

CSS Advanced Guide

html5dog.com

The CSS Advanced Guide is for those who want to push CSS to the extreme, highlighting methods that might not be immediately obvious to the intermediate developer.

It's a little small at the moment, but expect articles on alternate stylesheets, styling for print and more soon...

The Display Property

A key trick to the manipulation of HTML elements is understanding that there's nothing at all special about how most of them work. Most pages could be made up from just a few tags that can be styled any which way you choose. The browser's default visual representation of most HTML elements consist of varying font styles, margins, padding and, essentially, display types.

The most fundamental types of display are **inline**, **block-line** and **none** and they can be manipulated with the **display** property and the values **inline**, **block** and **none**.

inline does just what it says - elements that are displayed inline follow the flow of a line. **Strong**, **anchor** and **emphasis** elements are traditionally displayed inline.

block puts a line break before and after the element. **Header** and **paragraph** elements are examples of elements that are traditionally displayed block-line.

none, well, doesn't display the element, which may sound pretty useless but can be used to good effect with accessibility considerations (see Accessible Links), alternate stylesheets or advanced hover effects.

The original default style-sheet for this site for example, manipulates a few traditionally in-line and block-line elements to fit the design.

```
h1 {
    display: inline;
    font-size: 2em;
}
#header p {
    display: inline;
    font-size: 0.9em;
    padding-left: 2em;
}
```

This enabled the title 'htmldog.com' and the tag-line to be displayed next to each other rather than above and below each other while maintaining optimum accessibility.

```
#navigation, #seeAlso, #comments, #standards {  
    display: none;  
}
```

The above code is used in the print-only styles to basically 'turn-off' those elements, such as navigation, which are insignificant to the single page.

Note

display: none and **visibility: hidden** vary in that **display: none** takes the element completely out of play, where as **visibility: hidden** keeps the element and its flow in place without visually representing its contents. For example, if the second paragraph of 3 were set to **display: none**, the first paragraph would run straight in to the third whereas if it were set to **visibility: hidden**, there would be a gap where the paragraph should be.

Tables

Perhaps the best way to understand the table-related display property values is to think of HTML tables. **table** is the initial display and you can mimic the **tr** and **td** elements with the **table-row** and **table-cell** values respectively.

The **display** property goes further by offering **table-column**, **table-row-group**, **table-column-group**, **table-header-group**, **table-footer-group** and **table-caption** as values, which are all quite self-descriptive. The immediately obvious benefit of these values is that you can construct a table by columns, rather than the row-biased method used in HTML.

Finally, the value **inline-table** basically sets the table without line breaks before and after it.

Note

Getting carried away with CSS tables can seriously damage your accessibility. HTML should be used to convey meaning, so if you have tabular data it should be arranged in HTML tables. Using CSS tables exclusively could result in a mash of data that is completely unreadable without the CSS. Bad. And not in a Michael Jackson way.

Other display types

list-item is self descriptive and displays as in the way that you would usually expect an **li** HTML element to. To work properly then, elements displayed this way should be nested in a **ul** or **ol** element.

run-in makes an element either in-line or block-line depending on the display of its parent. It doesn't work on IE or Mozilla based browsers. Very helpful.

compact also makes the element inline or block-line depending on the context. It doesn't work that well either...

marker is used exclusively with the **:before** and **:after** pseudo elements to define the display of the value of the **content** property. The automatic display of the **content** property is already **marker**, so this is only useful if you are overriding a previous **display** property for the pseudo element.

Page Layout

Layout with CSS is easy. If you are used to laying out a page with tables, it may at first appear difficult, but it isn't, it's just different and actually makes much more sense.

You need to look at each part of the page as an individual chunk that you can shove wherever you choose. You can place these chunks absolutely or relative to another chunk.

Positioning

The **position** property is used to define whether an element is **absolute**, **relative**, **static** or **fixed**.

The value **static** is the default value for elements and renders the position in the normal order of things, as they appear in the HTML.

relative is much like **static**, but the element can be offset from its original position with the properties **top**, **right**, **bottom** and **left**.

absolute pulls an element out of the normal flow of the HTML and delivers it to a world all of its own. In this crazy little world, the absolute element can be placed anywhere on the page using **top**, **right**, **bottom** and **left**.

fixed behaves like **absolute**, but it will absolutely position an element in reference to the browser **window** as opposed to the web **page**, so, theoretically, fixed elements should stay exactly where they are on the

screen even when the page is scrolled. Why theoretically? Why else - this works great in browsers such as Mozilla and Opera, but in IE it doesn't work at all.

Layout using absolute positioning

You can create a traditional two-column layout with absolute positioning if you have something like the following HTML:

```
<div id="navigation">
  <ul>
    <li><a href="this.html">This</a></li>
    <li><a href="that.html">That</a></li>
    <li><a href="theOther.html">The Other</a></li>
  </ul>
</div>
<div id="content">
  <h1>Ra ra banjo banjo</h1>
  <p>Welcome to the Ra ra banjo banjo page. Ra ra banjo banjo.
  Ra ra banjo banjo. Ra ra banjo banjo. Ra ra banjo banjo.</p>
  <p>(Ra ra banjo banjo)</p>
</div>
```

And if you apply the following CSS:

```
#navigation {
  position: absolute;
  top: 0;left: 0;
  width: 10em;
}
#content {
  margin-left: 10em;
}
```

You will see that this will set the navigation bar to the left and set the width to 10 em's. Because the navigation is absolutely positioned, it has nothing to do with the flow of the rest of the page, so all that is needed is to set the left margin of the content area to be equal to the width of the navigation bar.

How bloody easy. And you aren't limited to this two-column approach. With clever positioning, you can arrange as many blocks as you like. If you wanted to add a third column, for example, you could add a 'navigation2' chunk to the HTML and change the CSS to:

```
#navigation {
  position: absolute;
  top: 0;
  left: 0;
  width: 10em;
}
```

```
#navigation2 {
    position: absolute;
    top: 0;
    right: 0;
    width: 10em;
}
#content {
    margin: 0 10em; /* setting top and bottom margin to 0 and
    right and left margin to 10em */
}
```

The only downside to absolutely positioned elements is that because they live in a world of their own, there is no way of accurately determining where they end. If you were to use the examples above and all of your pages had small navigation bars and large content areas, you would be okay, but, especially when using relative values for widths and sizes, you often have to abandon any hope of placing anything, such as a footer, below these elements. If you wanted to do such a thing, it would be necessary to float your chunks, rather than absolutely positioning them.

Floating

Floating an element will shift it to the right or left of a line, with surrounding content flowing around it.

Floating is normally used to position smaller elements within a page (in the original default CSS for this site, the 'Next Page' links in the HTML Beginner's Guide and CSS Beginner's Guide are floated right. See also the **:first-letter** example in Pseudo Elements), but it can also be used with bigger chunks, such as navigation columns.

Taking the HTML example below, you could apply the following CSS:

```
#navigation {
    float: left;
    width: 10em;
}
#navigation2 {
    float: right;
    width: 10em;
}
#content {
    margin: 0 10em;
}
```

If you do not want the next element to wrap around the floating objects, you can apply the clear property. `clear: left` will clear left floated elements, `clear: right` will clear right floated elements and `clear: both` will clear both left and right floated elements. So if, for example, you wanted a footer to your page, you could add a chunk of HTML with the id 'footer' and then add the following

CSS:

```
#footer {  
    clear: both;  
}
```

And there you have it. A footer that will appear underneath all columns, regardless of the length of any of them.

Note

This has been a general introduction to positioning and floating, with emphasis to the larger 'chunks' of a page, but remember, these methods can be applied to any element within those chunks too. With a combination of positioning, floating, margins, padding and borders, you should be able to represent ANY web design and there is nothing that can be done in tables that can not be done with CSS.

The ONLY reason for using tables for layout at all is if you are trying to accommodate ancient browsers. This is where CSS really shows its advantages - it results in a highly accessible page a fraction of the weight of an equivalent table-based page.

At-Rules

At-rules encapsulate a bunch of CSS rules and apply them specifically.

Importing

The **import** at-rule will bolt on another style sheet. For example, if you want to add the styles of another style sheet to your existing one, you would add something like:

```
@import url(addonstyles.css);
```

This is often used in place of the `<link>` tag to link a CSS file to an HTML page, by essentially having an internal style sheet that looks something like this:

```
<style type="text/css" media="all">@import  
url(monkey.css);</style>
```

The benefit of this is that older browsers, such as Netscape 4.x don't have a clue about at-rules and so won't link to the style-sheet, which, if you have well-structured markup, will leave functional (although un-styled) plain HTML.

Media types

The **media** at-rule will apply its contents to a specified media, such as print. For example:

```
@media print {
  body {
    font-size: 10pt;
    font-family: times new roman, times, serif;
  }
  #navigation {
    display: none;
  }
}
```

The media-type can be:

- _ **all** - for every media under, over, around and in the sun.
- _ **aural** - for speech synthesizers.
- _ **handheld** - for handheld devices.
- _ **print** - for printers.
- _ **projection** - for projectors.
- _ **screen** - for computer screens.

You can also use **braille**, **embossed**, **tty** or **tv**.

Note: having said all of that, the only media-types currently supported by IE are **all**, **screen** and **print**.

Character sets

The **charset** at-rule simply sets the character set encoding of an external stylesheet. It would appear at the top of the stylesheet and look something like **@charset "ISO-8859-1";**

Font faces

The **font-face** at-rule is used for a detailed description of a font and can embed an external font in your CSS.

It requires a **font-family** descriptor, which the font can be referenced to, the value of which can be an existing font name (so overwriting that font when conditions are met) or it can be a completely new name. To embed a font, the **src** descriptor is used. Other descriptors added to the **font-face** at-rule become conditions for that embedded font to be used, for example, if you were to add a **font-weight: bold** style to the at-rule, the **src** of the **font-family** will only be applied to a selector with the **font-family** property if the **font-weight** property is also set to **bold**.

You might use a font-face at-rule like this:

```
@font-face {
  font-family: somerandomfontname;
  src: url(somefont.eot);
  font-weight: bold;
}
p {
  font-family: somerandomfontname;
  font-weight: bold;
}
```

This will apply the somefont.eot font to paragraphs (it would not if the `p` selector was not set to `font-weight: bold`).

Note

Support for embedded fonts is patchy at best. Mozilla based browsers don't support them and have no immediate plans to do so. Only Internet Explorer seems to have any kind of decent support and this is by no means straightforward. To embed fonts with IE, you need to use Microsoft's WEFT software, which will convert the characters of a TrueType font into a condensed OpenType font (and this can then only be used on the URI that you specify). Because of this limited (and quite complicated) compatibility, it is best not to use fonts that do not have an adequate alternative system font.

Pages

The `page` at-rule is for **paged media** and is an advanced way to apply styles to printed media. It defines a **page block** that extends on the box model (see Margins and Padding) so that you can define the size and presentation of a single page.

There are a number of conventions that apply to `page` at-rules, such as there is no padding or border and this isn't a computer screen we're talking about - pixels and ems as units aren't allowed.

There are a number of specific properties that can be used, such as `size`, which can be set to `portrait`, `landscape`, `auto` or a length. The `marks` property can also be used to define crop marks.

```
@page {
  size: 15cm 20cm;
  margin: 3cm;marks: cross;
}
```

Pseudo classes for paged media

There are three pseudo classes that are used specifically in conjunction with the **page** at-rule, which would take the form of **@page :pseudo-class { stuff }**.

:first applies to the first page of the paged media.

:left and **:right** apply to left-facing and right-facing pages respectively. This might be used to specify a greater left margin on left-facing pages and a greater right margin on right-facing pages.

There are a number of other facets specific to the page at-rule such as page-breaks and named pages, but seeing as this at-rule works on hardly any browser, you've probably wasted enough time reading this part anyway. It's a nice enough idea though...

Pseudo Elements

Pseudo elements suck on to selectors much like pseudo classes, taking the form of **selector:pseudoelement { property: value; }**. There are four of the suckers.

First letters and First lines

The **first-letter** pseudo element applies to the first letter of an element and **first-line** to the top line of an element. You could, for examples create drop caps and a bold first-line for paragraphs like this:

```
p:first-letter {
    font-size: 3em;
    float: left;
}
p:first-line {
    font-weight: bold;
}
```

Before and after

The **before** and **after** pseudo elements are used in conjunction with the **content** property to place content either side of an element without touching the HTML.

The value of the **content** property can be **open-quote**, **close-quote**, **no-open-quote**, **no-close-quote**, any string enclosed in quotation marks or any image using **url(imagename)**.

```
blockquote:before {
    content: open-quote;
}
blockquote:after {
    content: close-quote;
}
li:before {
    content: "POW: "
}
p:before {
    content: url(images/jam.jpg)
}
```

Note

Sounds great, dunnit? Well, as with so many things (-sigh-), most users won't be able to see the before or after effects because IE just can't be bothered with them. Lazy lazy lazy.

Specificity

If you have two (or more) conflicting CSS rules that point to the same element, there are some basic rules that a browser follows to determine which one is most **specific** and therefore wins out.

It may not seem like something that important, and in most cases you won't come across any conflicts at all, but the larger and more complex your CSS files become, or the more CSS files you start to juggle with, the greater likelihood there is of conflicts turning up.

If the selectors are the same then the latest one will always take precedence. For example, if you had:

```
p { color: red; }
p { color: blue; }
```

p elements would be coloured blue because that rule came last.

However, you won't usually have identical selectors with conflicting declarations on purpose (because there's not much point). Conflicts quite legitimately come up, however, when you have nested selectors. In the following example:

```
div p { color: red; }
p { color: blue; }
```

It might seem that **p** elements *within a **div** element* would be coloured blue, seeing as a rule to colour **p** elements blue comes last, but they would actually be coloured red due to the specificity of the first selector. Basically, the more specific a selector, the more preference it will be given when it comes to conflicting styles.

The actual specificity of a group of nested selectors takes some calculating. Basically, you give every id selector ("#whatever") a value of 100, every class selector (".whatever") a value of 10 and every HTML selector ("whatever") a value of 1. Then you add them all up and hey presto, you have the specificity value.

- _ **p** has a specificity of 1 (1 HTML selector)
- _ **div p** has a specificity of 2 (2 HTML selectors; 1+1)
- _ **.tree** has a specificity of 10 (1 class selector)
- _ **div p.tree** has a specificity of 12 (2 HTML selectors and a class selector; 1+1+10)
- _ **#baobab** has a specificity of 100 (1 id selector)
- _ **body #content .alternative p** has a specificity of 112 (HTML selector, id selector, class selector, HTML selector; 1+100+10+1)

So if all of these examples were used, **div p.tree** (with a specificity of 12) would win out over **div p** (with a specificity of 2) and **body #content .alternative p** would win out over all of them, *regardless of the order*.