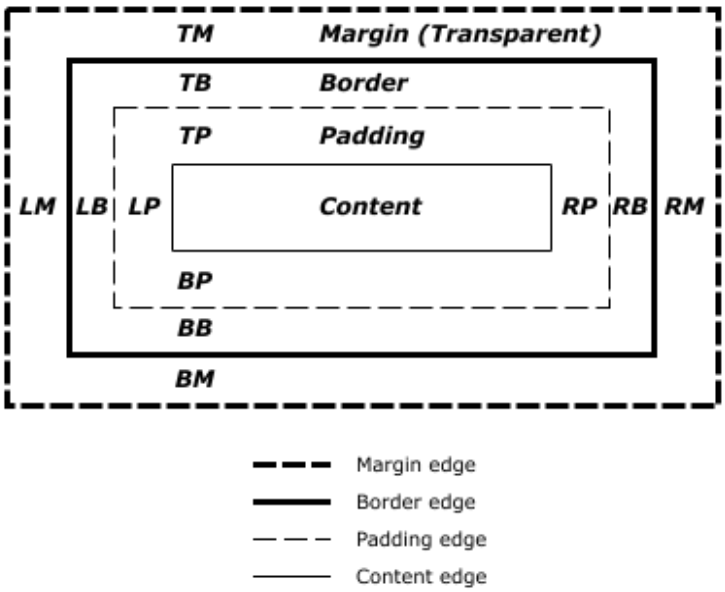
## 3.2 Sizing

Sizing comes in three main flavors: `px`, `%`, and `em`. The first of which, `px`, is a special CSS unit that depends on the device but looks the same across all devices, the second, %, is an amount relative to the size of the parent element. The third, `em`, is the not related to the box model, but rather the font size relative to the parent element; this should be used for sizing fonts.

## 3.3 Box model

The CSS box model applies to all block-level elements, or those that have their `display` style property set to either `block` or `inline-block`.

A block-level element is one that is preceded and followed by a line break and has no elements to its left or right. `<div>`, `<p>`, `<ul>`, and `<li>` are some examples of elements that are block-level by default. For a complete list, see *Additional Resources*. The `inline-block` property produces an element that uses a box model like a `block` element but allows elements to its sides like an `inline` element.




There are four parts to the box model. From the inside out, they are: content, padding, border, and margin. The content is, as expected, the contents of the element, like text or other elements. The padding is a visible region of the element in which no content can appear. Outside of this is the border which marks the end of the element's visible region. The last component of the box model is the margin, or a region in which other (statically positioned) elements can not exist.

Each of these properties can be set using a shorthand property like `margin: 20px` or `border: 2px solid black`. Additionally, by using the `-left`, `-top`, `-bottom`, and `-right` modifiers, the sizes (and colors and styles, in the case of borders) of the components' constituent parts can be set individually (for instance, `padding-top:10px; padding-bottom:20px;` ).

## 3.4 Positioning

Other than the ubiquitous `inherit` value, the `position` CSS property can be either static, relative, absolute, or fixed. `static` is the default and causes the element to be positioned according to the usual rules.

For the next three, positioning is accomplished using the `top`/`bottom`, `left`/`right`, and `z-index` properties. The first two groups represent the

distance of the element's bounding box's top/bottom and left/right edges from those of its positioning reference. The third determines the order in which the element is drawn; an element with a higher `z-index` is drawn on top one one with a lower `z-index`.

A `position` of `relative` causes any position changes to be relative to the `static` position of the element.

An absolutely positioned element is positioned relative to the entire document (the whole page). It is essentially popped out of its parent element(s) and pasted somewhere else on screen.

An element that is `fixed` is positioned relative to the browser's window so that any change in the document's position (i.e. through scrolling or zooming), does not cause the element to move.

## 3.5 Other style attributes

As with the Selectors section, this is also meant to give you an idea of some of the more interesting CSS properties.

- `color`: hex, rgb(), or name
  sets the text color
  ex: `color:  #beefed; /* a light blue */`

- `background-color`: `hex`, `rgb()`, or `name`
  sets the background-color of the element's content and padding
  ex: `background-color:  orange;`

- `width`: px or %
  sets the width of the element's content

- `height`: px or %
  sets the height of the element's content

- `border-radius`: px or %
  sets the radius of the element's corners
  use this to make rounded corners
  be careful, this may be copyrighted

- `font-size`: em