

98-130

TECHNIQUES FOR WEB INTERFACES

LESSON 3: "VISUAL FORMATTING MODEL"

LECTURE OVERVIEW

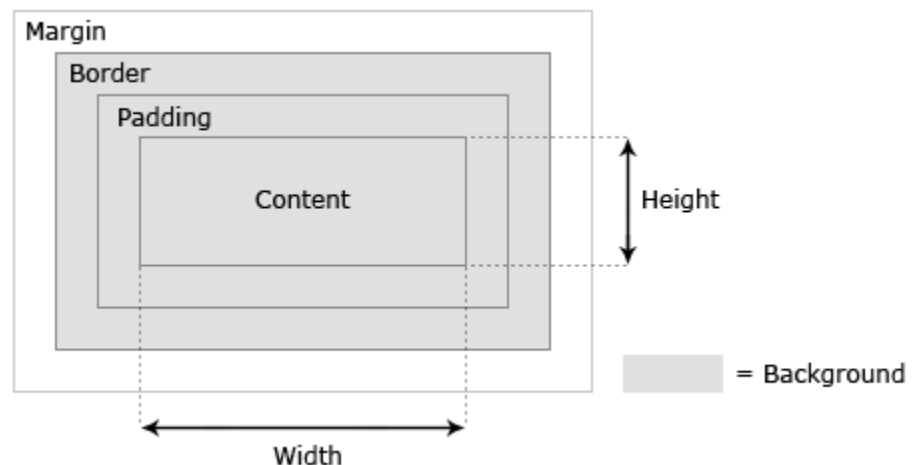
- The box model
 - The `display` property: block, inline, none
 - The `position` property: static, relative, absolute, fixed
 - Third dimension: `z-index`
 - Floats and clearing
-

RELATED READING

- Kilian Valkhof: "*Understanding CSS Positioning*" (Parts I and II)
kilianvalkhof.com/2008/css-xhtml/understanding-css-positioning-part-1/
kilianvalkhof.com/2008/css-xhtml/understanding-css-positioning-part-2/
 - W3Schools: Sample Border Styles
http://www.w3schools.com/css/css_border.asp
-

THE BOX MODEL

The CSS *box model* describes the structure and dimensions of an HTML element. We will introduce a few CSS properties: border, margin, padding, width, height, and background as we go. Every HTML element makes a *box* of some kind, and the layout of this box is defined by the box model:



The grey area indicates the background of the element, which can be a background-image or a background-color. There are a few different properties dealing with the background, including:

- `background-color: <color here>;`
Explained in the quiz.
- `background-image: url(someimage.gif);`
Pretty straightforward. Remember the `url(...)` thing though. Note that you can only have one background image per element; this is a nasty little thing, and CSS3 (discussed later!) will make it possible for us to have multiple background images.
- `background-position: <some position>;`
This will position the background image. For example, we can say `background-position: top right;` and the background will be aligned with the top right of our element, or we can say `center center` and it will be aligned in the center. Note that the first keyword (top, center, bottom) will specify the vertical alignment and the second keyword (left, center, right) will specify horizontal alignment.
We can also use pixel (px) or % values instead of keywords; these will offset the background image from the top left of the element, but in a classic CSS screw-up, the ordering is mixed up. Contrary to how the keywords above work, the first unit specifies *horizontal* offset and the second specifies *vertical* offset. So we can say `background-position: 40px 70px;` and the background image will be 40px to the right and 70px down.
It's best to stick with one kind of unit in this declaration. Some browsers will really get confused if you say `background-position: top 50px`, so say `50px 0px` instead (note the order switched!).
- `background-repeat: repeat, repeat-x, repeat-y, or no-repeat;`
`repeat-x` will cause the background to tile horizontally only, while `repeat-y` will cause it tile vertically only and `no-repeat` means don't tile at all.
Defaults to `repeat`, which will repeat in both directions.
- `background: <color> <image> <repeat> <position>;`
This is a shorthand property that makes it easy to specify them all in one go. The order is important, but you can leave any values out if you don't need to declare them.

The `width` and `height` properties can be defined using any units of measurement (as discussed in the last lecture) that we want: for example, `width: 50%` or `height: 300px`. Note, though, that they do *not* define the entire width or height of the box! They define the width and height of the *content*! This is really confusing to many people (including the developers of

IE5.5 and below). To get the *total* width or height of your entire element, you have to add up the values of your margin, border, padding, and width/height.

This brings us to padding and margin; there are really four values each: top, right, bottom, and left. We can specify these individually using `padding-top`, `padding-bottom`, `padding-left`, `padding-right` (and similarly for `margin-top`, etc.), or we can use the shorthand `padding: <top> <right> <bottom> <left>` and `margin: <top> <right> <bottom> <left>` properties. (An easy way to remember is that it goes clockwise!) So:

```
padding: 4px 5px 6px 7px;
```

is the same as:

```
padding-top: 4px;
padding-right: 5px;
padding-bottom: 6px;
padding-left: 7px;
```

This leaves us only with border to explain. Again, we can specify `border-top`, `border-left`, etc. We also have a shorthand `border` property, which just says “apply this border to all sides,” but we can’t really specify one side or the other. All border declarations take the form of `<width> <style> <color>`. The most common border styles are solid, dashed, and dotted, though there are others (look at the Related Reading). For example:

```
border-left: 5px solid black;
```

will apply a solid 5-pixel-wide black border to the left of the element.

```
border: 1px dotted #aaaaaa;
```

will apply a small dotted grey border to all sides of the element.

The box model describes how all of these properties coexist and, put together, define the overall look and structure of the box.

MARGIN COLLAPSING There is a bit of a trick with margins that we haven’t mentioned yet, in the form of *margin collapsing*. Whenever the vertical margins of two elements touch, they will “collapse” into a single margin (equal in size to the bigger of the two). Note that this doesn’t happen for horizontal margins (`margin-left` and `margin-right`). If you understand those past two sentences, you know basically everything about margin collapsing that you need to know. This is really handy, for example, if you’re defining the margins of paragraphs; one paragraph’s bottom margin will meet up with the second paragraph’s top margin. Margin collapsing ensures that we don’t have double the margin we want in this case.

Optional Side Note: The technical details behind margin collapsing are confusing, however, because collapsing doesn't apply to floated elements, absolutely positioned elements, etc. (soon to be discussed)

DISPLAY: BLOCK, INLINE, NONE

So if every element renders a box of some kind, the `display` property determines what *kind* of box. The two main kinds are *block* and *inline*. Since each HTML element has some default box type, their corresponding HTML elements are referred to as *block-level elements* and *inline elements*. We can change the defaults by using the `display` property.

BLOCK-LEVEL ELEMENTS Block-level elements, like `P`, `H1`, and `DIV`, will always start on a new line, and by default take up 100% of the width of their containing element. We can change all the box model properties, including width, height, margin, and padding, and can switch to using a block box by saying `display: block`.

INLINE ELEMENTS Inline elements, like `STRONG`, `A`, and `SPAN` will not generate a new line; instead, they are treated as part of the flow of the document and can wrap around to new lines. However, we cannot set the dimensions (width, height) and can only set horizontal margins (`margin-left` and `margin-right`). We can switch to using an inline box by saying `display: inline`.

Optional Side Note: We can set `padding` on all sides of an inline element, but only horizontal padding will affect other elements. So we could, for instance, make the background extend vertically (remember the box model; background is included in the padding and border!) but not horizontally. This isn't really useful though, honestly.

DISPLAY: NONE We can make an element *not* generate a box by setting it to `display: none`. This will basically make it disappear. This is pretty handy for dynamically hiding/showing content using Javascript, and even has some uses for showing special content to blind people that normal users can't see (we'll discuss this in a later lecture, along with an even better method to do this).

OTHER BOX TYPES There are several other kinds of boxes, including `inline-block` (that don't generate new lines and line up next to each other like inline boxes, but have controllable dimensions like block boxes) and `list-item` (as used by the `LI` element). But as a whole, as beginners we don't really need to worry about these.

POSITION: STATIC, RELATIVE, ABSOLUTE, FIXED

The `position` property provides interesting ways to change the element's fundamental position. The primary values that `position` can take are `static` (default), `relative`, `absolute`, and `fixed`.

STATIC This is the default value for `position`. The element is positioned normally, and you cannot use `left`, `top`, `z-index`, etc. to modify its position (these properties will be discussed in a bit).

RELATIVE Relative positioning keeps the element exactly where it was under the default `static` scheme, with a couple of major differences. Most importantly, you can use `left`, `right`, `top`, and `bottom` to shift the element from its normal position. For example, examine the following CSS:

```
div#some_content {  
    position: relative;  
    left: 3px;  
    top: 50px;  
}
```

This will take the `DIV` with an ID of `some_content` and shift it down by 50 pixels and to the right by 3 pixels (aka, 3px from its left edge and 50px from its top edge). Note that we can also use negative values for these 'shift' properties: for example, `left: -50px`.

There are two other differences between relative and static position schemes: the ability to use `z-index` (discussed later) and how the two handle absolutely positioned children (discussed next).

ABSOLUTE Absolute positioning is a very powerful tool. Simply declaring an element to have `position: absolute` will take it completely out of the normal flow of the page; by this, I mean that other elements will be positioned as if the absolutely positioned element never existed.

So where does the absolutely positioned element end up? It's up to you. The `left`, `right`, `top`, and `bottom` properties are used in the same way that we saw previously; however, this time, the element isn't shifted according to its normal position. Instead, it gets shifted based on the edges of its *nearest non-static positioned ancestor* (if there are none, it defaults to the entire web page/body). To demonstrate, let's take a look at a simple example:

```
...<body>  
    <div id="primary_content">  
        Lorem ipsum...  
    </div>
```

```
</body>...
```

With the following CSS:

```
#primary_content {  
    position: absolute;  
    top: 200px;  
    right: 100px;  
}
```

The `primary_content` DIV in this example will have a right edge that is 100 pixels to the left of the *right edge of the page*, and a top edge that is 200 pixels down from the *top edge of the page*. Note that this is in contrast to the relative positioning scheme, which would have just shifted it down 200px and to the left 100px from wherever it would've been in the first place.

As previously mentioned, though, if we have a non-static positioned ancestor (like another DIV set to `position: relative`), it'll be shifted relative to that instead of the entire page.

FIXED Fixed positioning is the same as absolute positioning, with one major difference. Instead of being shifted relative to some containing block, a fixed element will always be shifted according to the *browser window*. This is important, because it means that when a user scrolls, that element will remain in the same place!

This can be pretty powerful and cool, but slightly dangerous and trickier than you'd think. For example, we can 'fix' a small navigation menu in place on the side of our page somewhere; this way, when the user scrolls down the navigation is still available to him without scrolling up (a major boon for usability). However, if the browser window is too small to accommodate the menu, the user won't be able to see the rest of it without resizing their browser.

Z-INDEX

As we've seen, positioning with `relative`, `absolute`, and `fixed` provides ample opportunities for the elements on our page to overlap with each other. The `z-index` property provides a fairly simple way to specify which elements should stack on top of one another. An element with a higher `z-index` will appear on top of an element with a lower one. `z-index` can be negative, and defaults to 0.

FLOATS AND CLEARING

We can also *float* an element to the left or to the right, and inline content will wrap around it (important: *block* elements won't!). The default value for `float` is `none`, but we can set it to `left` or `right` to get this behavior. The best way to learn this is by example, so that's exactly what we'll do.

CLEARING Using the `clear` property, we can override and “stop” a float. This property can be `none` (default), `left` (no floating elements allowed along the left side of the element), `right` (no floating elements allowed along the right side), or `both` (no floating elements allowed along both sides). For example, let's say we have an image floated to the left, with two paragraphs wrapping around it:

```

<p>Lorem ipsum blah blah...</p>
<p>I don't want to be next to an image!</p>
```

```
.thumbnail { float: left; }
```

If we give the second `P` tag the `clear: left;` declaration, it will no longer wrap around the image; instead, it will start right below the image.

Clearing works by using some crazy invisible margin trickery. When we said `clear: left`, the browser actually added just enough `margin-top` to our `P` element, pushing it down to the point where it was no longer affected by the floated image.

OPTIONAL: NUANCES The nuances of floats are pretty tricky, and we won't fully cover them. An important thing to note is that floating an element, like positioning it absolutely, takes it out of the normal flow of the page. In other words, it essentially takes up no space. So if we have a containing `DIV` that has some floated elements, like:

```
<div id="container">
  <div id="some_box">Lorem ipsum.</div>
  <div id="another_box">And another one.</div>
</div>
```

With the CSS:

```
#some_box, #another_box { float: left; }
#container { background-color: yellow; }
```

What will happen? The two inner `DIVs` will be floated against one another, but remember, they take up no space! So the container `DIV` will not expand to fit them, and the yellow background color won't either.

We can fix this in two different ways. One way involves what we learned earlier about how clearing works; if we have some other element that clears the floats, a load of top margin will be added. The containing `DIV` will then expand to fit this cleared element along with its margin, and thus will seem to contain the floats as well!

```
<div id="container">
```

```
    <div id="some_box">Lorem ipsum.</div>
    <div id="another_box">And another one.</div>
    <br style="clear: left;" />
</div>
```

The other way is to change a property called `overflow` for the container `DIV`. This is ideal because it doesn't involve an extra non-semantic element (like the `BR` above), but it requires that we know what `overflow` does. Essentially, this property changes what happens when there is too much content (overflow) for a containing element that has dimensions set. It defaults to `visible`, which means that the overflow is displayed but is displayed outside of the element's box (in other words, the containing element won't expand to contain it; it'll just kind of hang off awkwardly). If we change `overflow` to `hidden` (don't display the overflow at all) or `auto` (display a scroll-bar on the containing `DIV` if necessary), then the containing `DIV` will actually expand to include its floats as well. We won't worry about exactly why this happens. Keep in mind, though, that you possibly don't want `overflow` to be one of those values.

```
#container { overflow: auto; }
```

By the way, there *is* a way to clear a float without using an extra element. The details are nasty, but feel free to Google for "clearfix" if you want.

NEXT LECTURE

This lecture basically concludes our basic quick CSS tutorial. The next two lectures will primarily be case studies and examples, and we'll learn how to do everything from multi-column layouts to tab-style navigation. Hopefully, our knowledge of the visual formatting model will become a bit more concrete as we try out examples.