**The Box Model: http://guistuff.com/css/css\_boxmodel.html**

Practical usage of CSS requires some understanding of how a page is 'rendered' by a browser. That is to say, how styles are attached to elements, and these elements turned into pixels on the browser viewing pane. CSS and HTML/XHTML are rather intermingled when it comes to rendering by the browser, since most HTML/XHTML tags have inherent style elements that are attached to them, regardless of whether or not they're specifically styled. In fact, many HTML/XHTML tags could be replaced by a properly crafted class that was simply attached to a SPAN element. We know, of course, that there's a reason for the different tags, and that semantic web coding makes for better-structured sites, but if visuals were the only thing that concerned us, then CSS could replace almost any inherent HTML/XHTML tag property we could think of. This is exactly how web browsers "think". They don't see an H1 tag and look up the spec for rendering an H1 element in HTML, they know that, even with no CSS around, an H1 element is simply text with the properties of display: block, margin-top: 1.Xem, margin-bottom:1.Xem, font-size: 2.Xem, approximately. Within the browser, there's complete separation between structure and style.

CSS treats visual elements as rectangles that flow along the document. Although CSS is a broad specification that applies to many devices, this text will focus on how these elements are rendered in web browsers.

To understand how a page is rendered, the box model must first be explained. Visually, it's actually rather simple:



All visual elements within a page are considered to be rectangular boxes, each with its own [padding](http://guistuff.com/css/prop_padding.html), [border](http://guistuff.com/css/prop_border.html), and [margin](http://guistuff.com/css/prop_margin.html) properties. The [background-color](http://guistuff.com/css/prop_background-color.html) of an element will effect its content area and its padding, but not the border or margin. The border type can be set to different styles, but its color defaults to the 'foreground' color, which also effects the text within the element. The margin is always transparent, and can be considered the space with which the element distances itself ('pushes' itself away) from adjacent elements (this analogy helps in some cases, but hinders in others).

Width (size) for padding, border, and margin default to 0 if not set explicitly. Margins are the only properties that may be negative. When referring to a box's width and height, the abstract definition refers to that box's entire contents, including padding, border, and margin. Otherwise, the content area box may be referred to specifically, which means that only the width and height of the content area are considered. In practice, the box's width and height differs from one browser implementation to the other -- this will be discussed below.

**Block elements and Inline elements**

There are two types of boxes -- **block**, and **inline**. In CSS, the [display](http://guistuff.com/css/prop_display.html) property can set an element to either inline or block. Inline elements are elements that do not form their own content block. Text, images, and span areas are all **inline** elements. Their contents is distributed in lines, normally flowing left-to-right, top-to-bottom. Some elements appear to break the normal flow of content, and create a division when they are placed within a document -- for example: DIV, P, Hx, and TABLE elements. These are all inherently **block** elements and create their own content blocks. They have the properties of block elements without the need to explicitly set their [display](http://guistuff.com/css/prop_display.html) property to block. You may still turn any inline element into a block element by setting its [display](http://guistuff.com/css/prop_display.html) property to block. The other way around, however (turning inherently block elements into inline elements via the display property), is not fully supported and is not recommended.

**The Document Tree**

Not to be confused with the more complex 'Document Object Model', the Document Tree is a relationship set that XML, HTML, and XHTML adhere to. Technically, non-strict HTML can stray from this set of relationships and still be rendered by a browser, but this is only because the browser "guesses" the author's intentions and fixes the errors on-the-fly.
Basically, elements within an HTML/XHTML page form a "document tree". This means that every element within the document has exactly one **parent**, except for the HTML element, which is the **root** element, and thus has none. That is the first 'relationship' of the document tree (the Parent-Child relationship), but there are other relationships between elements that will determine how they will be displayed. The two most basic types of relationships are the **preceding** element, and the **following** element. Both of these elements are optional, since the element may be the only 'Child'. Let's take a look at an example:

<html> <head> <title>Page Title</title> </head> <body> <h1>Welcome!</h1> <p>This is some <em>example</em> text.</p> <p>This is the second paragraph on this page.</p> </body> </html>

If you were to break down the above document into its relational tree, you'd notice quite a few relationship definitions. Since this is not a discussion about data structures, we'll focus on the ones that actually effect how elements are displayed on a page. The first relationship actually doesn't effect much, apart from being present: the HTML element is the **root** element, and has no parent elements. Let's skip to the BODY of the page. There's an H1 element, which we know to be a Block-level element, followed by two P elements (also Block-level elements), the first of which contains an EM element. The BODY is the parent of H1, and the two P's, but not the EM. Technically, the BODY element is an an **Ancestor** of the EM element, since it is a parent of the EM's parent (the ansector relationship cascades indefinately, so the HTML element is actually the ancestor of all of the document's elements). Finally, let's reach the two basic relationships mentioned above: preceding and following. The H1 has no preceding elements, as it's the first child of the BODY. It does, however, have a following element -- the first P (the one that contains the EM element). The first P (again, the one with the EM element) has both a preceding element, the H1, and a following element, the last P. The last P has one preceding element, the first P, and no following elements, since it's the last child element of the BODY. The EM has no preceding or following element, since it's an 'only child'. In fact, just one more relationship hasn't been covered, even though it's presented nicely within this example: All of the child elements of the BODY element are **Siblings**. So the H1 and the two P's are Sibling elements.

Why is all of this data-stucture-related material here then? How does it effect how things are rendered in a web browser? Well, as mentioned above, boxes have directions to all of their box-model properties. Their padding, border, and margins can all be assigned different values to different directions. Look at the relationships of the elements from the example above. The H1, and both P's, will be placed one after the other, since they're all Block elements. However, the EM within th first P is an inline element, which means it will flow along with the document. In other words, while the boxes of the H1 and both P's mainly concern themselves with vertical properties, the EM element will be surrounded with elements on all sides. While the preceding element of the last P is the first P, which is above it, the preceeding element of the EM element is the text "This is some ", which is in fact considered by the browser to be another element, a sub-content element of the first P, which is placed to the left of the EM element. Block and inline elements effect whether preceding or following elements are placed vertically, or horizontally.

**Box Hierarchy**

Vertical vs. Horizontal placement is just one thing to keep in mind. Consdier how this type of relationship applies to the boxes within the document. All boxes within a document are contained within other (parent) boxes, or blocks. If there is no block to contain the most external box, then the browser window is considered to be the outer-most box (or the root block). Hence, when assigning percentage values to the [width](http://guistuff.com/css/prop_width.html) and [height](http://guistuff.com/css/prop_height.html) properties of non-absolutely-positioned elements, these percentages are of the corresponding width and height of that element's containing (parent) block. Now consider how this would apply to designing lists, or nested DIVs and SPANs. You have to take many variables into account, or you'll be surprized (and likely rather annoyed) at the outcome.

**The Box Model Problem**

If all of this wasn't enough to get your mind in a knot, there's still the browser compatibility problem to deal with. Yes, there's a major compatibility issue that persists and will continue to persist in modern browsers. There are actually **two** Box Models, as it were. Don't worry, it doesn't mean you have to re-learn anything, just to understand how Mozilla (FireFox) sees things as opposed to how Internet Explorer sees them. The W3C CSS specification defines the width of the box as the width of the **content area alone**, not including the padding, border, or margin. Internet Explorer implements another Box Model, which is valid, it's simply not the default: in it, the width of a box includes both padding and border. This sounds like a bigger problem than it is, though it's still a respectably-sized problem. Before going into the 'how do I throw in the magic words into my CSS' solution, consider that there's a strategic solution. Instead of creating elements and setting their size to the shifting width value, try to design pages so that you 'pour' content into padding-less, border-less elements instead. For example:

<style type="text/css"> <!-- #container { width: 200px; } .fluid { border-style:solid; border-width: 5px; border-color: #999; padding: 5px; width:auto; } --> </style> <div id="container"> <p class="fluid">Content!</p> </div>

It's important to note that the width of the fluid class is set to auto, and not 100%. If it is set to 100%, then you're back to square one, since Mozilla/FireFox will assume that your intention was to set the content area to 100% of the parent's width, which is 200px. However, if your design needs to implement the second Box Model, which many consider to be the more intuitive one, there is a special sauce that you can use:

box-sizing: border-box -moz-box-sizing: border-box;

By placing the above declarations in your CSS, you're explicitly telling the browser that you want the box size to include its padding and border. Browser support may vary, and you may want to check compatibility. As usual, you can always have two different style sheets for the two models/browsers, but then you're involving scripting when you can possibly avoid it.

There's a lot more to cover when it comes to document rendering, but hopefully this answers a few primary threads of interest that pertain to the Box Model and some of the Visual Formatting Model. Remember to test your pages on multiple browsers, and, when possible, platforms. Also, when it doubt, stick to the standards, and then condition your way to the cross-browser goal. Check which browsers support which parts of the standards properly, and fall back from there, not the other way around.