SMASHING MAGAZINE

CSS Essentials

Imprint

Version 1: June 2012

ISBN: 978-3-943075-37-3

Cover Design: Ricardo Gimenes

PR & Press: Stephan Poppe

eBook Strategy: Thomas Burkert

Technical Editing: Talita Telma Stöckle, Andrew Rogerson

Idea & Concept: Smashing Media GmbH

ABOUT SMASHING MAGAZINE

Smashing Magazine is an online magazine dedicated to Web designers and developers worldwide. Its rigorous quality control and thorough editorial work has gathered a devoted community exceeding half a million subscribers, followers and fans. Each and every published article is carefully prepared, edited, reviewed and curated according to the high quality standards set in Smashing Magazine's own publishing policy. Smashing Magazine publishes articles on a daily basis with topics ranging from business, visual design, typography, front-end as well as back-end development, all the way to usability and user experience design. The magazine is — and always has been — a professional and independent online publication neither controlled nor influenced by any third parties, delivering content in the best interest of its readers. These guidelines are continually revised and updated to assure that the quality of the published content is never compromised.

ABOUT SMASHING MEDIA GMBH

Smashing Media GmbH is one of the world's leading online publishing companies in the field of Web design. Founded in 2009 by Sven Lennartz and Vitaly Friedman, the company's headquarters is situated in southern Germany, in the sunny city of Freiburg im Breisgau. Smashing Media's lead publication, Smashing Magazine, has gained worldwide attention since its emergence back in 2006, and is supported by the vast, global Smashing community and readership. Smashing Magazine had proven to be a trustworthy online source containing high quality articles on progressive design and coding techniques as well as recent developments in the Web design industry.

About this eBook

When developers push aside CSS to concentrate on JavaScript performance, they might be overlooking some great applications of CSS. This eBook CSS Essentials explores some practical implementations of CSS, including usage of pseudo-elements in CSS, decoupling HTML from CSS, Modern CSS layouts with equal height columns, taming CSS selectors, among others. These techniques will help improve both the performance and maintainability of your Web pages in various browsers.

Table of Contents

Backgrounds In CSS: Everything You Need To Know

Michael Martin

Backgrounds are a core part of CSS. They are one of the fundamentals that you simply need to know. In this article, we will cover the basics of using CSS backgrounds, including properties such as **`background-attachment.`** We'll show some common tricks that can be done with the background as well as what's in store for backgrounds in CSS 3 (including four new background properties!).

Working With Backgrounds in CSS 2

OVERVIEW

We have five main background properties to use in CSS 2. They are as follows:

- **`background-color:`** specifies the solid color to fill the background with.
- **`background-image:`** calls an image for the background.
- **`background-position:`** specifies where to place the image in the element's background.
- **`background-repeat:`** determines whether the image is tiled.

- **`background-attachment:`** determines whether the image scrolls with the page.

These properties can all be merged into one short-hand property: **`background.`** One important thing to note is that the background accounts for the contents of the element, including the padding and border. It does not include an element's margin. This works as it should in Firefox, Safari and Opera, and now in IE8. But in IE7 and IE6 the background does not include the border, as illustrated below.

Background does not extend to the borders in IE7 and IE6.

BASIC PROPERTIES

Background color

The **`background-color`** property fills the background with a solid color. There are a number of ways to specify the color. The follow commands all have the same output:

```
background-color: blue;
background-color: rgb(0, 0, 255);
background-color: #0000ff;
```

The **`background-color`** property can also be set to **`transparent`**, which makes any elements underneath it visible instead.

Background image

The **`background-image`** property allows you to specify an image to be displayed in the background. This can be used in conjunction with **`background-color`**, so if your image is not tiled, then any space that the image doesn't cover will be set to the background color. Again, the code is very simple. Just remember that the path is relative to the style sheet. So, in the following snippet, the image is in the same directory as the style sheet:

```
background-image: url(image.jpg);
```

But if the image was in a sub-folder named *images*, then it would be:

```
background-image: url(images/image.jpg);
```

Background repeat

By default, when you set an image, the image is repeated both horizontally and vertically until the entire element is filled. This may be what you want, but sometimes you want an image to be displayed only once or to be tiled in only one direction. The possible values (and their results) are as follows:

```
background-repeat: repeat;     /* The default value. Will tile
the image in both directions. */
```

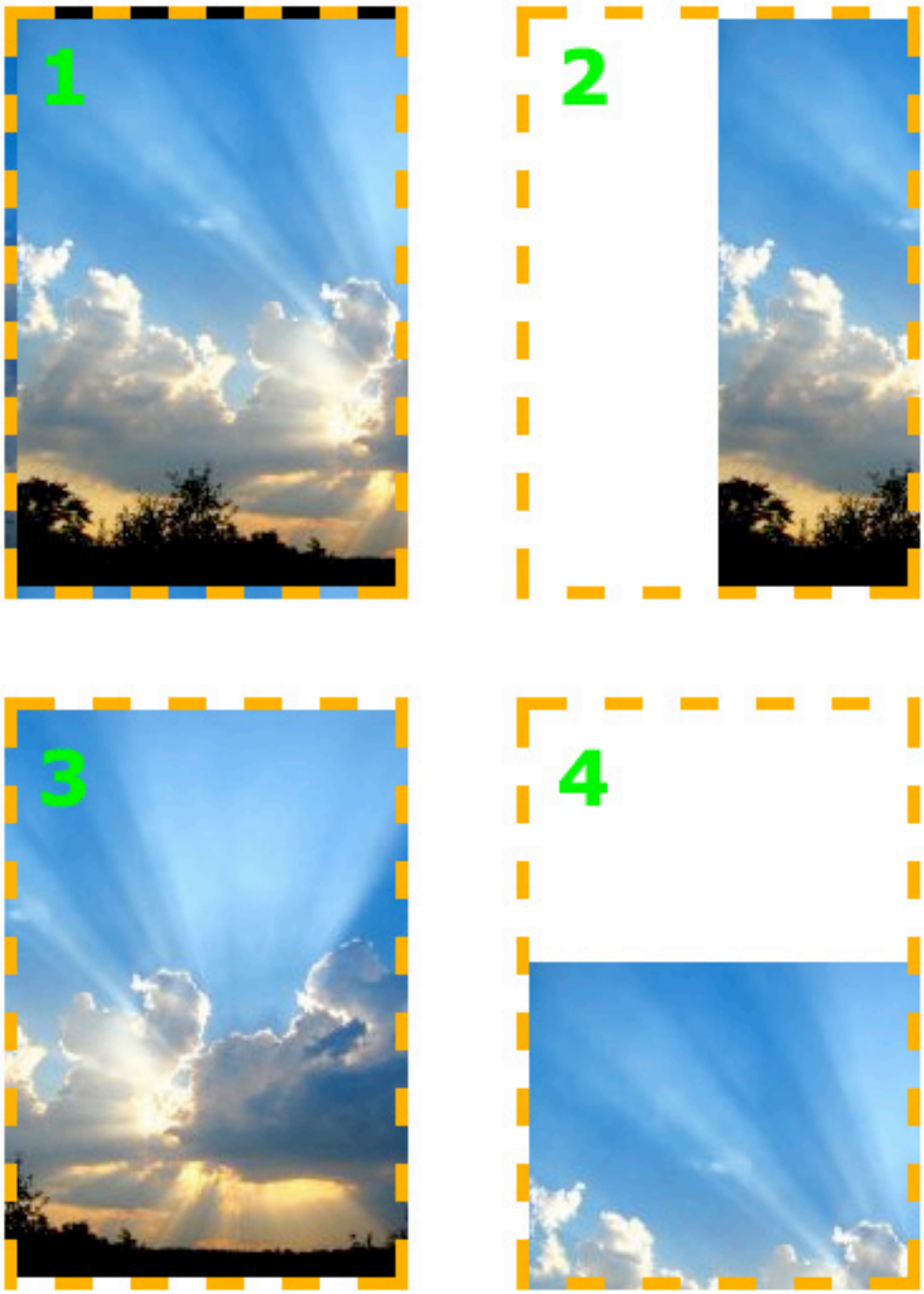
```
background-repeat: no-repeat;  /* No tiling. The image will be
used only once. */
background-repeat: repeat-x;   /* Tiles horizontally (i.e.
along the x-axis) */
background-repeat: repeat-y;   /* Tiles vertically (i.e. along
the y-axis) */
background-repeat: inherit;    /* Uses the same background-
repeat property of the element's parent. */
```

Background position

The **`background-position`** property controls where a background image is located in an element. The trick with **`background-position`** is that you are actually specifying where the top-left corner of the image will be positioned, relative to the top-left corner of the element.

In the examples below, we have set a background image and are using the **`background-position`** property to control it. We have also set **`background-repeat`** to **`no-repeat`**. The measurements are all in pixels. The first digit is the x-axis position (horizontal) and the second is the y-axis position (vertical).

```
/* Example 1: default. */
background-position: 0 0;  /* i.e. Top-left corner of element.
*/
/* Example 2: move the image to the right. */
background-position: 75px 0;
/* Example 3: move the image to the left. */
background-position: -75px 0;
/* Example 4: move the image down. */
background-position: 0 100px;
```



The image can be set to any position you like.

The **`background-position`** property also works with other values, keywords and percentages, which can be useful, especially when an element's size is not set in pixels.

The keywords are self-explanatory. For the x-axis:

- left
- center
- right

And for the y-axis:

- top
- center
- bottom

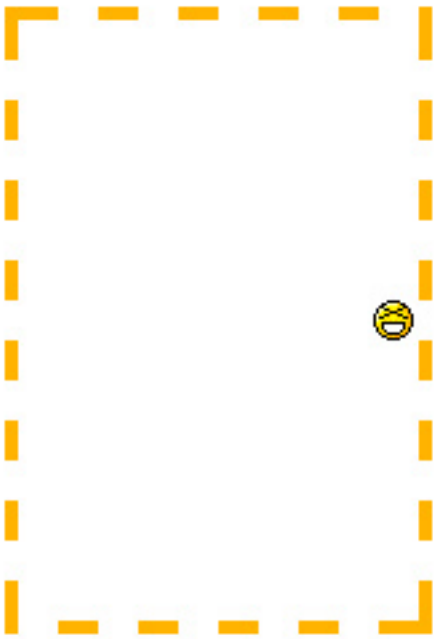
Their order is exactly like that of the pixels values, x-axis value first, and y-axis value second, as so:

```
background-position: top right;
```

Percentage values are similar. The thing to remember here is that when you specify a percentage, the browser sets the part of the image at that percentage to align with that percentage of the element. This makes more sense in an example. Let's say you specify the following:

```
background-position: 100% 50%;
```

This goes 100% of the way across the image (i.e. the very right-hand edge) and 100% of the way across the element (remember, the starting point is always the top-left corner), and the two line up there. It then goes 50% of the way down the image and 50% of the way down the element to line up there. The result is that the image is aligned to the right of the element and exactly half-way down it.



The smiley face is aligned as it would be if it was set to ***`right center`****.*

Background attachment

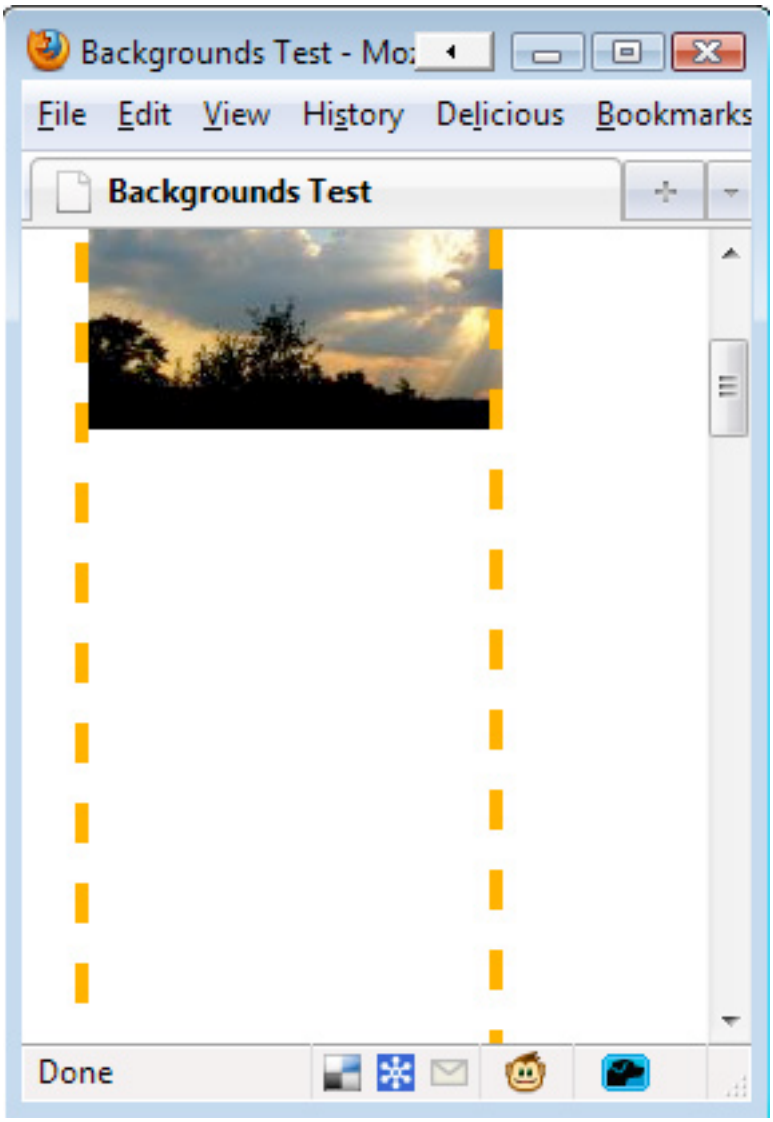
The **`background-attachment`** property determines what happens to an image when the user scrolls the page. The three available properties are **`scroll`**, **`fixed`** and **`inherit`**. **`Inherit`** simply tells the element to follow the **`background-attachment`** property of its parent.

To understand **`background-attachment`** properly, we need to first think of how the page and view port interact. The view port is the section of your browser that displays the Web page (think of it like your browser but without all the toolbars). The view port is set in its position and never moves.

When you scroll down a Web page, the view port does not move. Instead, the content of the page scrolls upward. This makes it seem as if the view port is scrolling down the page. Now, when we set background-attachment:scroll;, we are telling the background that when the element scrolls, the background must scroll with it. In simpler terms, the background sticks to the element. This is the default setting for **`background-attachment`**.

Let's see an example to make it clearer:

```
background-image: url(test-image.jpg);
background-position: 0 0;
background-repeat: no-repeat;
background-attachment: scroll;
```

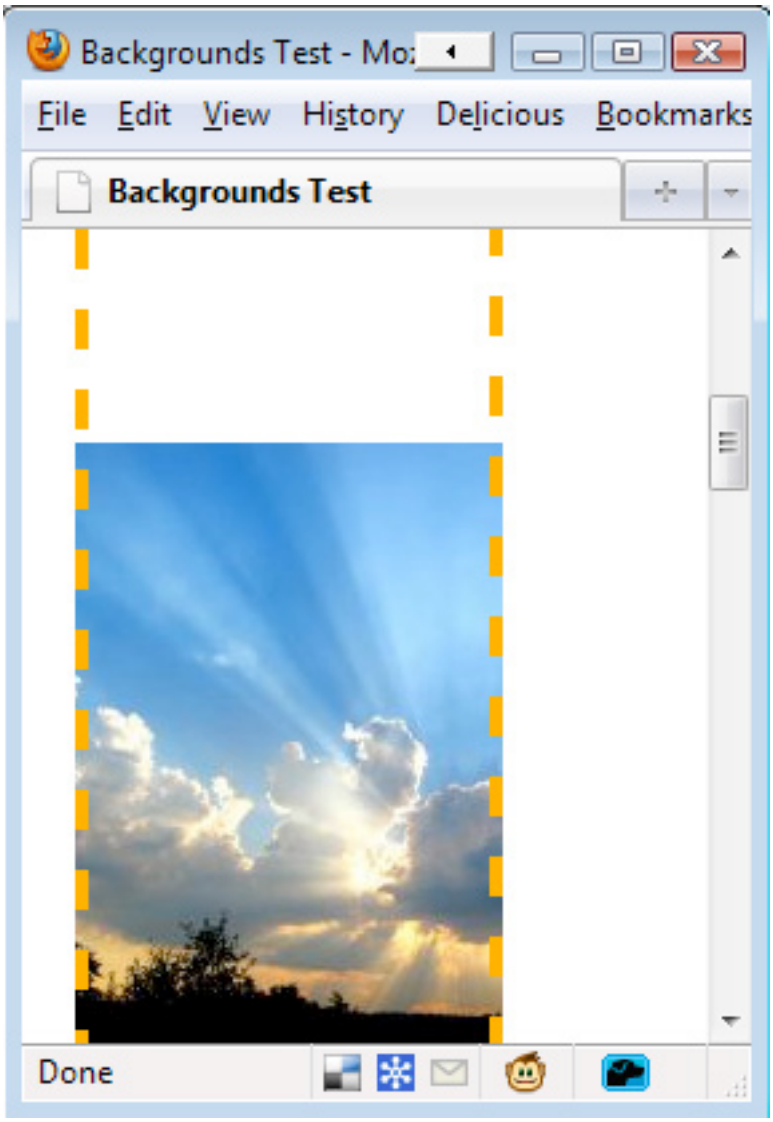


As we scroll down the page, the background scrolls up until it disappears.

But when we set the **`background-attachment`** to **`fixed`**, we are telling the browser that when the user scrolls down the page, the background should stay fixed where it is — i.e. not scroll with the content.

Let's illustrate this with another example:

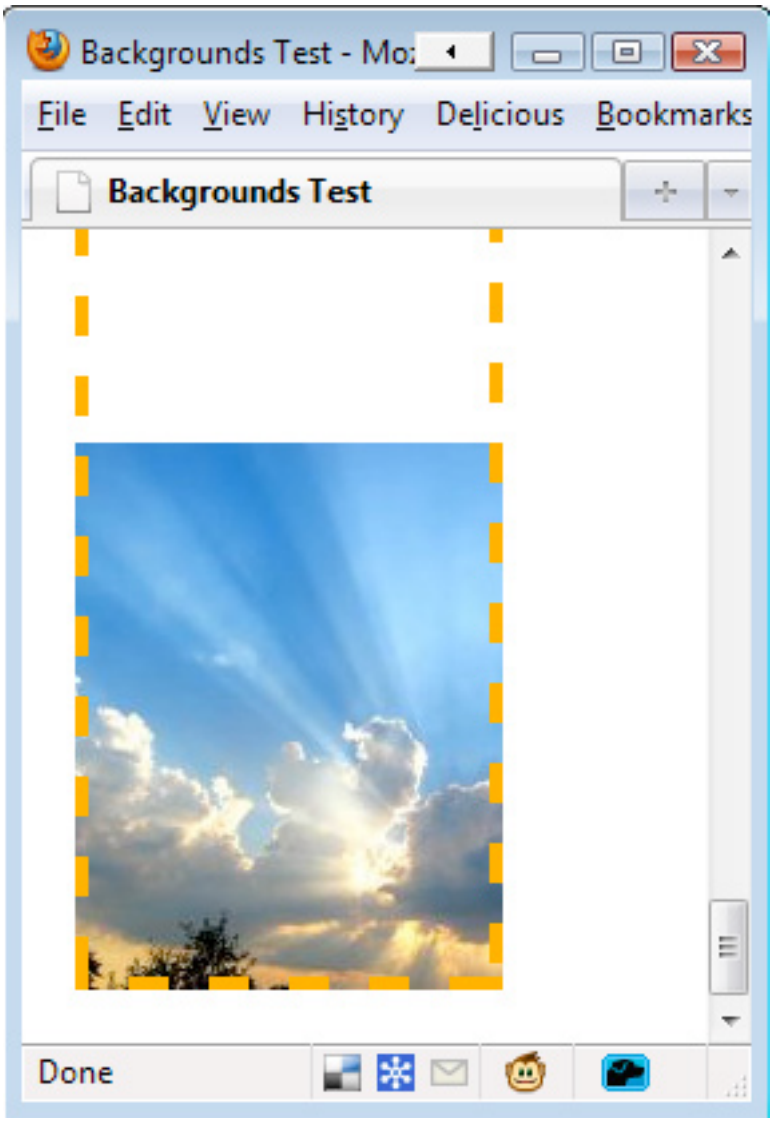
```
background-image: url(test-image.jpg);
background-position: 0 100%;
background-repeat: no-repeat;
background-attachment: fixed;
```



We have scrolled down the page here, but the image remains visible.

The important thing to note however is that the background image only appears in areas where its parent element reaches. Even though the image is positioned relative to the view port, it will not appear if it's parent element is not visible. This can be shown with another example. In this one, we have aligned the image to the bottom-left of the view port. But only the area of the image inside the element is visible.

```
background-image: url(test-image.jpg);
background-position: 0 100%;
background-repeat: no-repeat;
background-attachment: fixed;
```



Part of the image has been cut off because it begins outside of the element.

The Background Shorthand Property

Instead of writing out all of these rules each time, we can combine them into a single rule. It takes the following format:

```
background: <color> <image> <position> <attachment> <repeat>
```

For example, the following rules...

```
background-color: transparent;
background-image: url(image.jpg);
background-position: 50% 0 ;
background-attachment: scroll;
background-repeat: repeat-y;
```

... could be combined into the following single line:

```
background: transparent url(image.jpg) 50% 0 scroll repeat-y;
```

And you don't have to specify every value. If you leave a property out, its default value is used instead. For example, the line above has the same output as:

```
background: url(image.jpg) 50% 0 repeat-y;
```

Common Uses Of Backgrounds

Aside from their obvious use of making elements more attractive, backgrounds can be used for other purposes.

FAUX COLUMNS

When using the CSS property of `float` to position layout elements, ensuring that two (or more) columns are the same length can be difficult. If the lengths are different, then the background of one of the columns will be shorter than the background of the other, spoiling your design.

Faux columns is a very simple background trick that was first written up on A List Apart. The idea is simple: instead of applying a separate background for each column, apply one background image to the parent element of the columns with that image containing the designs for each column.

TEXT REPLACEMENT

Our choice of fonts on the Web is very limited. We could get custom fonts by using tools such as sIFR, but they require users to have JavaScript enabled. An easier solution that works on any browser is to create an image of the text in the font you want and use it as the background instead. This way, the text still appears in the markup for search engines and screen readers, but browsers will show your preferred font.

For example, your HTML markup could look like this:

```
<h3 class="blogroll">Blogroll</h3>
```

If we have a 200 by 75 pixel image of this text in a nicer font, then we could display that instead using the CSS as follows:

```
h3.blogroll {
  width: 200px;
  height: 75px;  /* So that the element will show the whole
image. */
  background:url(blogroll-text.jpg) 0 0 no-repeat;    /* Sets
the background image */
  text-indent: -9999px; /* Hides the regular text by moving it
9999 pixels to the left */
  }
```

EASIER BULLET POINTS

Bullet in an unordered list can look clunky. Instead of dealing with all of the different list-style properties, we can simply hide the bullets and replace them with a background image. Because the image can be anything you like, the bullets will look much nicer.

Here, we turn a regular unordered list into one with sleek bullets:

```
ul {
  list-style: none; /* Removes default bullets. */
  }

ul li {
  padding-left: 40px; /* Indents list items, leaving room for
background image on the left. */
  background: url(bulletpoint.jpg) 0 0 no-repeat;
  }
```

Backgrounds in CSS 3

CSS 3 has many changes in store for backgrounds. The most obvious is the option for multiple background images, but it also comes with four new properties as well as adjustments to current properties.

MULTIPLE BACKGROUND IMAGES

In CSS 3, you will be able to use more than one image for the background of an element. The code is the same as CSS 2, except you would separate images with a comma. The first declared image is positioned at the top of the element, with subsequent images “layered” beneath it, like so:

```
background-image: url(top-image.jpg), url(middle-image.jpg),
url(bottom-image.jpg);
```

NEW PROPERTY: BACKGROUND CLIP

This brings us back to the issue discussed at the beginning of this article, about backgrounds being shown beneath the border. This is described as the “background painting area.”

The **`background-clip`** property was created to give you more control over where backgrounds are displayed. The possible values are:

- **`background-clip: border-box;`**
  backgrounds displayed beneath the border.
- **`background-clip: padding-box;`**
  backgrounds displayed beneath the padding, not the border.
- **`background-clip: content-box;`**
  backgrounds displayed only beneath content, not border or padding.
- **`background-clip: no-clip;`**
  the default value, same as border-box.

NEW PROPERTY: BACKGROUND ORIGIN

This is used in conjunction with **`background-position.`** You can have the **`background-position`** calculated from the border, padding or content boxes (like **`background-clip`**).

- **`background-origin: border-box;`**
  background-position is calculated from the border.
- **`background-origin: padding-box;`**
  background-position is calculated from the padding box.
- **`background-origin: content-box;`**
  background-position is calculated from the content.

A good explanation of the differences between **`background-clip`** and **`background-origin`** is available on CSS3.info.

NEW PROPERTY: BACKGROUND SIZE

The **`background-size`** property is used to resize your background image. There are a number of possible values:

- **`background-size: contain;`**
  scales down image to fit element (maintaining pixel aspect ratio).
- **`background-size: cover;`**
  expands image to fill element (maintaining pixel aspect ratio).
- **`background-size: 100px 100px;`**
  scales image to the sizes indicated.
- **`background-size: 50% 100%;`**
  scales image to the sizes indicated. Percentages are relative to the size of containing element.

You can read up on some examples of special cases on the CSS 3 specifications website.

NEW PROPERTY: BACKGROUND BREAK

In CSS 3, elements can be split into separate boxes (e.g. to make an inline element span across several lines). The **`background-break`** property controls how the background is shown across the different boxes.

The possible values are:

- **`Background-break: continuous;`**: the default value. Treats the boxes as if no space is between them (i.e. as if they are all in one box, with the background image applied to that).
- **`Background-break: bounding-box;`**: takes the space between boxes into account.
- **`Background-break: each-box;`**: treats each box in the element separately and re-draws the background individually for each one.

CHANGES TO BACKGROUND COLOR

The **`background-color`** property has a slight enhancement in CSS 3. In addition to specifying the background color, you can specify a "fall-back color" that is used if the bottom-layer background image of the element cannot be used.

This is done by adding a forward slash before the fall-back color, like so:

```
background-color: green / blue;
```

In this example, the background color would be green. However, if the bottom-most background image cannot be used, then blue would be shown instead of green. If you do not specify a color before the slash, then it is assumed to be transparent.

CHANGES TO BACKGROUND REPEAT

When an image is repeated in CSS 2, it is often cut off at the end of the element. CSS 3 introduces two new properties to fix this:

- **`space`**: an equal amount of space is applied between the image tiles until they fill the element.
- **`round`**: the image is scaled down until the tiles fit the element.

An example of **`background-repeat: space;`** is available on the CSS 3 specifications website.

CHANGES TO BACKGROUND ATTACHMENT

The **`background-attachment`** has a new possible value: **`local.`** This comes into play with elements that can scroll (i.e. are set to **`overflow: scroll;`**). When **`background-attachment`** is set to **`scroll,`** the background image will not scroll when the element's content is scrolled.

With **`background-attachment:local`** now, the background scrolls when the element's content is scrolled.

Conclusion

To sum up, there is a lot to know about backgrounds in CSS. But once you wrap your head around it, the techniques and naming conventions all make good sense and it quickly becomes second nature.

If you're new to CSS, just keep practicing and you'll get the hang of backgrounds fast enough. If you're a seasoned pro, I hope you're looking forward to CSS 3 as much as I am!

The Mystery Of The CSS Float Property

Louis Lazaris

Years ago, when developers first started to make the transition to HTML layouts without tables, one CSS property that suddenly took on a very important role was the **`float`** property. The reason that the **`float`** property became so common was that, by default, block-level elements will not line up beside one another in a column-based format. Since columns are necessary in virtually every CSS layout, this property started to get used — and even overused — prolifically.

The CSS **`float`** property allows a developer to incorporate table-like columns in an HTML layout without the use of tables. If it were not for the CSS **`float`** property, CSS layouts would not be possible except using absolute and relative positioning — which would be messy and would make the layout unmaintainable.

In this chapter, we'll discuss exactly what the **`float`** property is and how it affects elements in particular contexts. We'll also take a look at some of the differences that can occur in connection with this property in the most commonly-used browsers. Finally, we'll showcase a few practical uses for the CSS **`float`** property. This should provide a well-rounded and thorough discussion of this property and its impact on CSS development.

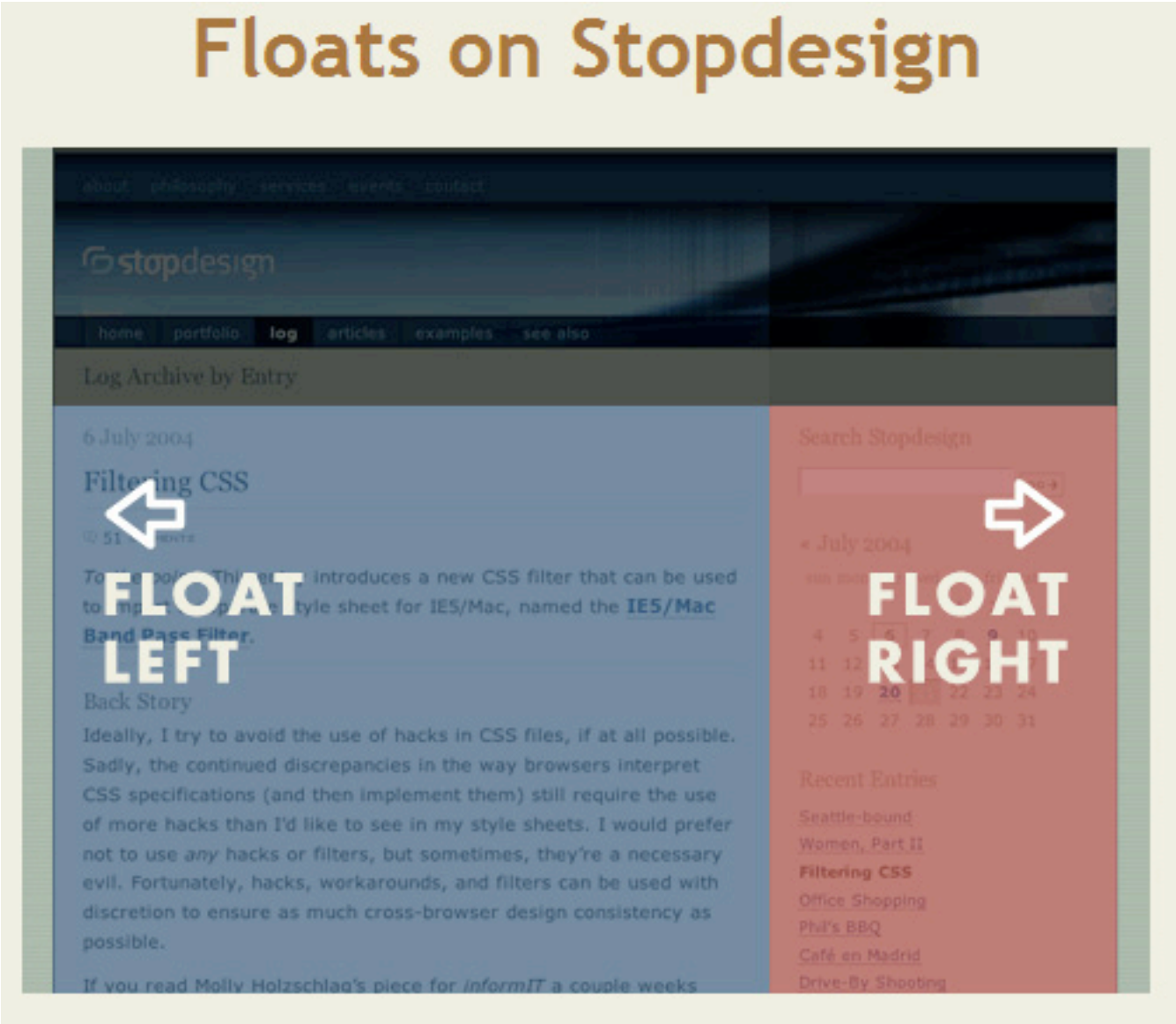
Definition and Syntax

The purpose of the CSS **`float`** property is, generally speaking, to push a block-level element to the left or right, taking it out of the flow in relation to other block elements. This allows naturally-flowing content to wrap around the floated element. This concept is similar to what you see every day in print literature, where photos and other graphic elements are aligned to one side while other content (usually text) flows naturally around the left- or right-aligned element.

Flickr photo by presentday

The photo above shows the “Readers’ sites” section of .net magazine, which displays 3 readers’ photos each aligned left in their respective columns with text wrapping around the aligned images. That is the basic idea behind the **`float`** property in CSS layouts, and demonstrates one of the ways it has been used in table-less design.

The effectiveness of using floats in multi-columned layouts was explained by Douglas Bowman in 2004 in his classic presentation No More Tables:



No More Tables

Bowman explained the principles behind table-less design, using Microsoft's old site as a case study. Floats were used prominently in his mock overhaul of the Microsoft layout.

SYNTAX

The **`float`** CSS property can accept one of 4 values: **`left`**, **`right`**, **`none`**, and **`inherit`**. It is declared as shown in the code below.

```
#sidebar {
  float: left;
}
```

The most commonly-used values are **`left`** and **`right`**. A value of **`none`** is the default, or initial **`float`** value for any element in an HTML page. The value **`inherit`**, which can be applied to nearly any CSS property, does not work in Internet Explorer versions up to and including 7.

The **`float`** property does not require the application of any other property on a CSS element for **`float`** to function correctly, however, **`float`** will work more effectively under specific circumstances.

Generally, a floated element should have an explicitly set width (unless it is a replaced element, like an image). This ensures that the float behaves as expected and helps to avoid issues in certain browsers.

```
#sidebar {
  float: left;
  width: 350px;
}
```

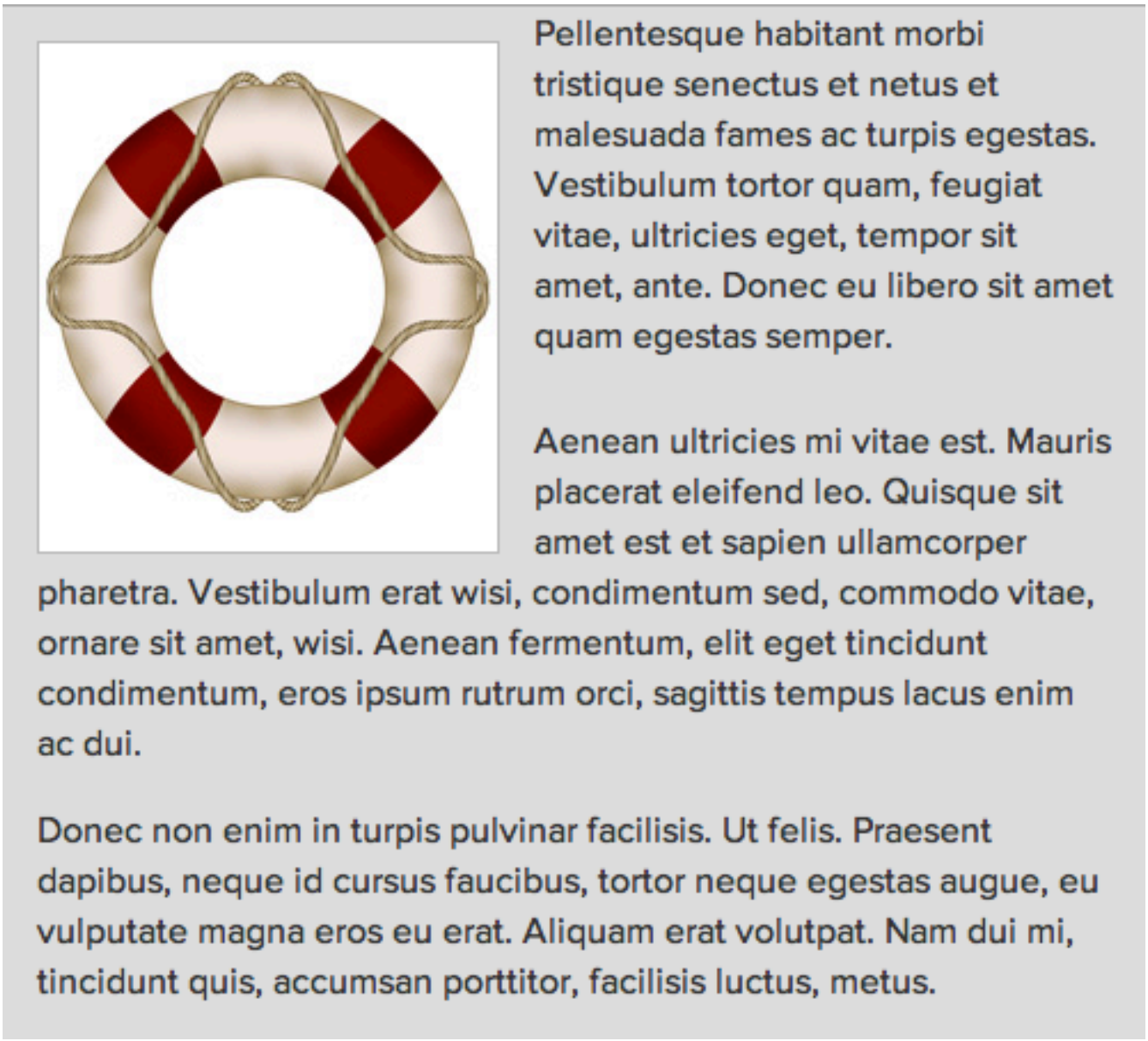
SPECIFICS ON FLOATED ELEMENTS

Following is a list of exact behaviors of floated elements according to CSS2 Specifications:

- A left-floated box will shift to the left until its leftmost margin edge (or border edge if margins are absent) touches either the edge of the containing block, or the edge of another floated box
- If the size of the floated box exceeds the available horizontal space, the floated box will be shifted down
- Non-positioned, non-floated, block-level elements act as if the floated element is not there, since the floated element is out of flow in relation to other block elements
- Margins of floated boxes do not collapse with margins of adjacent boxes
- The root element (**`<html>`**) cannot be floated
- An inline element that is floated is converted to a block-level element

Float in Practice

One of the most common uses for the **`float`** property is to float an image with text wrapping it. Here is an example:



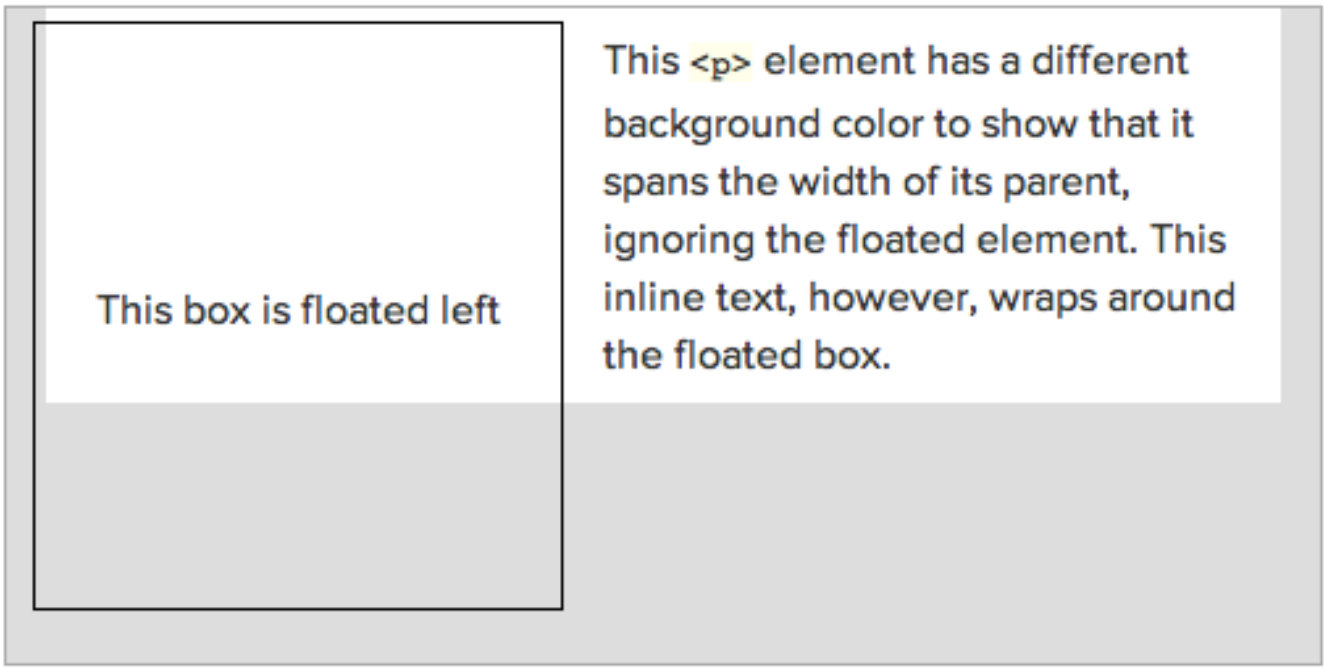
Lifesaver Image from stock.xchng

The CSS applied to the image in the box above is as follows:

```
img {
  float: left;
  margin: 0 15px 5px 0;
  border: solid 1px #bbb;
}
```

The only property required to make this effect work is the **`float`** property. The other properties (margin and border) are there for aesthetic reasons. The other elements inside the box (the **<p>** tags with text inside them) do not need any styles applied to them.

As mentioned earlier, floated elements are taken out of flow in relation to other block elements, and so other block elements remain in flow, acting as if the floated element is not even there. This is demonstrated visually below:



In the above example, the **<p>** element is a block-level element, so it ignores the floated element, spanning the width of the container (minus padding). All non-floated, block-level elements will behave in like manner.

The text in the paragraph is inline, so it flows around the floated element. The floated box is also given margin settings to offset it from the paragraph, making it visually clear that the paragraph element is ignoring the floated box.

Clearing Floats

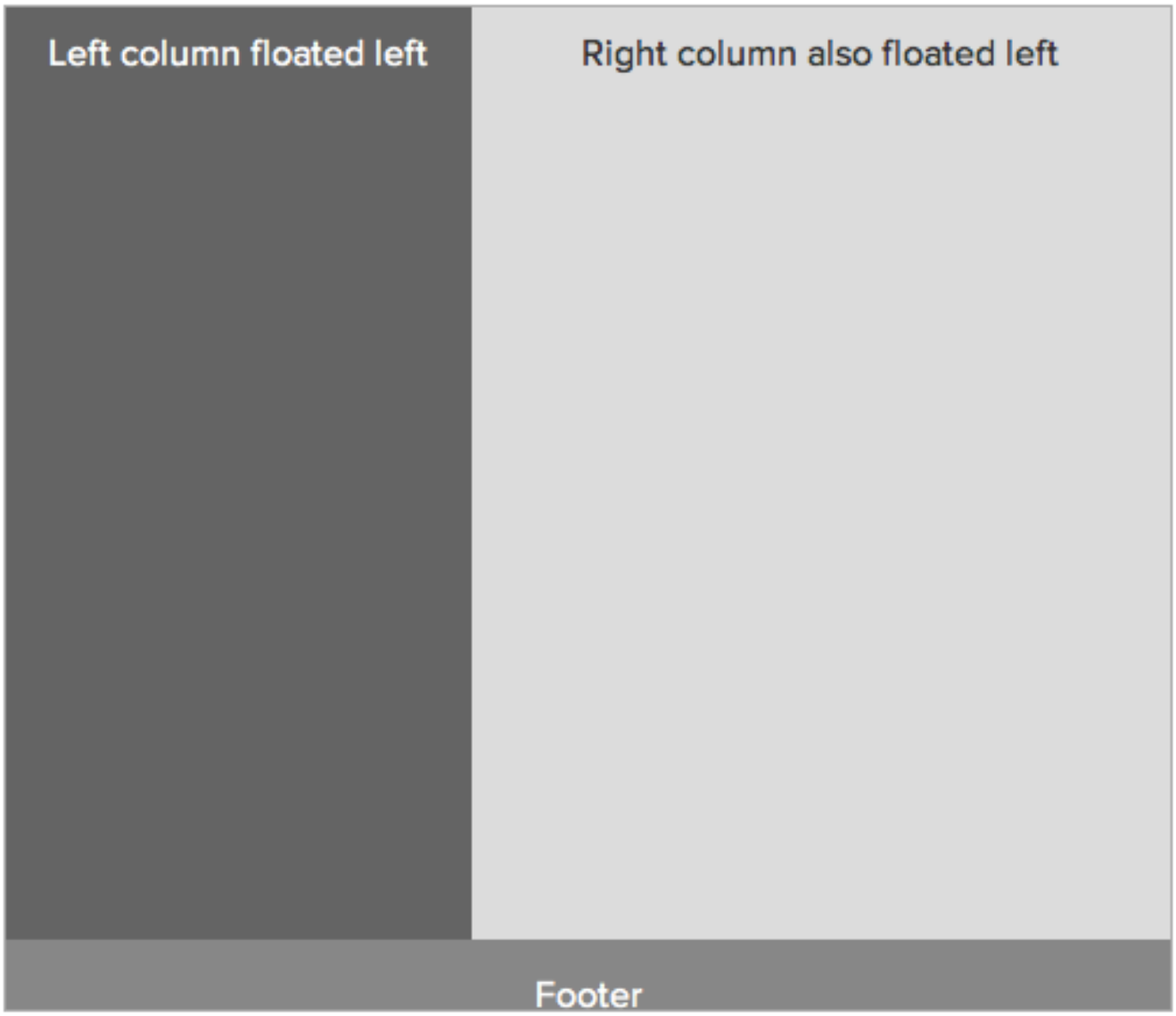
Layout issues with floats are commonly fixed using the CSS **`clear`** property, which lets you "clear" floated elements from the left or right side, or both sides, of an element.

Let's take a look at an example that commonly occurs — a footer wrapping to the right side of a 2-column layout:

If you view this page in IE6 or IE7, you won't see any problems. The left and right columns are in place, and the footer is nicely tucked underneath. But in Firefox, Opera, Safari, and Chrome, you'll see the footer jump up beside the left column. This is caused by the float applied to the columns. This is the correct behavior, even though it is a more problematic one. To resolve this issue, we use the aforementioned **`clear`** property, applied to the footer:

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

The result is shown below:

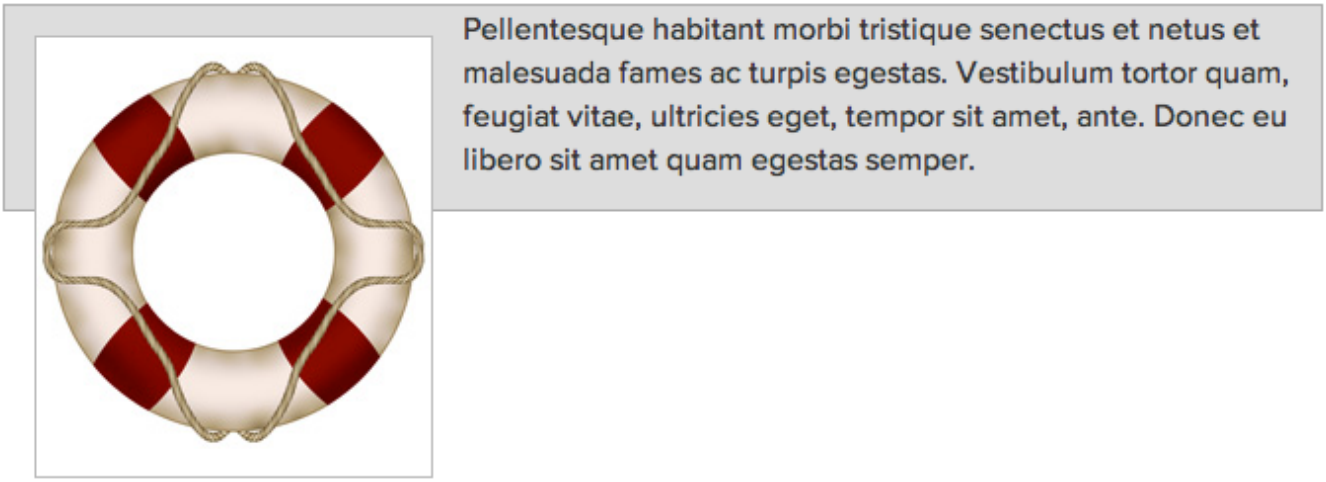


The **`clear`** property will clear only floated elements, so applying **`clear: both`** to both columns would not cause the footer to drop down, because the footer is not a floated element.

So use **`clear`** on non-floated elements, and even occasionally on floated elements, to force page elements to occupy their intended space.

Fixing the Collapsed Parent

One of the most common symptoms of float-heavy layouts is the “collapsing parent”. This is demonstrated in the example below:



Notice that the bottom of the floated image appears outside its parent. The parent does not fully expand to hold the floated image. This is caused by what we discussed earlier: the floated element is out of the flow in relation to other block elements, so all block elements will render as if the floated element is not even there. This is not a CSS bug; it's in line with CSS specifications. All browsers render the same in this example. It should be pointed out that, in this example, adding a width to the container prevents the issue from occurring in IE, so this would normally be something you would have to resolve in Firefox, Opera, Safari, or Chrome.

SOLUTION 1: FLOAT THE CONTAINER

The easiest way to fix this problem is to float the containing parent element:

Now the container expands to fit all the child elements. But unfortunately this fix will only work in a limited number of circumstances, since floating the parent may have undesirable effects on your layout.

SOLUTION 2: ADDING EXTRA MARKUP

This is an outdated method, but is an easy option. Simply add an extra element at the bottom of the container and "clear" it. Here's how the HTML would look after this method is implemented:

```
<div id="container">
<img src="http://media.smashingmagazine.com/wp-content/
uploads/2009/10/lifesaver.jpg" width="200" height="222"
alt="" />
<p>Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Vestibulum tortor quam,
feugiat vitae, ultricies eget, tempor sit amet, ante. Donec eu
libero sit amet quam egestas semper.</p>
  <div class="clearfix"></div>
</div>
```

And the resulting CSS applied to the new element:

```
.clearfix {
  clear: both;
}
```

You could also do this by means of a **
** tag with an inline style. In any case, you will have the desired result: the parent container will expand to enclose all of its children. But this method is not recommended since it adds nonsemantic code to your markup.

SOLUTION 3: THE :AFTER PSEUDO-ELEMENT

The **:after** pseudo-element adds an element to the rendered HTML page. This method has been used quite extensively to resolve float-clearing issues. Here is how the CSS looks:

```
.clearfix:after {
    content: ".";
    display: block;
    height: 0;
    clear: both;
    visibility: hidden;
}
```

The CSS class “clearfix” is applied to any container that has floating children and does not expand to enclose them.

But this method does not work in Internet Explorer up to version 7, so an IE-only style needs to be set with one of the following rules:

```
.clearfix {
  display: inline-block;
}
.clearfix {
  zoom: 1;
}
```

Depending on the type of issue you're dealing with, one of the above two solutions will resolve the issue in Internet Explorer. It should be noted that the **`zoom`** property is a non-standard Microsoft proprietary property, and will cause your CSS to become invalid.

So, because the **`:after`** pseudo-element solution does not work in IE6/7, is a little bit bloated code-wise, and requires additional invalid IE-only styles, this solution is not the best method, but is probably the best we've considered so far.

SOLUTION 4: THE OVERFLOW PROPERTY

By far the best, and easiest solution to resolve the collapsing parent issue is to add **`overflow: hidden`** or **`overflow: auto`** to the parent element. It's clean, easy to maintain, works in almost all browsers, and does not add extra markup.

You'll notice I said "almost" all browsers. This is because it does not work in IE6. But, in many cases, the parent container will have a set width, which fixes the issue in IE6. If the parent container does not have a width, you can add an IE6-only style with the following code:

```
// This fix is for IE6 only
.clearfix {
  height: 1%;
  overflow: visible;
}
```

In IE6, the **`height`** property is incorrectly treated as **`min-height`**, so this forces the container to enclose its children. The **`overflow`** property is then set back to “visible”, to ensure the content is not hidden or scrolled.

The only drawback to using the **`overflow`** method (in any browser) is the possibility that the containing element will have scrollbars or hide content. If there are any elements with negative margins or absolute positioning inside the parent, they will be obscured if they go beyond the parent’s borders, so this method should be used carefully. It should also be noted that if the containing element has to have a specified **`height`**, or **`min-height`**, then you would definitely not be able to use the **`overflow`** method.

So, there really is no simple, cross-browser solution to the collapsing parent issue. But almost any float clearing issue can be resolved with one of the above methods.

Float-Related Bugs in Internet Explorer

Over the years, there have been numerous articles published online that discuss common bugs in connection with floats in CSS layouts. All of these, not surprisingly, deal with problems specific to Internet Explorer.

Changing the Float Property with JavaScript

To change a CSS value in JavaScript, you would access the **`style`** object, converting the intended CSS property to "camel case". For example, the CSS property "margin-left" becomes **`marginLeft`**; the property **`background-color`** becomes **`backgroundColor`**, and so on. But with the **`float`** property, it's different, because **`float`** is already a reserved word in JavaScript. So, the following would be incorrect:

```
myDiv.style.float = "left";
```

Instead, you would use one of the following:

```
// For Internet Explorer
myDiv.style.styleFloat = "left";
// For all other browsers
myDiv.style.cssFloat = "left";
```

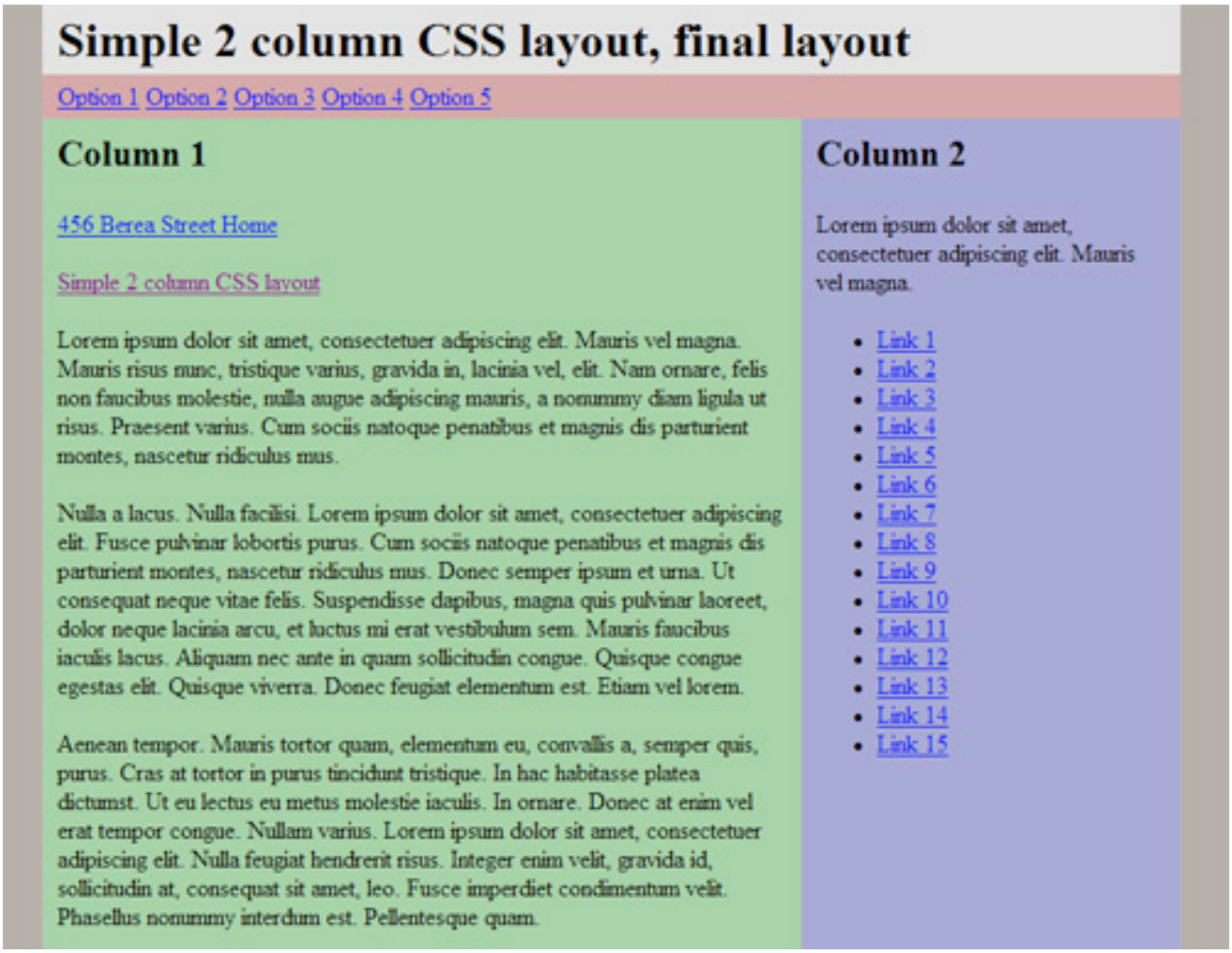
Practical Uses for Float

Floats can be used to resolve a number of design challenges in CSS layouts. Some examples are discussed here.

2-COLUMN, FIXED-WIDTH LAYOUT

Roger Johansson of 456 Berea Street outlines an 8-step tutorial to create a simple, cross-browser, 2-column, horizontally centered layout. The `float` property is integral to the chemistry of this layout.

> *"The layout consists of a header, a horizontal navigation bar, a main content column, a sidebar, and a footer. It is also horizontally centered in the browser window. A pretty basic layout, and not at all difficult to create with CSS once you know how to deal with the inevitable Internet Explorer bugs."*



Simple 2 column CSS layout

3-COLUMN, EQUAL-HEIGHT LAYOUT

Petr Stanicek of pixy.cz demonstrates a cross-browser 3-column layout, again using **`float`**:

> *"No tables, no absolute positioning (no positioning at all), no hacks(!), same height of all columns. The left and right columns have fixed width (150px), the middle one is elastic."*

3-Column Layout with CSS

FLOATED IMAGE WITH CAPTION

Similar to what we discussed earlier under “Float in Practice”, Max Design describes how to float an image with a caption, allowing text to wrap around it naturally.

Floating an Image and Caption

HORIZONTAL NAVIGATION WITH UNORDERED LISTS

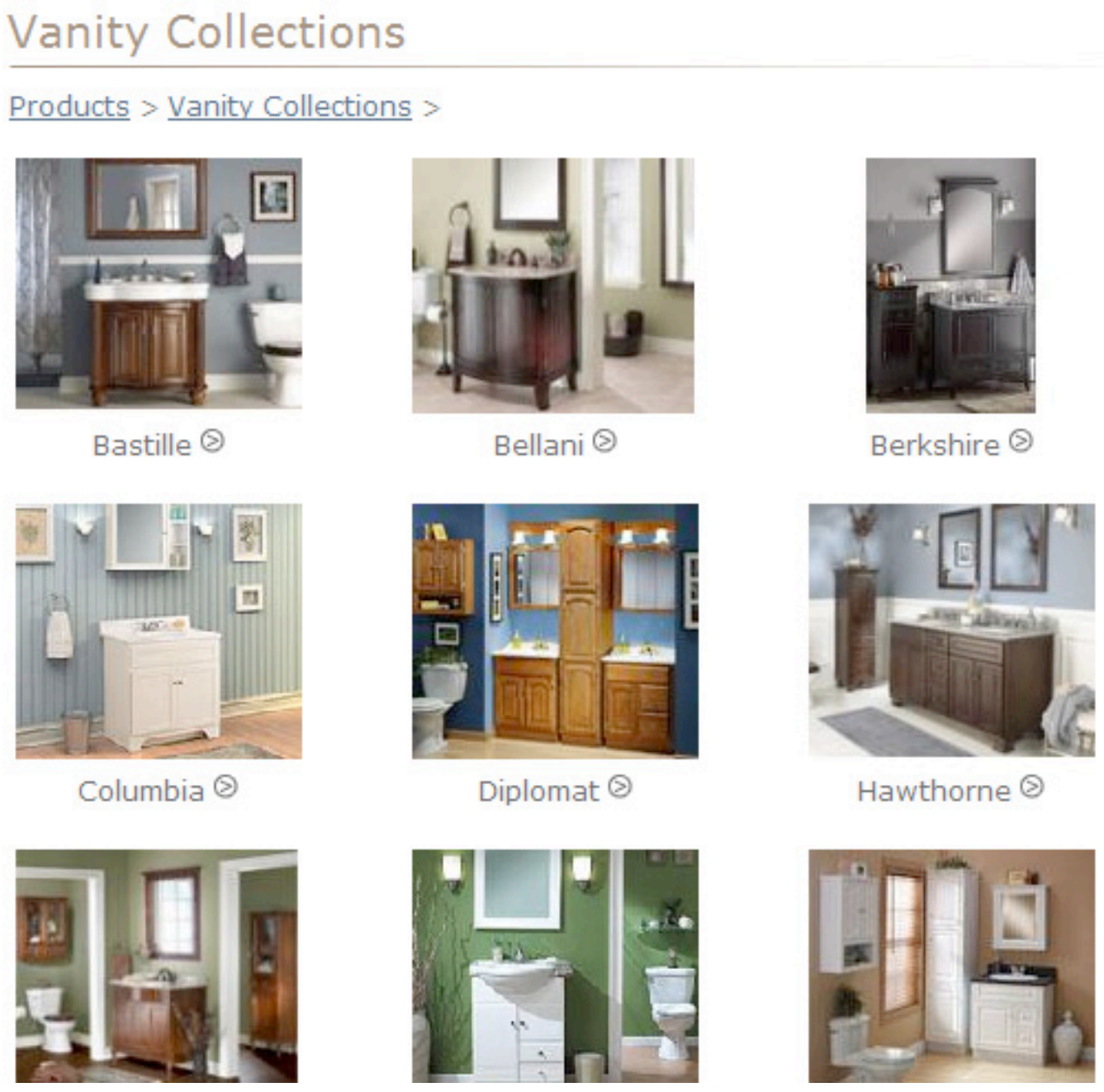
The **`float`** property is a key ingredient in coding sprite-based horizontal navigation bars. Chris Spooner of Line25 describes how to create a Menu of Awesomeness, in which the **`<li>`** elements that hold the navigation buttons are floated left:

How to Create a CSS Menu Using Image Sprites

To demonstrate the importance of the **`float`** property in this example, here is a screen shot of the same image after using firebug to remove the **`float: left`**:

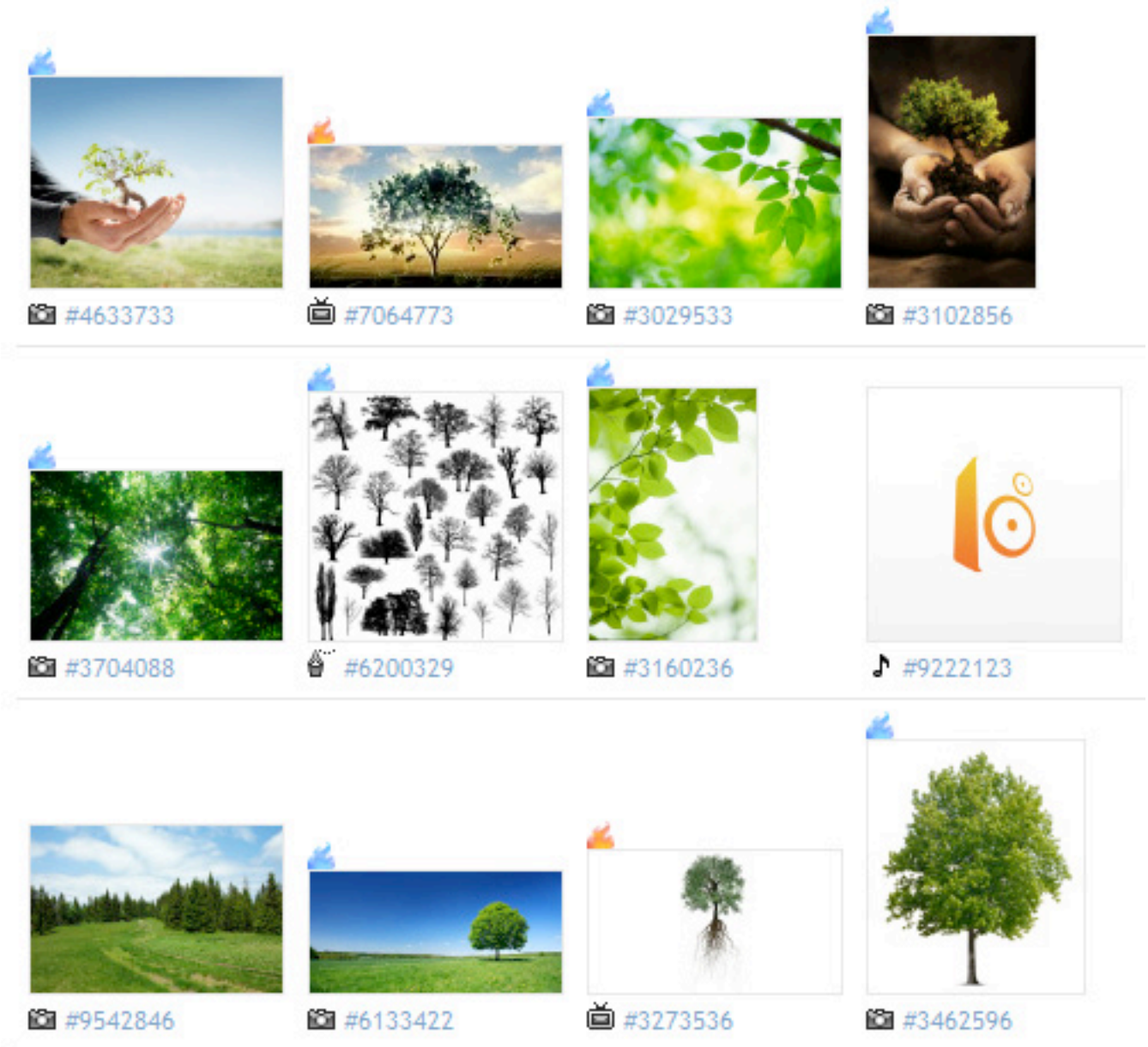
GRID-BASED PHOTO GALLERIES

A simple use for the **`float`** property is to left-float a series of photos contained in an unordered list, which gets the same result as what you would see in a table-based layout.



Foremost Canada's product page (above) displays their products in grid-based format, next to a left-column sidebar. The photos are displayed in an unordered list with the float property for all **`<li>`** elements set to **`float: left`**. This works better than a table-based grid, because the number of photos in the gallery can change and the layout would not be affected.

Paragon Furniture's futons page (above) is another example of a product page using an unordered list with floated list items.



iStockphoto's search results page (above) is a similarly-structured grid of photos, but this time the photos are contained in left-floated ** ** elements, instead of **<li>** elements.

ALIGNING AN <INPUT> FIELD WITH A BUTTON

Default styling on form elements across different browsers can be a pain to deal with. Often times, in a single-field form like a search form, it is necessary to place the **<input>** element next to the submit button. Here is a simple search form, with an image used for the submit button:

In every browser, the result is the same: The button appears slightly higher than the input field. Changing margins and padding does nothing. The simple way to fix this is to float the input field left, and add a small right margin. Here is the result:

Conclusion

As was mentioned at the outset, without the CSS **float** property, table-less layouts would be, at worst, impossible, and, at best, unmaintainable. Floats will continue to be prominent in CSS layouts, even as CSS3 begins to gain prominence — even though there have been a few discussions about layouts without the use of floats.

Hopefully this discussion has simplified some of the mysteries related to floats, and provided some practical solutions to a number of issues faced by CSS developers today.

The Z-Index CSS Property: A Comprehensive Look

Louis Lazaris

Most CSS properties are quite simple to deal with. Often, applying a CSS property to an element in your markup will have instant results — as soon as you refresh the page, the value set for the property takes effect, and you see the result immediately. Other CSS properties, however, are a little more complex and will only work under a given set of circumstances.

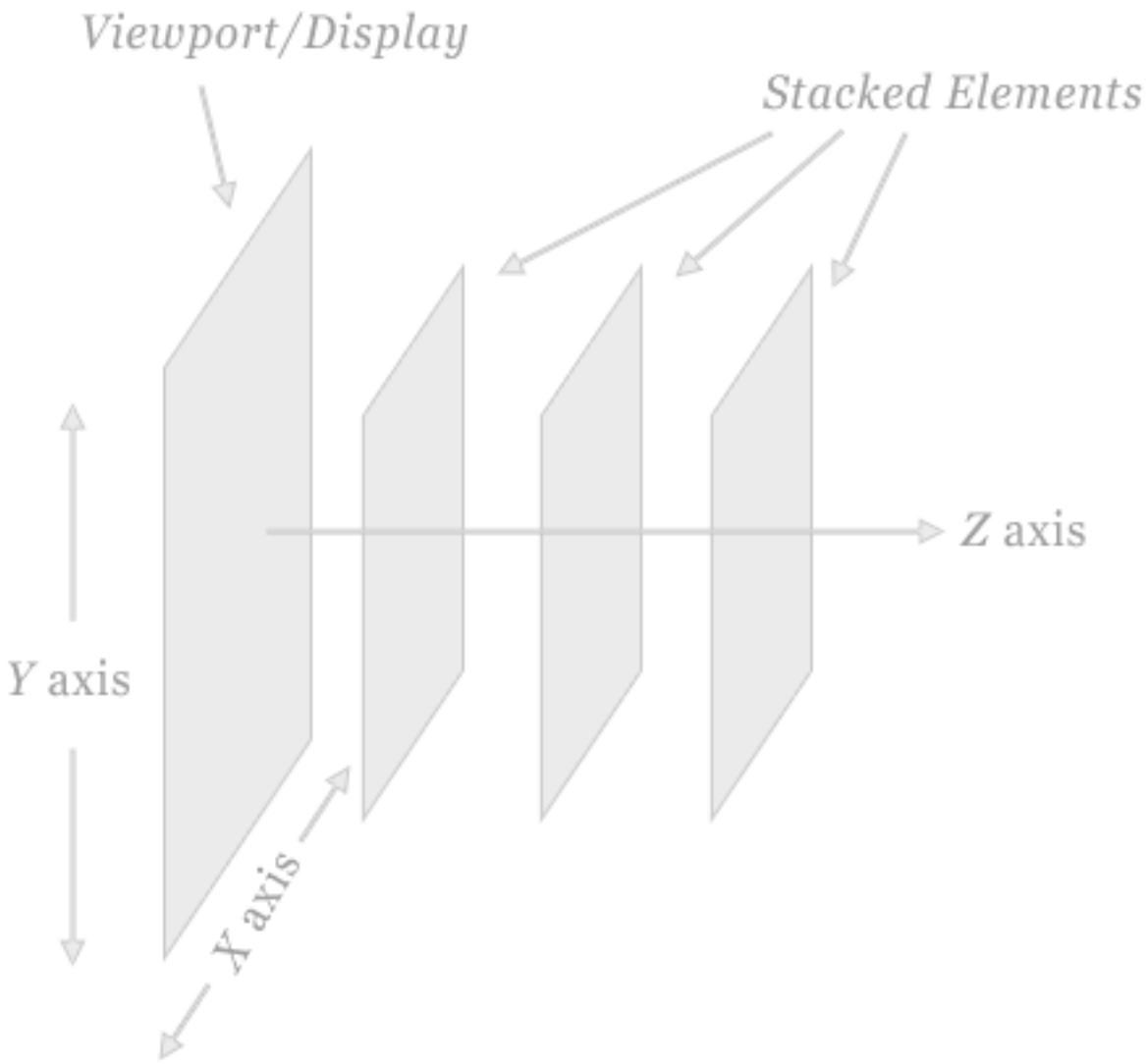
The **`z-index`** property belongs to the latter group. **`z-index`** has undoubtedly caused as much confusion and frustration as any other CSS property. Ironically, however, when **`z-index`** is fully understood, it is a very easy property to use, and offers an effective method for overcoming many layout challenges.

In this article, we'll explain exactly what **`z-index`** is, how it has been misunderstood, and we'll discuss some practical uses for it. We'll also describe some of the browser differences that can occur, particularly in previous versions of Internet Explorer and Firefox. This comprehensive look at **`z-index`** should provide developers with an excellent foundation to be able to use this property confidently and effectively.

What is it?

The **`z-index`** property determines the stack level of an HTML element. The "stack level" refers to the element's position on the *Z axis* (as opposed to the *X axis* or *Y* axis). A higher **`z-index`** value means the element will be closer to the top of the stacking order. This stacking order runs perpendicular to the display, or viewport.

3-DIMENSIONAL REPRESENTATION OF THE *Z* AXIS:



In order to clearly demonstrate how `z-index` works, the image above exaggerates the display of stacked elements in relation to the viewport.

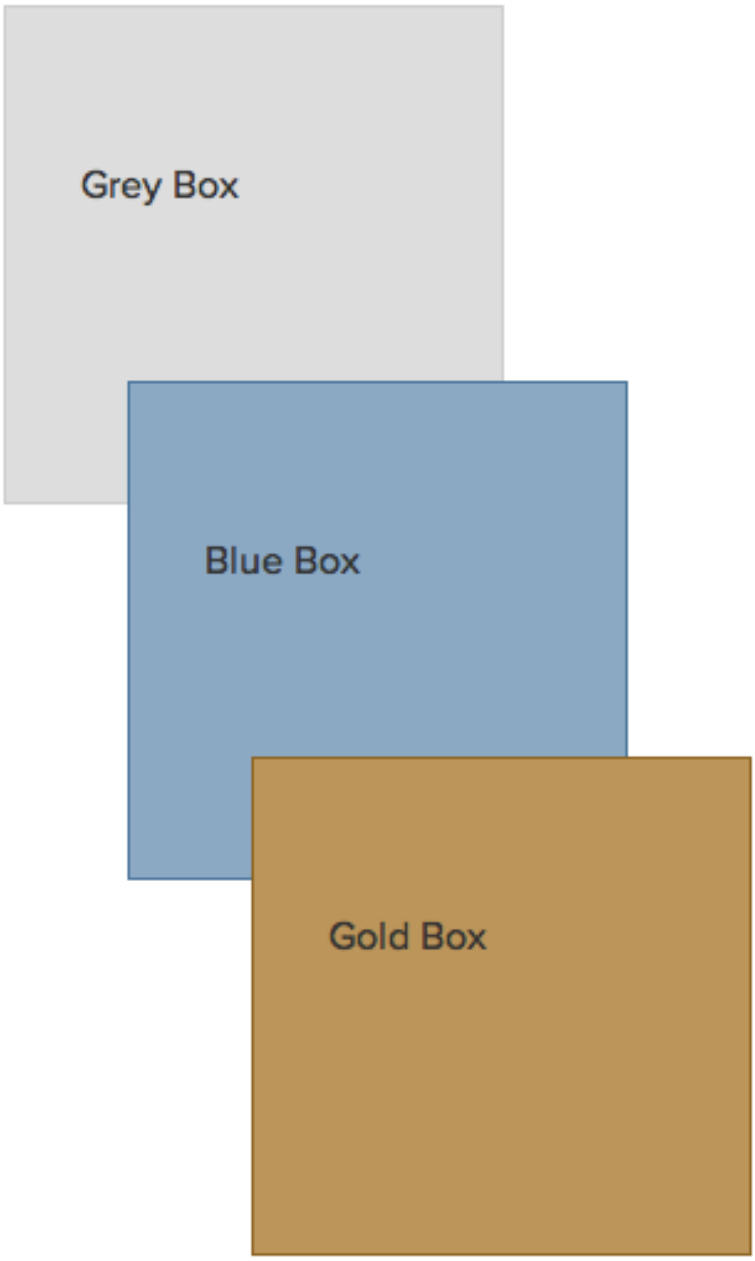
The Natural Stacking Order

In an HTML page, the natural stacking order (i.e. the order of elements on the *Z* axis) is determined by a number of factors. Below is a list showing the order that items fit into a stacking context, starting with the bottom of the stack. This list assumes none of the items has **`z-index`** applied:

- Background and borders of the element that establish stacking context
- Elements with negative stacking contexts, in order of appearance
- Non-positioned, non-floated, block-level elements, in order of appearance
- Non-positioned, floated elements, in order of appearance
- Inline elements, in order of appearance
- Positioned elements, in order of appearance

The **`z-index`** property, when applied correctly, can change this natural stacking order.

Of course, the stacking order of elements is not evident unless elements are positioned to overlap one another. Thus, to see the natural stacking order, negative margins can be used as shown below:

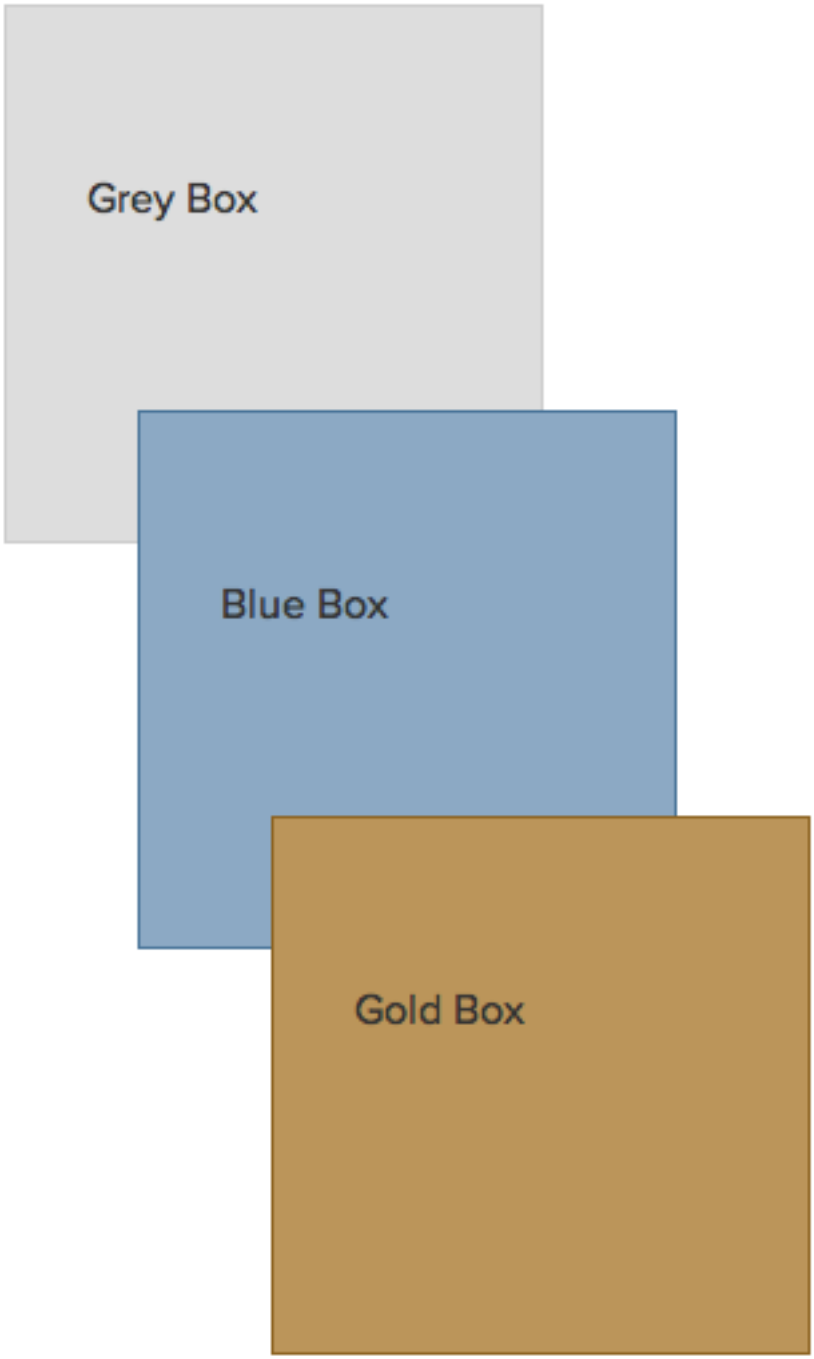


The boxes above are given different background and border colors, and the last two are indented and given negative top margins so you can see the natural stacking order. The grey box appears first in the markup, the blue box second, and the gold box third. The applied negative margins clearly demonstrate this fact. These elements do not have **`z-index`** values set; their stacking order is the natural, or default, order. The overlaps that occur are due to the negative margins.

Why Does it Cause Confusion?

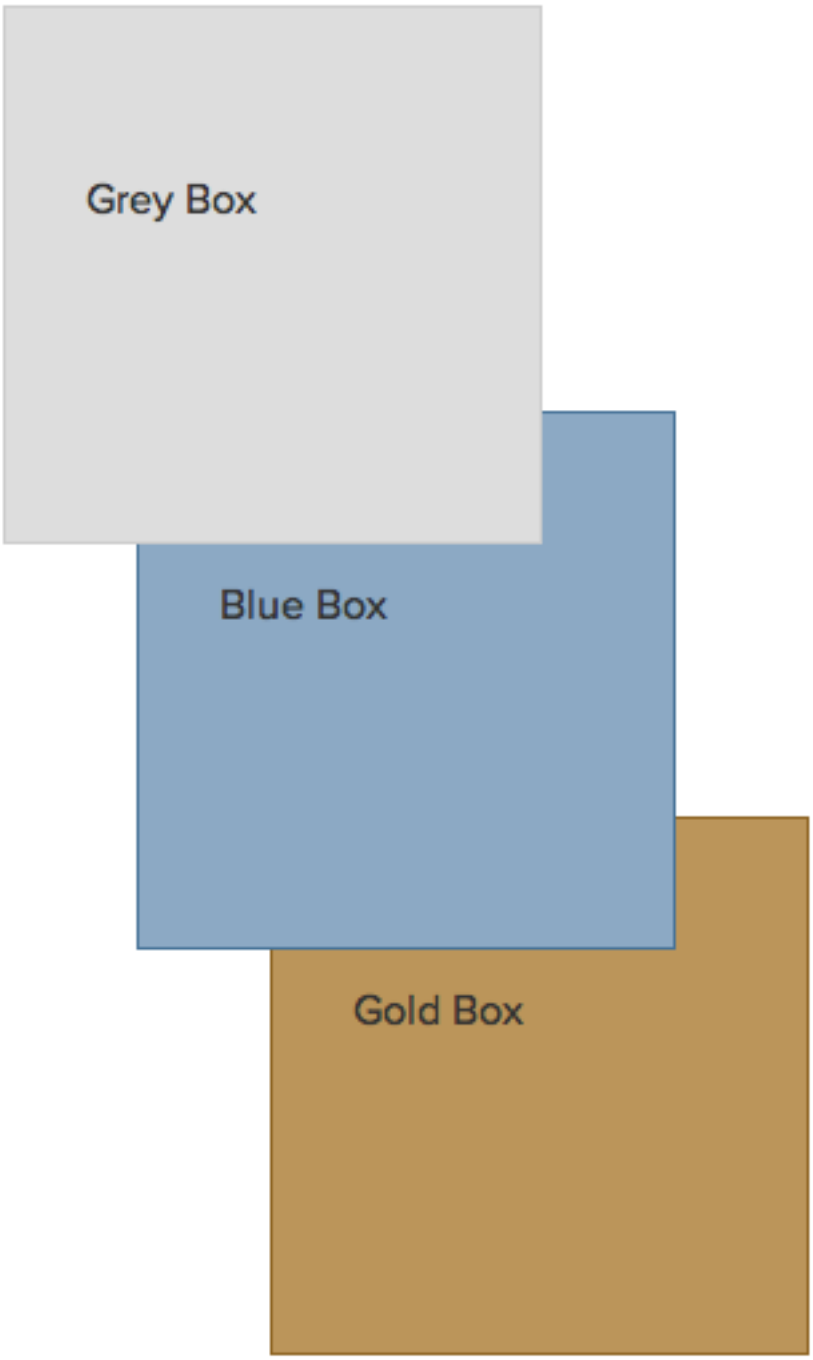
Although **`z-index`** is not a difficult property to understand, due to false assumptions it can cause confusion for beginning developers. This confusion occurs because **`z-index` will only work on an element whose `position`** property has been explicitly set to **`absolute`, `fixed`**, or **`relative`**.

To demonstrate that **`z-index`** only works on positioned elements, here are the same three boxes with **`z-index`** values applied to attempt to reverse their stacking order:



The grey box has a **`z-index`** value of “9999″; the blue box has a **`z-index`** value of “500″; and the gold box has a **`z-index`** value of “1″. Logically, you would assume that the stacking order of these boxes should now be reversed. But that is not the case, because none of these elements has the **`position`** property set.

Here are the same boxes with **`position: relative`** added to each, and their **`z-index`** values maintained:



Now the result is as expected: The stacking order of the elements is reversed; the grey box overlaps the blue box, and the blue box overlaps the gold.

Syntax

The **z-index** property can affect the stack order of both block-level and inline elements, and is declared by a positive or negative integer value, or a value of **auto**. A value of **auto** gives the element the same stack order as its parent.

Here is the CSS code for the third example above, where the **z-index** property is applied correctly:

```
#grey_box {
  width: 200px;
  height: 200px;
  border: solid 1px #ccc;
  background: #ddd;
  position: relative;
  z-index: 9999;
}
#blue_box {
  width: 200px;
  height: 200px;
  border: solid 1px #4a7497;
  background: #8daac3;
  position: relative;
  z-index: 500;
}
#gold_box {
  width: 200px;
  height: 200px;
  border: solid 1px #8b6125;
```

```
    background: #ba945d;
    position: relative;
    z-index: 1;
  }
```

Again, it cannot be stressed enough, especially for beginning CSS developers, that the **`z-index`** property will not work unless it is applied to a positioned element.

JavaScript Usage

If you want to affect the **`z-index`** value of an element dynamically via JavaScript, the syntax is similar to how most other CSS properties are accessed, using “camel casing” to replace hyphenated CSS properties, as in the code shown below:

```
var myElement = document.getElementById("gold_box");
myElement.style.position = "relative";
myElement.style.zIndex = "9999";
```

In the above code, the CSS syntax “z-index” becomes “zIndex”. Similarly, “background-color” becomes “backgroundColor”, “font-weight” becomes “fontWeight”, and so on.

Also, the position property is changed using the above code to again emphasize that **`z-index`** only works on elements that are positioned.

Improper Implementations in IE and Firefox

Under certain circumstances, there are some small inconsistencies in Internet Explorer versions 6 and 7 and Firefox version 2 with regards to the implementation of the **`z-index`** property.

`<SELECT>` ELEMENTS IN IE6

In Internet Explorer 6, the **`<select>`** element is a windowed control, and so will always appear at the top of the stacking order regardless of natural stack order, position values, or **`z-index`**. This problem is illustrated in the screen capture below:

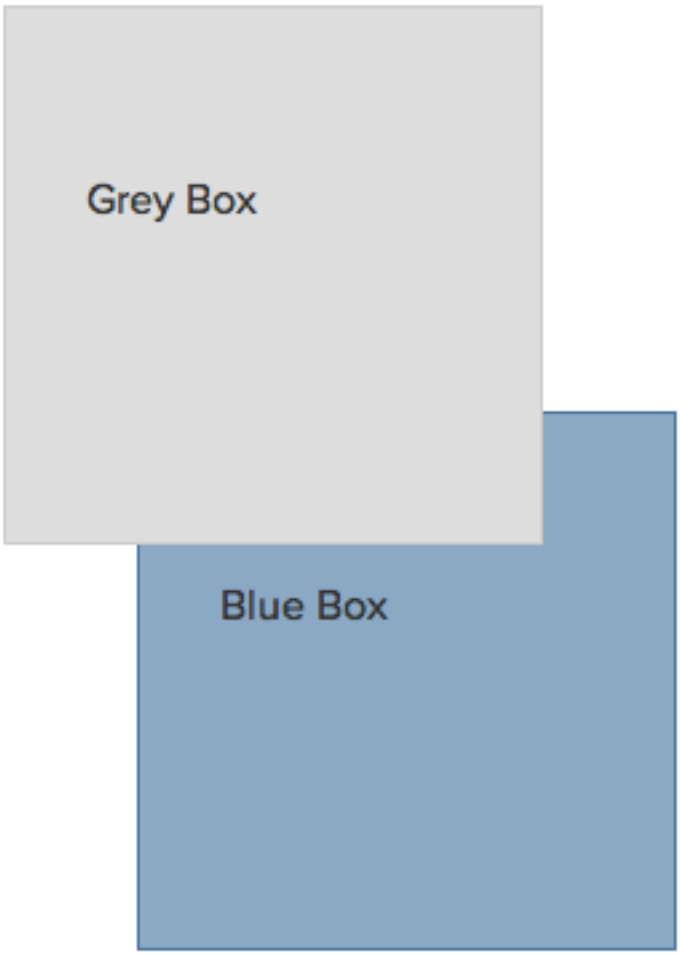
The **<select>** element appears first in the natural stack order and is given a **z-index** value of “1″ along with a position value of “relative”. The gold box appears second in the stack order, and is given a **z-index** value of “9999″. Because of natural stack order and **z-index** values, the gold box should appear on top, which it does in all currently used browsers except IE6:

Unless you're viewing this page with IE6, you'll see the gold box above overlapping the **<select>** element.

This bug in IE6 has caused problems with drop-down menus that fail to overlap **<select>** elements. One solution is to use JavaScript to temporarily hide the **<select>** element, then make it reappear when the overlapping menu disappears. Another solution involves using an **<iframe>**.

POSITIONED PARENTS IN IE6/7

Internet Explorer versions 6 and 7 incorrectly reset the stacking context in relation to the nearest positioned ancestor element. To demonstrate this somewhat complicated bug, we'll display two of the boxes again, but this time we'll wrap the first box in a "positioned" element:

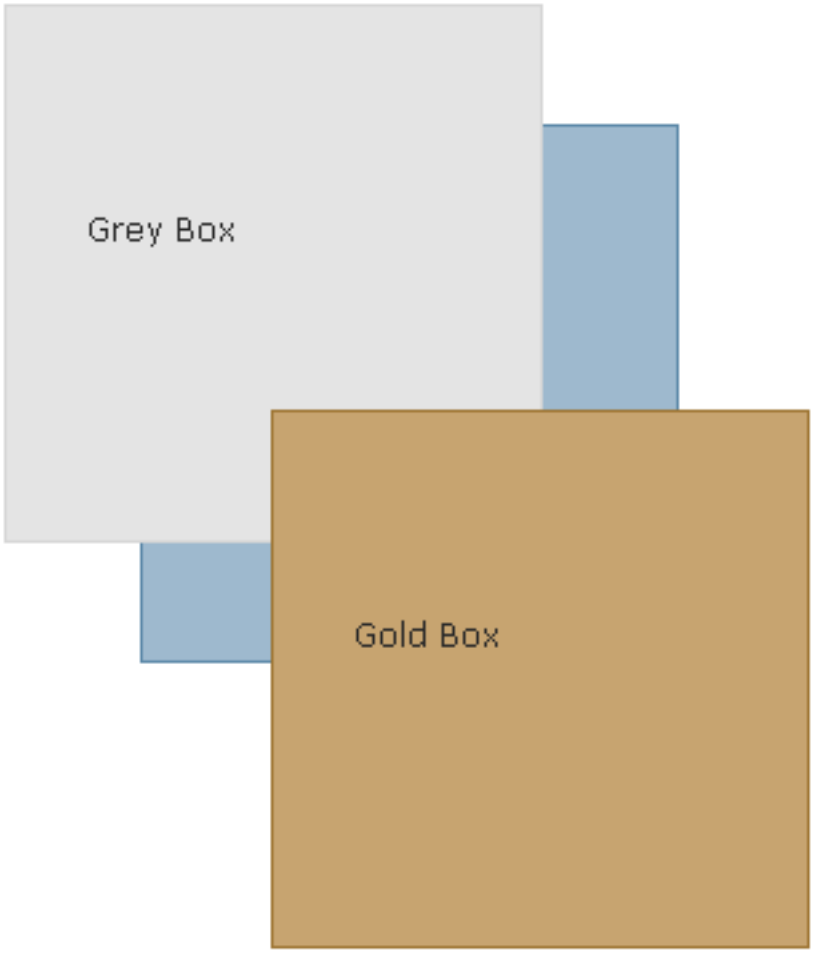


The grey box has a **`z-index`** value of “9999″; the blue box has a **`z-index`** value of “1″ and both elements are positioned. Therefore, the correct implementation is to display the grey box on top of the blue box.

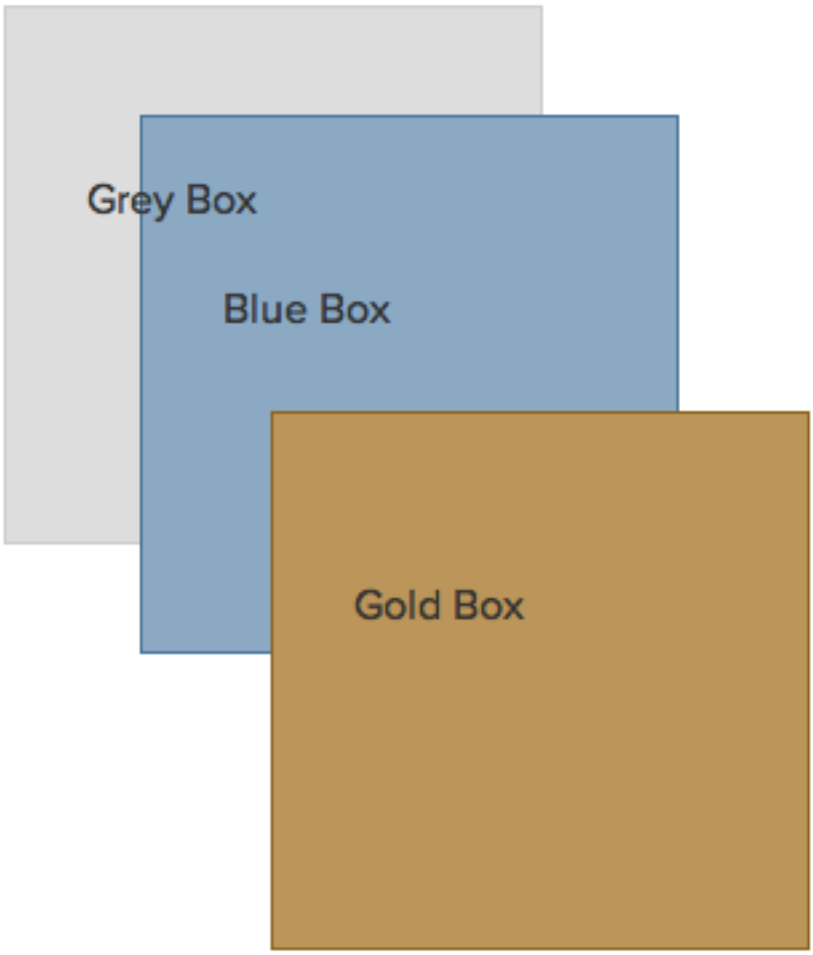
If you view this page in IE6 or IE7, you’ll see the blue box overlapping the grey box. This is caused by the positioned element wrapping the grey box. Those browsers incorrectly “reset” the stacking context in relation to the positioned parent, but this should not be the case. The grey box has a much higher **`z-index`** value, and should therefore be overlapping the blue box. All other browsers render this correctly.

NEGATIVE VALUES IN FIREFOX 2

In Firefox version 2, a negative **`z-index`** value will position an element behind the stacking context instead of in front of the background and borders of the element that established the stacking context. Here is a screen capture displaying this bug in Firefox 2:



Below is the HTML version of the above screen capture, so if you view this page in Firefox 3 or another currently-used browser, you'll see the proper implementation: The background of the grey box (which is the element that establishes the stacking context) appears below everything else, and the grey box's inline text appears above the blue box, which agrees with the "natural stacking order" rules outlined earlier.



Showcase of Various Usage

Applying the **`z-index`** property to elements on a page can provide a quick solution to various layout challenges, and allows designers to be a little more creative with overlapping objects in their designs. Some of the practical and creative uses for the **`z-index`** property are discussed and shown below.

OVERLAPPING TABBED NAVIGATION

The CTCOnlineCME website uses overlapping transparent PNG images with **`z-index`** applied to the "focused" tab, demonstrating practical use for this CSS property:

CSS TOOLTIPS

The **`z-index`** property can be used as part of a CSS-based tooltip, as shown in the example below from trentrichardson.com:

LIGHT BOX

There are dozens of quality light box scripts available for free use, such as the JQuery plugin FancyBox. Most, if not all of these scripts utilize the **`z-index`** property:

Light box scripts use a semi-opaque PNG image to darken the background, while bringing a new element, usually a window-like **`<div>`** to the foreground. The PNG overlay and the **`<div>`** both utilize **`z-index`** to ensure those two elements are above all others on the page.

DROP-DOWN MENUS

Drop down menus such as Brainjar's classic Revenge of the Menu Bar use **`z-index`** to ensure the menu buttons and their drop-downs are at the top of the stack.

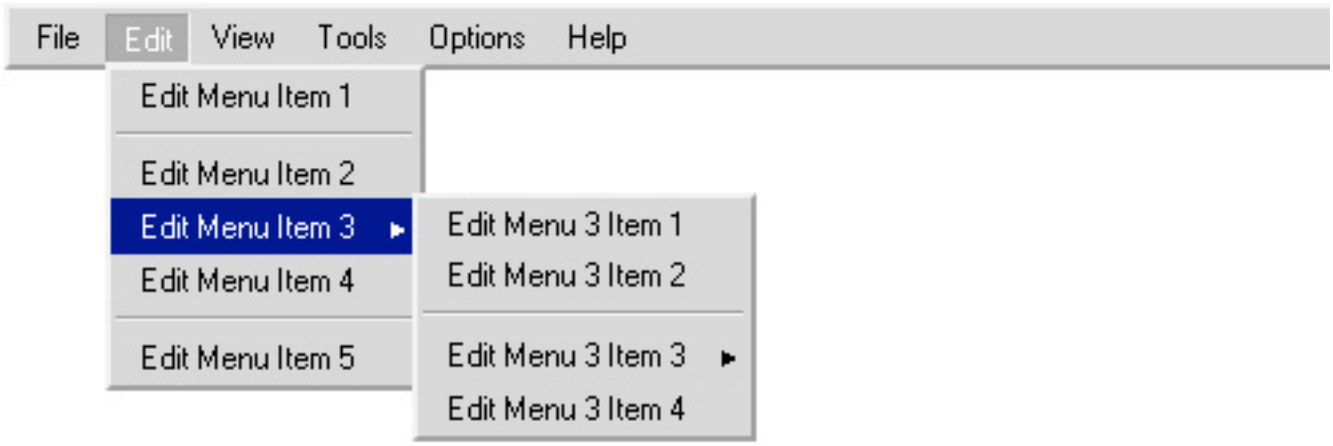
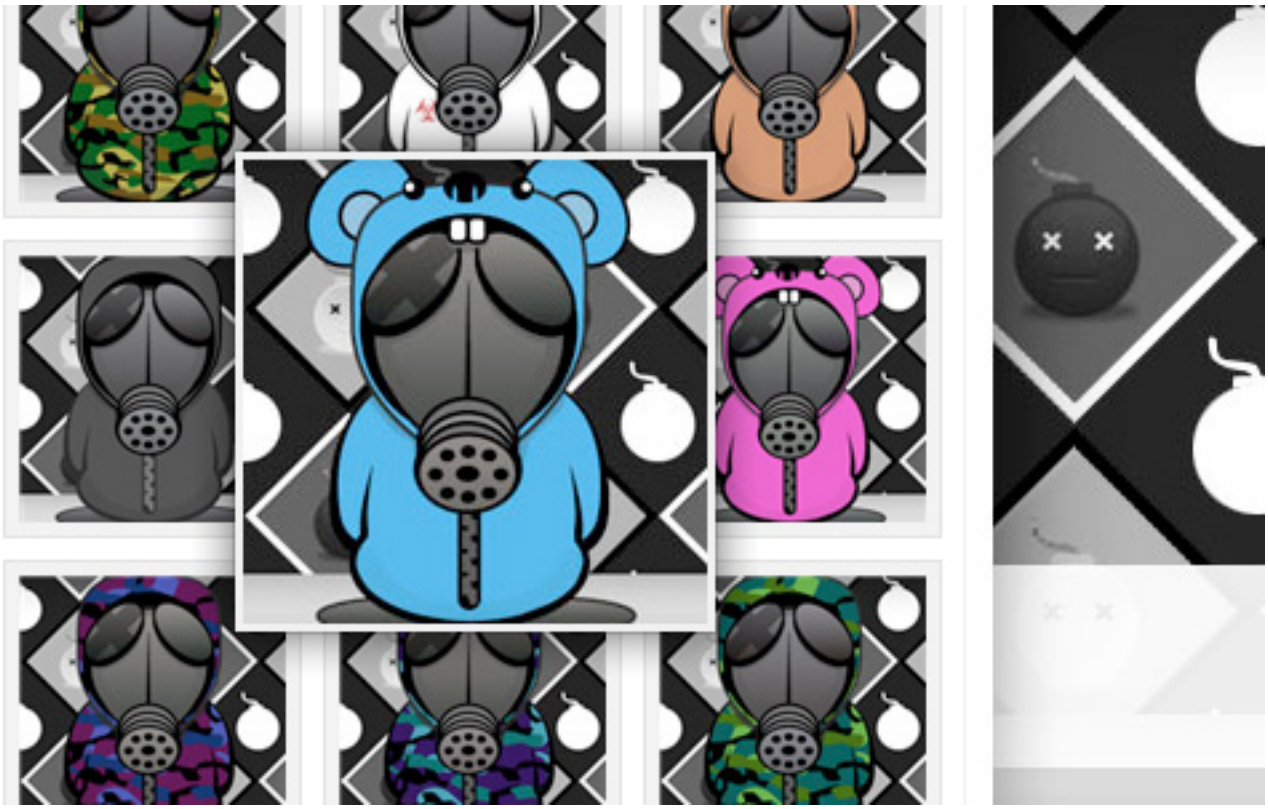


PHOTO GALLERY EFFECTS

A unique combination of JQuery animation and **`z-index`** can create a unique effect for use in a slideshow or photo gallery, as shown in the example below from the usejquery.com website:

Polaroid Photo Gallery by Chris Spooner utilizes some CSS3 enhancements combined with **`z-index`** to create a cool re-stacking effect on hover.

In this Fancy Thumbnail Hover Effect Soh Tanaka changes **`z-index`** values in a JQuery-based script:

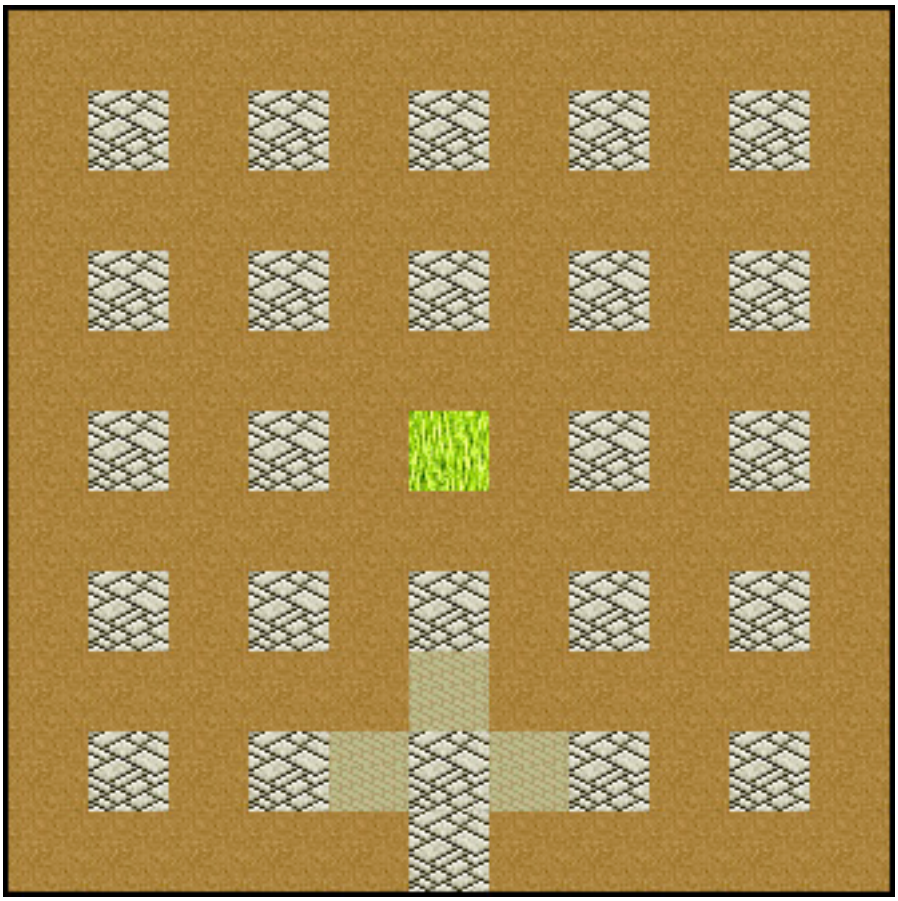


CSS EXPERIMENTS BY STU NICHOLLS

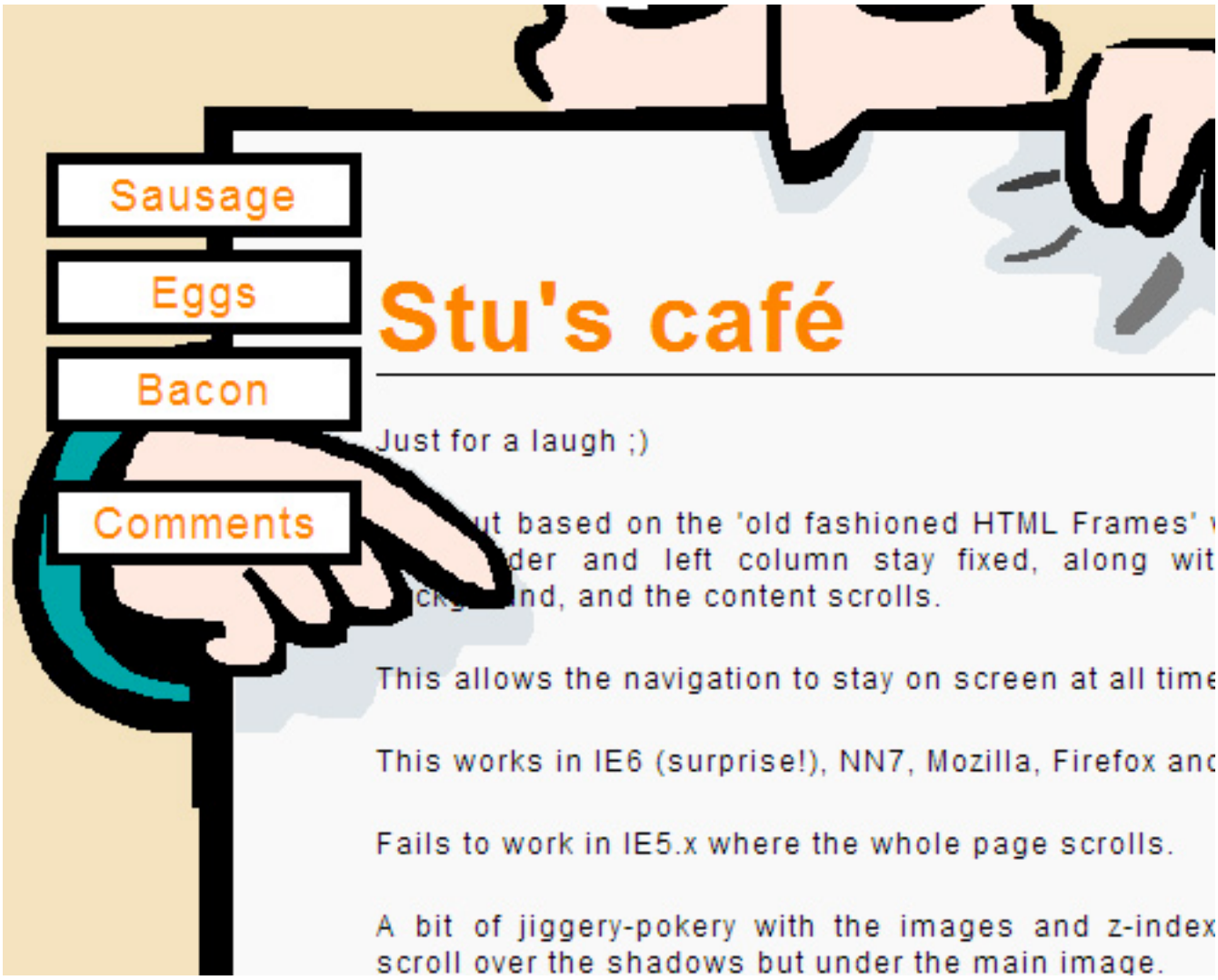
Stu Nicholls describes a number of CSS experiments on his website CSSplay. Here are a few that make creative use of the **`z-index`** property:

CSS image map

CSS game



CSS Frames Emulation



LAYERED LAYOUT ENHANCEMENTS

The 24 ways website implements **`z-index`** to enhance the site's template, weaving year and date columns that stretch the length and width of the site's content, for a very interesting effect.

FANCY SOCIAL BOOKMARKING BOX

The Janko At Warp Speed site uses **`z-index`** in a “fancy share box”:

PERFECT FULL PAGE BACKGROUND IMAGE

This technique was described by Chris Coyier and used on the ringvemedia.com website. It implements **`z-index`** on content sections to ensure they appear above the “background” image, which is not a background image, but only mimics one:

Conclusion

Stacking contexts in CSS are a complex topic. This article did not attempt to discuss every detail on that topic, but instead has attempted to provide a solid discussion of how a web page's stacking contexts are affected by **`z-index`**, which, when fully understood, becomes a powerful CSS property.

Beginning developers should now have a good understanding of this property and avoid some of the common problems that arise when trying to implement it. Additionally, advanced developers should have a stronger understanding of how proper use of **`z-index`** can provide solutions to countless layout issues and open doors to a number of creative possibilities in CSS design.

CSS Sprites: Useful Techniques, Or Potential Nuisance?

Louis Lazaris

Ah, the ubiquitous CSS sprites — one of the few web design techniques that was able to bypass "trend" status almost instantly, planting itself firmly into the category of best practice CSS. Although it didn't really take off until well after A List Apart explained and endorsed it, it was discussed as a CSS solution as early as July, 2003 by Petr Stanícek.

Most web developers today have a fairly strong grasp of this technique, and there have been countless tutorials and articles written on it. In almost every one of those tutorials, the claim is made that designers and developers should be implementing CSS sprites in order to minimize HTTP requests and save valuable kilobytes. This technique has been taken so far that many sites, including Amazon, now use mega sprites.

Is this much-discussed benefit really worthwhile? Are designers jumping on the CSS sprite bandwagon without a careful consideration of all the factors? In this article, I'm going to discuss some of the pros and cons of using CSS sprites, focusing particularly on the use of "mega" sprites, and why such use of sprites could in many cases be a waste of time.

Browsers Cache All Images

One of the benefits given by proponents of the sprite method is the load time of the images (or in the case of mega sprites, the single image). It's argued that a single GIF image comprising all the necessary image states will be significantly lower in file size than the equivalent images all sliced up. This is true. A single GIF image has only one color table associated with it, whereas each image in the sliced GIF method will have its own color table, adding up the kilobytes. Likewise, a single JPEG or PNG sprite will likely save on file size over the same image sliced-up into multiple images. But is this really such a significant benefit?

By default, image-based browsers will cache all images — whether the images are sprites or not. So, while it is certainly true that bandwidth will be saved with the sprite technique, this only occurs on the initial page load, and the caching will extend to secondary pages that use the same image URLs.

The Firefox cache displaying images from amazon.com that the browser cached (type "about:cache" in the address bar in Firefox to view this feature).

When you combine that with the fact that internet speeds are higher on average today than they were when this technique was first expounded upon in 2003-2004, then there may be little reason to use the mega sprite method. And just to be clear, as already mentioned, I'm not saying sprites should never be used; I'm saying they should not be *overused* to attain limited benefits.

Time Spent Slicing a Design Will Increase

Think about how a simple 3-state image button sprite is created: The different states need to be placed next to one another, forming a single image. In your Photoshop mockup (or other software), you don't have the different states adjacent to each other in that manner; they need to be sliced and combined into a new separate image, outside of the basic website mockup.

If any changes are required for any one of the image states, the entire image needs to be recompiled and re-saved. Some developers may not have a problem with this. Maybe they keep their button states separate from the mockup in a raw original, making it easier to access them. But this complicates things, and will never be as simple as slicing a single image and exporting it.

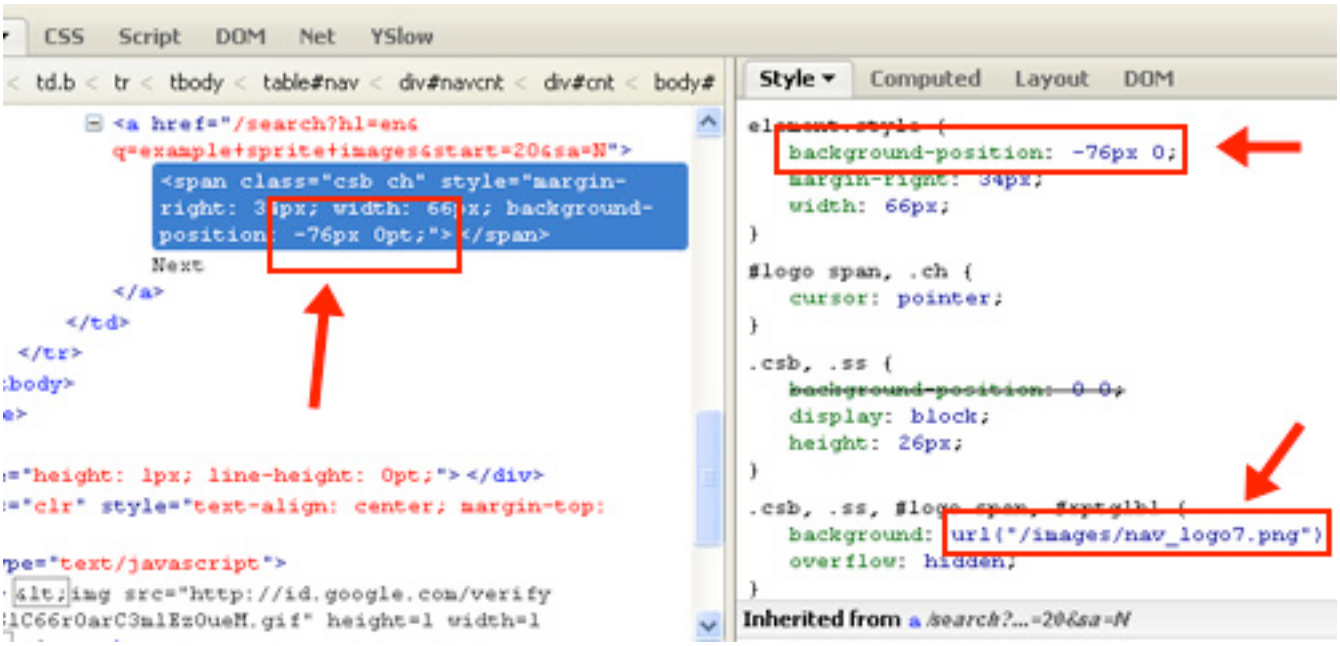
For the minimal benefit of a few kilobytes and server requests saved (which only occurs on initial page load), is the mega sprite method really a practical solution for anything other than a 3-state button?

Time Spent Coding and Maintaining Will Increase

After an image is sliced and exported, the trouble doesn't end there. While button sprites are simple to code into your CSS once you're accustomed to the process, other kinds of sprites are not so simple.

A single button will usually be a **<ul>** element that has a set width. If the sprites for the button are separate for each button, it's simple: The width and height of the **<ul>** will be the same as the width and height of the list item and anchor, with the sprite aligned accordingly for each state. The position of the sprite is easily calculated based on the height and/or width of each button.

But what about a mega sprite, like the one used by Amazon, mentioned earlier, or the one used by Google? Can you imagine maintaining such a file, and making changes to the position of the items in the CSS? And what about the initial creation of the CSS code? Far from being a simple button whose state positions are easily calculated, the mega sprite will often require continuous testing and realigning of the image states.



Some of the CSS used to position Google's sprite image

It's true that the Amazon sprite saves about 30 or more HTTP requests, and that is definitely a significant improvement in performance. But when that benefit is weighed against the development and maintenance costs, and the caching and internet speed issues are factored in, the decision to go with sprites in the mega format may not be so convincing.

DO SPRITES REALLY REQUIRE "MAINTENANCE"?

Of course, some may feel that sprites do not cause a major headache for them. In many cases, after a sprite is created and coded, it's rarely touched again, and isn't affected by any ongoing website maintenance. If you feel that sprite maintenance won't be an issue for you, then the best choice may be to use the mega sprite method.

Not Everything Should Be a Background

Another reason not to promote the overuse of CSS sprites is that it could cause developers to use background images incorrectly. Experienced developers who consider accessibility in their projects understand that not every image should be a background. Images that convey important content should be implemented through inline images in the XHTML, whereas background images should be reserved for buttons and decorative elements.

Amazon correctly places content images as inline elements, and decorative ones as backgrounds.

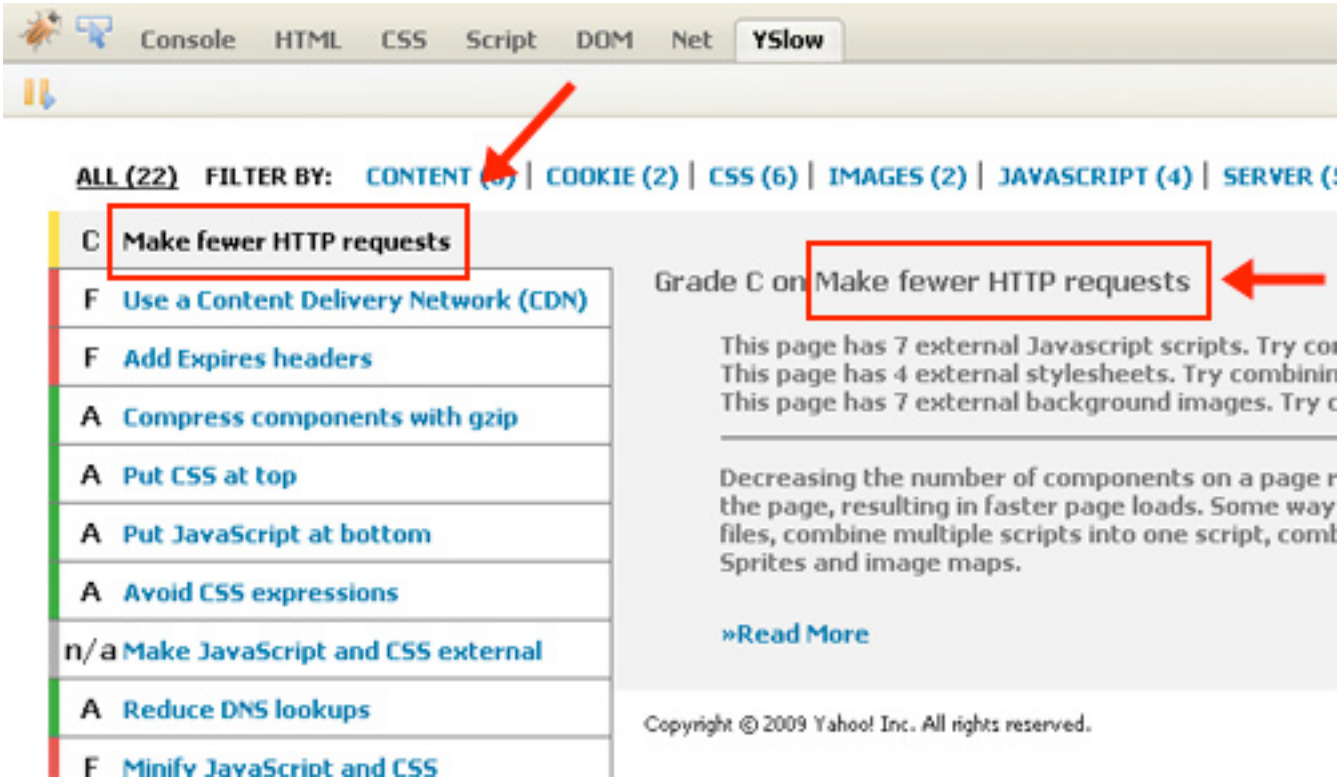
IMPROPER USE OF SPRITES AFFECTS ACCESSIBILITY

Because of the strong emphasis placed on using CSS sprites, some beginning developers intending on saving HTTP requests may incorrectly assume that all sliced images should be placed as backgrounds — even images that convey important information. The results would be a less accessible site, and would limit the potential benefits of the title and alt attributes in the HTML.

So, while CSS sprites in and of themselves are not wrong, and do not cause accessibility problems (in fact, when used correctly, they improve accessibility), the over-promotion of sprites without clearly identifying drawbacks and correct use could hinder the progress the web has made in areas of accessibility and productivity.

What About HTTP Requests?

Many will argue, however (and for good reason) that the most important part of improving a site's performance is minimizing HTTP requests. It should also be noted that one study conducted showed that 40-60% of daily website visitors come with an empty browser cache. Is this enough to suggest that mega sprites should be used in all circumstances? Possibly. Especially when you consider how important a user's first visit to a website is.



The YSlow Firefox add-on that analyzes performance shows the number of HTTP requests being made

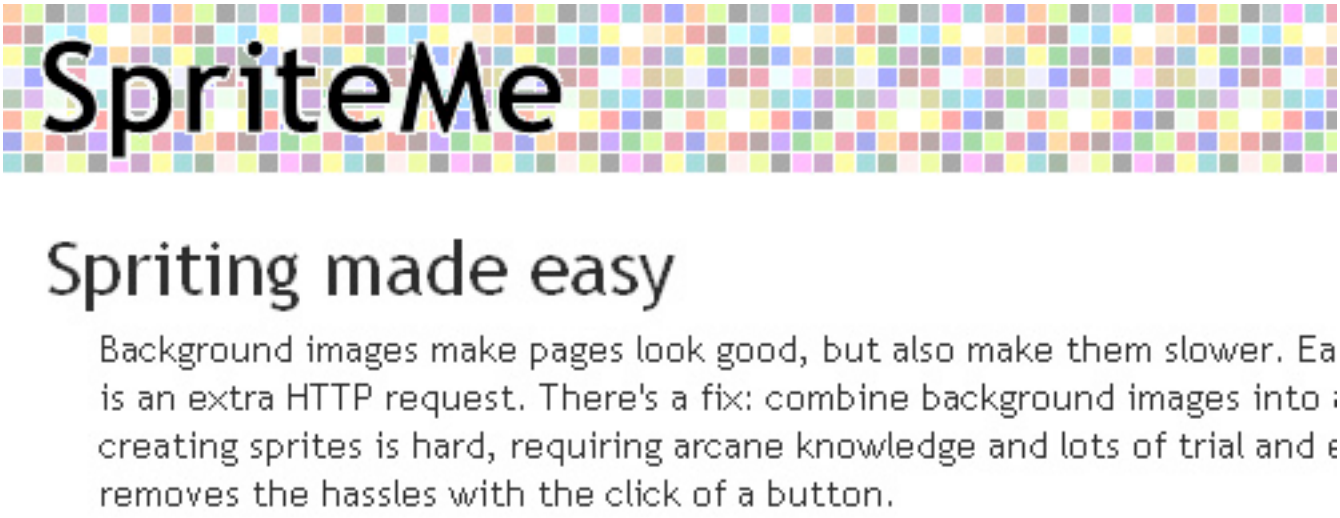
While it is true that older browsers generally only allowed two simultaneous HTTP connections, Firefox since version 3.0 and Internet Explorer 8 by default allow 6 simultaneous HTTP connections. This means 6 simultaneous connections per server. To quote Steve Souders:

> *It's important to understand that this is on a per server basis. Using multiple domain names, such as 1.mydomain.com, 2.mydomain.com, 3.mydomain.com, etc., allows a web developer to achieve a multiple of the per server connection limit. This works even if all the domain names are CNAMEs to the same IP address.*

So, while there could be a benefit to using CSS sprites beyond just button states, in the future, as internet connection speeds increase and newer browser versions make performance improvements, the benefits that come from using mega sprites could become almost irrelevant.

What About Sprite Generators?

Another argument in favor of mega sprites is the apparent ease with which sprites can be created using a number of sprite generators. A detailed discussion and review of such tools is well beyond the scope of this article, so I'll avoid that here. But from my research of these tools, the help they provide is limited, and maintenance of the sprites will still take a considerable amount of work, which again should be weighed against the benefits.



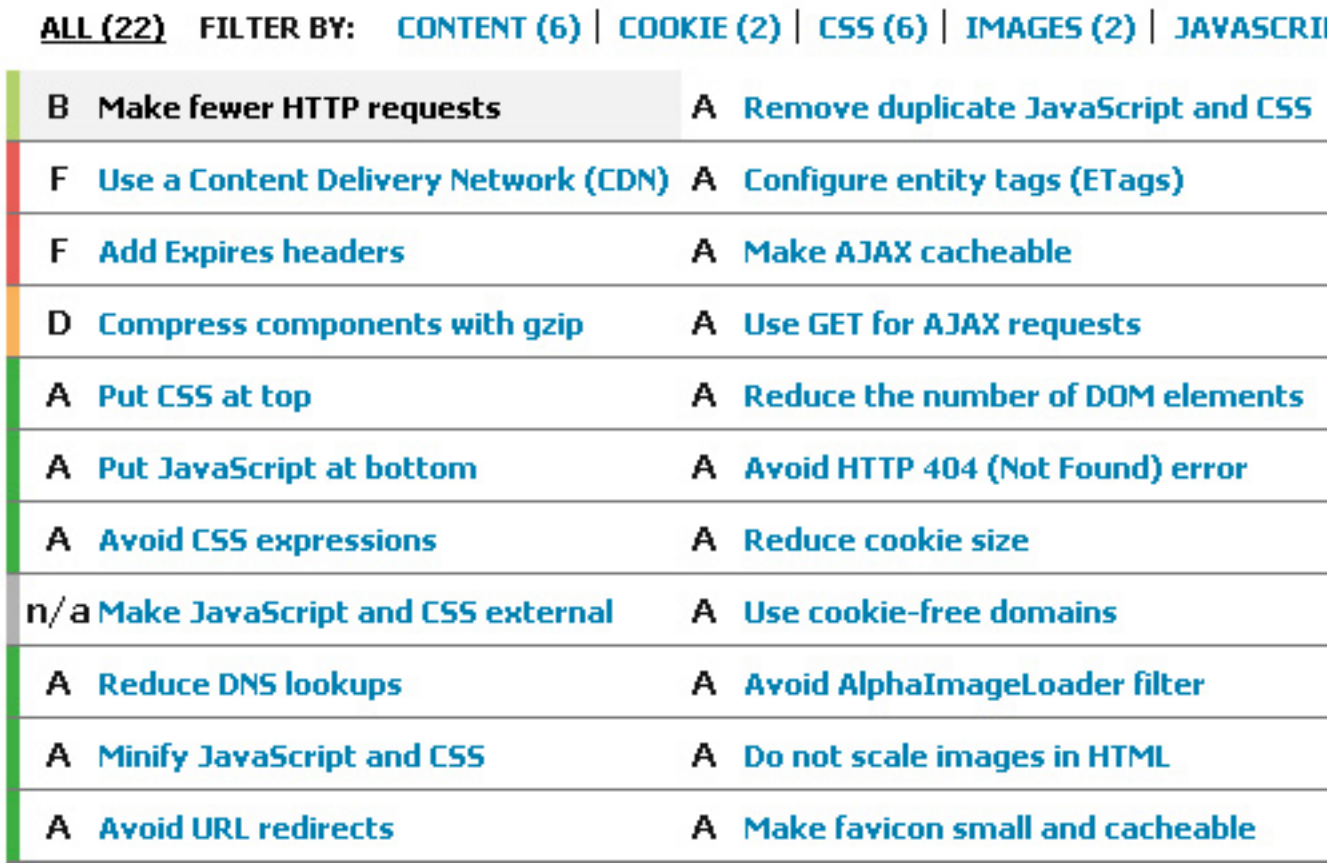
Some tools, like the one by Project Fondue, offer CSS output options. Steve Souders' tool SpriteMe is another one that offers CSS coding options. SpriteMe will convert an existing website's background images into a single sprite image (what I've been referring to as a "mega" sprite) that you can download and insert into your page with the necessary CSS code. While this tool will assist with the creation of sprites, it doesn't seem to offer much help in the area of sprite maintenance. Souders' tool seems to be useless for a website that is redesigned or realigned, and only seems to provide benefit to an existing design that has not yet used the mega sprite method.

Improvements to current tools could be made, and newer tools could appear in the future. But is it possible, due to some of the drawbacks mentioned above, that developers are focusing a lot of effort on a very minimal gain?

Focus Should be on Multiple Performance Issues

As mentioned, the number of HTTP requests is an important factor to consider in website performance. But there are other ways to lower that number, including combining scripts and stylesheets, and using remote locations for library files.

There are also areas outside of HTTP requests that a developer can focus on to improve site performance. These might include GZipping, proper placement of external scripts, optimizing CSS syntax, minifying large JavaScript files, improving Ajax performance, avoiding JavaScript syntax that's known to cause performance issues, and much more.



YSlow indicates many areas outside of HTTP requests that can improve site performance

If developers take the time to consider all factors in website performance, and weigh the pros and cons correctly, there may be good reason to avoid overusing CSS sprites, and focusing on areas whose performance return is worth the investment.

Conclusion

Please don't misinterpret anything that I've said here. Many top bloggers and developers have for years extolled the benefits of using sprites, and in recent years taken those suggestions further in promoting the use of mega sprites — so those opinions should be taken seriously. However, not everyone has the luxury of working in a company that has policies and systems in place that make website maintenance tasks simple and streamlined. Many of us work on our own, or have to inherit projects created by others. In those cases, mega sprites may cause more trouble than they're worth.

What do you think? Should we reconsider the use of mega sprites in CSS development? Do the statistics in favor of the savings on HTTP requests warrant the use of sprites for all background images? Or have sprites in CSS development evolved from a useful, intuitive and productive technique to a time-consuming nuisance?

Modern CSS Layouts: The Essential Characteristics

Zoe Mickley Gillenwater

Now is an exciting time to be creating CSS layouts. After years of what felt like the same old techniques for the same old browsers, we're finally seeing browsers implement CSS 3, HTML 5 and other technologies that give us cool new tools and tricks for our designs.

But all of this change can be stressful, too. How do you keep up with all of the new techniques and make sure your Web pages look great on the increasing number of browsers and devices out there? In part 1 of this article, you'll learn the five essential characteristics of successful modern CSS websites. In part 2 of this article, you'll learn about the techniques and tools that you need to achieve these characteristics.

We won't talk about design trends and styles that characterize modern CSS-based layouts. These styles are always changing. Instead, we'll focus on the broad underlying concepts that you need to know to create the most successful CSS layouts using the latest techniques. For instance, separating content and presentation is still a fundamental concept of CSS Web pages. But other characteristics of modern CSS Web pages are new or more important than ever. A modern CSS-based website is: progressively enhanced, adaptive to diverse users, modular, efficient and typographically rich.

- Progressively enhanced,
- Adaptive to diverse users,

- Modular,
- Efficient,
- Typographically rich.

Progressive Enhancement

Progressive enhancement means creating a solid page with appropriate markup for content and adding advanced styling (and perhaps scripting) to the page for browsers that can handle it. It results in web pages that are usable by all browsers but that do not look identical in all browsers. Users of newer, more advanced browsers get to see more cool visual effects and nice usability enhancements.

The idea of allowing a design to look different in different browsers is not new. CSS gurus have been preaching this for years because font availability and rendering, color tone, pixel calculations and other technical factors have always varied between browsers and platforms. Most Web designers avoid "pixel perfection" and have accepted the idea of their designs looking slightly different in different browsers. But progressive enhancement, which has grown in popularity over the past few years, takes it a step further. Designs that are progressively enhanced may look more than slightly different in different browsers; they might look *very* different.

For example, the tweetCC website has a number of CSS 3 properties that add attractive visual touches, like drop-shadows behind text, multiple columns of text and different-colored background "images" (without there having to be actually different images). These effects are seen to various extents in different browsers, with old browsers like IE 6 looking the "plainest." However, even in IE 6, the text is perfectly readable, and the design is perfectly usable.

tweetCC in Safari.

tweetCC in IE 6.

In CSS 3-capable browsers like Safari (top), the tweetCC website shows a number of visual effects that you can't see in IE 6 (bottom).

These significant differences between browsers are perfectly okay, not only because that is the built-in nature of the Web, but because progressive enhancement brings the following benefits:

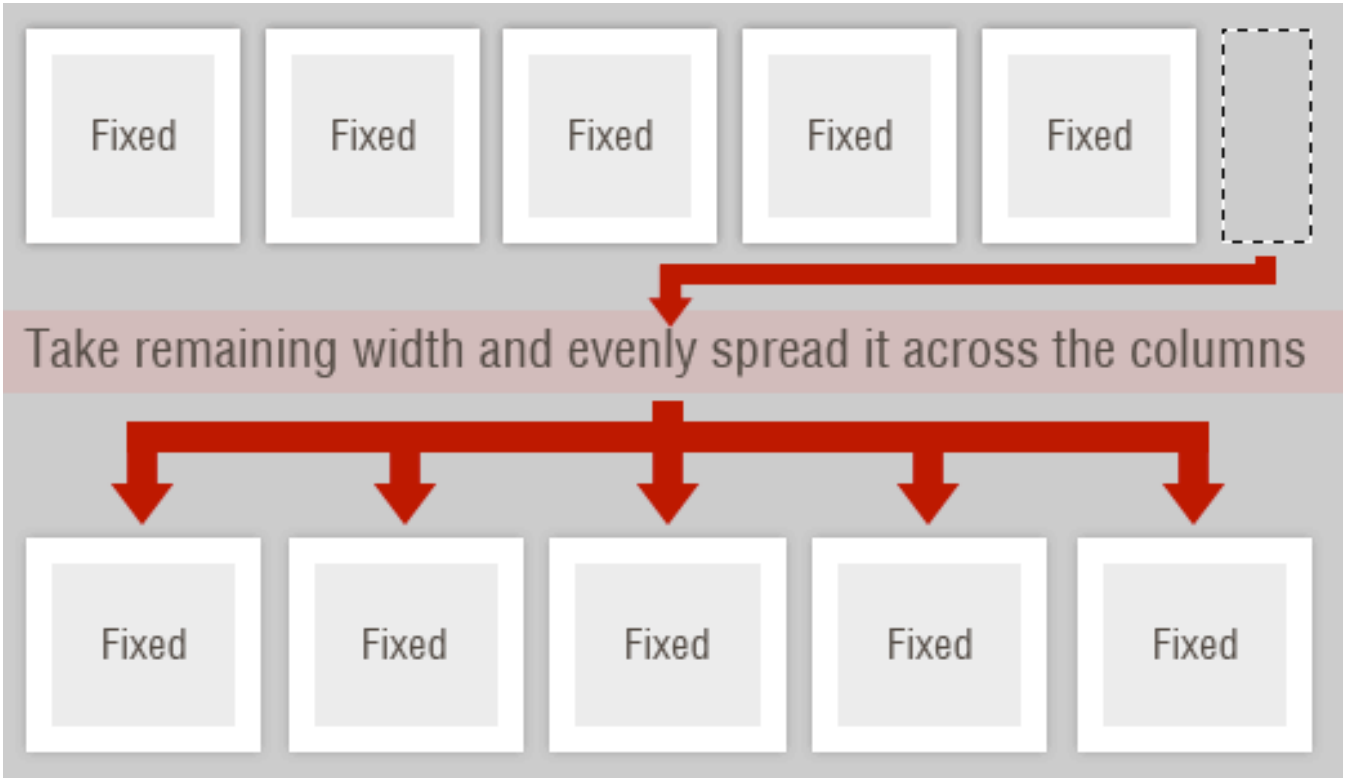
- **More robust pages**
  Rather than use the graceful degradation method to create a fully functional page and then work backwards to make it function in less-capable browsers, you focus first on creating a solid "base" page that works everywhere.

- **Happier users**
  You start building the page making sure the basic functionality and styling is the same for everyone. People with old browsers, mobile devices and assistive technology are happy because the pages are clean and reliable and work well. People with the latest and greatest browsers are happy because they get a rich, polished experience.
- **Reduced development time**
  You don't have to spend hours trying to get everything to look perfect and identical in all browsers. Nor do you have to spend much time reverse-engineering your pages to work in older browsers after you have completed the fully functional and styled versions (as is the case with the graceful degradation method).
- **Reduced maintenance time**
  If a new browser or new technology comes out, you can add new features to what you already have, without altering and possibly breaking your existing features. You have only one base version of the page or code to update, rather than multiple versions (which is the case with graceful degradation).
- **More fun**
  It's just plain fun to be able to use cool and creative new techniques on your Web pages, and not have to wait years for old browsers to die off.

Adaptive to Diverse Users

Modern CSS-based Web pages have to accommodate the diverse range of browsers, devices, screen resolutions, font sizes, assistive technologies and other factors that users bring to the table. This concept is also not new but is growing in importance as Web users become increasingly diverse. For instance, a few years ago, you could count on almost all of your users

having one of three screen resolutions. Now, users could be viewing your pages on 10-inch netbooks, 30-inch widescreen monitors or anything in between, not to mention tiny mobile devices.



In his article "Smart columns with CSS and jQuery" Soh Tanaka describes his techniques that adapts the layout depending on the current browser window size.

Creating Web designs that work for all users in all scenarios will never possible. But the more users you can please, the better: for them, for your clients and for you. Successful CSS layouts now have to be more flexible and adaptable than ever before to the increasing variety of ways in which users browse the Web.

Consider factors such as these when creating CSS layouts:

- **Browser**
  Is the design attractive and usable with the most current and popular browsers? Is it at least usable with old browsers?
- **Platform**
  Does the design work on PC, Mac and Linux machines?
- **Device**
  Does the design adapt to low-resolution mobile devices? How does it look on mobile devices that have full resolution (e.g. iPhones)?
- **Screen resolution**
  Does the design stay together at multiple viewport (i.e. window) widths? Is it attractive and easy to read at different widths? If the design does adapt to different viewport widths, does it correct for extremely narrow or wide viewports (e.g. by using the `min-width` and `max-width` properties)?
- **Font sizes**
  Does the design accommodate different default font sizes? Does the design hold together when the font size is changed on the fly? Is it attractive and easy to read at different font sizes?
- **Color**
  Does the design make sense and is the content readable in black and white? Would it work if you are color blind or have poor vision or cannot detect color contrast?
- **JavaScript presence**
  Does the page work without JavaScript?

- **Image presence**
  Does the content make sense and is it readable without images (either background or foreground)?
- **Assistive technology/disability**
  Does the page work well in screen readers? Does the page work well without a mouse?

This is not a comprehensive list; and even so, you would not be able to accommodate every one of these variations in your design. But the more you can account for, the more user-friendly, robust and successful your website will be.

Modular

Modern websites are no longer collections of static pages. Pieces of content and design components are reused throughout a website and even shared between websites, as content management systems (CMS), RSS aggregation and user-generated content increase in popularity. Modern design components have to be able to adapt to all of the different places they will be used and the different types and amount of content they will contain.

livepage.apple.comObject Oriented CSS is Nicole Sulivan's attempt to create a framework that would allow developers to write fast, maintainable, standards-based, modular front end code.

Modular CSS, in a broad sense, is CSS that can be broken down into chunks that work independently to create design components that can themselves be reused independently. This might mean separating your style into multiple sheets, such as *layout.css*, *type.css*, and *color.css*. Or it might mean creating a collection of universal CSS classes for form layout that you can apply to any form on your website, rather than have to style each form individually. CMS', frameworks, layout grids and other tools all help you create more modular Web pages.

Modular CSS offers these benefits (depending on which techniques and tools you use):

- **Smaller file sizes**
  When all of the content across your website is styled with only a handful of CSS classes, rather than an array of CSS IDs that only work on particular pieces of content on particular pages, your style sheets will have many fewer redundant lines of code.
- **Reduced development time**
  Using frameworks, standard classes and other modular CSS tools keeps you from having to re-invent the wheel every time you start a new website. By using your own or other developers' tried and true CSS classes, you spend less time testing and tweaking in different browsers.
- **Reduced maintenance time**
  When your style sheets include broad, reusable classes that work anywhere on your website, you don't have to come up with new styles when you add new content. Also, when your CSS is lean and well organized, you spend less time tracking down problems in your style sheets when browser bugs pop up.
- **Easier maintenance for others**
  In addition to making maintenance less time-consuming for you, well-organized CSS and smartly named classes also make maintenance easier for developers who weren't involved in the initial development of the style sheets. They'll be able to find what they need and use it more easily. CMS' and frameworks also allow people who are not as familiar with your website to update it easily, without screwing anything up.

- **More design flexibility**
  Frameworks and layout grids make it easy, for instance, to switch between different types of layout on different pages or to plug in different types of content on a single page.
- **More consistent design**
  By reusing the same classes and avoiding location-specific styling, you ensure that all elements of the same type look the same throughout the website. CMS' and frameworks provide even more insurance against design inconsistency.

Efficient

Modern CSS-based websites should be efficient in two ways:

- Efficient for you to develop,
- Efficient for the server and browser to display to users.

As Web developers, we can all agree that efficiency on the development side is a good thing. If you can save time while still producing high-quality work, then why wouldn't you adopt more efficient CSS development practices? But creating pages that perform efficiently for users is sometimes not given enough attention. Even though connection speeds are getting faster and faster, page load times are still very important to users. In fact, as connection speeds increase, users might expect all pages to load very quickly, so making sure your website can keep up is important. Shaving just a couple of seconds off the loading time can make a big difference.

We've already discussed how modular CSS reduces development and maintenance time and makes your workflow a lot faster and more efficient. A myriad of tools are out there to help you write CSS quickly, which we'll cover in part 2 of this article. You can also streamline your CSS development

process by using many of the new effects offered by CSS 3, which cut down on your time spent creating graphics and coding usability enhancements.

Some CSS 3 techniques also improve performance and speed. For instance, traditional rounded-corner techniques require multiple images and DIVs for just one box. Using CSS 3 to create rounded corners requires no images, thus reducing the number of HTTP calls to the server and making the page load faster. No images also reduces the number of bytes the user has to download and speeds up page loading. CSS 3 rounded-corners also do not require multiple nested DIVs, which reduces page file size and speeds up page loading again. Simply switching to CSS 3 for rounded corners can give your website a tremendous performance boost, especially if you have many boxes with rounded corners on each page.

Writing clean CSS that takes advantage of shorthand properties, grouped selectors and other efficient syntax is of course just as important as ever for improving performance. Many of the more advanced tricks for making CSS-based pages load faster are also not new but are increasing in usage and importance. For instance, the CSS Sprites technique, whereby a single file holds many small images that are each revealed using the CSS **`background-position`** property, was first described by Dave Shea in 2004 but has been improved and added to a great deal since then. Many large enterprise websites now rely heavily on the technique to minimize HTTP requests. And it can improve efficiency for those of us working on smaller websites, too. CSS compression techniques are also increasingly common, and many automated tools make compressing and optimizing your CSS a breeze, as you'll also learn in part 2 of this article (next chapter).

Rich Typography

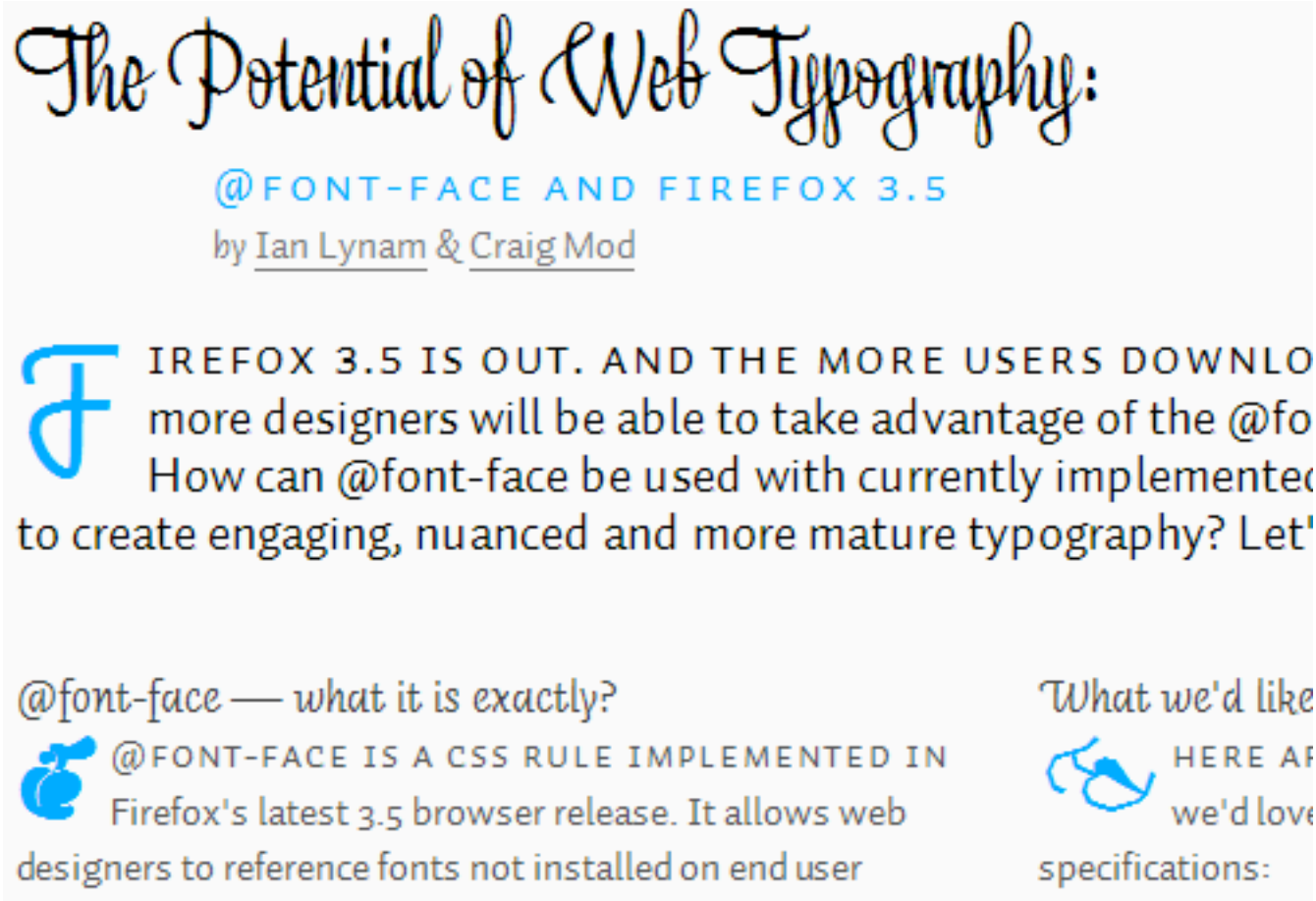
Rich typography may seem out of place with the four concepts we have just covered. But we're not talking about any particular style of typography or fonts, but rather the broader concept of creating readable yet unique-looking text by applying tried and true typographic principles using the newest technologies. Typography is one of the most rapidly evolving areas of Web design right now. And boy, does it need to evolve! While Web designers have had few limits on what they could do graphically with their designs, their limits with typography have been glaring and frustrating.

Until recently, Web designers were limited to working with the fonts on their end users' machines. Image replacement tricks and clever technologies such as sIFR have opened new possibilities in the past few years, but none of these is terribly easy to work with. In the past year, we've finally made great strides in what is possible for type on the Web because of the growing support for CSS 3's **`@font-face`** property, as well as new easy-to-use technologies and services like Cufón and Typekit.

The **`@font-face`** rule allows you to link to a font on your server, called a "Web font," just as you link to images. So you are no longer limited to working with the fonts that most people have installed on their machines. You can now take advantage of the beautiful, unique fonts that you have been dying to use.

@font-face in action: Teehanlax.com

Craigmod



Nicewebtype

The three screenshots above are all examples of what @font-face can do.

The main problem with **`@font-face`**, aside from the ever-present issue of browser compatibility, is that most font licenses—even those of free fonts—do not allow you to serve the fonts over the Web. That's where **`@font-face`** services such as Typekit, Fontdeck and Kernest are stepping in. They work with type foundries to license select fonts for Web design on a "rental" basis. These subscription-based services let you rent fonts for your website, giving you a much wider range of fonts to work with, while avoiding licensing issues.

For A Beautiful Web uses the Typekit font embedding service for the website name, introductory text and headings.

Ruler of the Interwebs uses the Kernest font embedding service for the website name and headings.

We still have a long way to go, but the new possibilities make typography more important to Web design than ever before. To make your design truly stand out, use these modern typographic techniques, which we'll cover in even greater detail in the next chapter.

Summary

We've looked at five characteristics of modern CSS websites:

- Progressively enhanced,
- Adaptive to diverse users,
- Modular,
- Efficient,
- Typographically rich.

In the following chapter, we'll go over the techniques and tools that will help you implement these important characteristics on your CSS-based Web pages.

Modern CSS Layouts, Part 2: The Essential Techniques

Zoe Mickley Gillenwater

In the previous chapter “Modern CSS Layouts, Part 1: The Essential Characteristics”, you learned that modern, CSS-based web sites should be progressively enhanced, adaptive to diverse users, modular, efficient and typographically rich. Now that you know *what* characterizes a modern CSS web site, *how* do you build one? Here are dozens of essential techniques and tools to learn and use to achieve the characteristics of today’s most successful CSS-based web pages.

Just as in the previous chapter, we’re not going to be talking about design trends and styles; these styles are always changing. Instead, we’re focusing on the specific techniques that you need to know to create modern CSS-based web pages of any style. For each technique or tool, we’ll indicate which of the five characteristics it helps meet. To keep this shorter than an encyclopedia, we’ll also just cover the basics of each technique, then point you to some useful, hand-picked resources to learn the full details.

You can jump straight to:

- CSS3
- HTML5
- IE Filtering
- Flexible Layouts
- Layout Grids

- Efficient CSS Development Practices
- CSS Performance
- Font Embedding and Replacement

CSS3

CSS3, the newest version of CSS that is now being partially supported by most browsers, is the primary thing you need to know in order to create modern CSS web sites, of course. CSS is a styling language, so it's no surprise that most of what's new in CSS3 is all about visual effects. But CSS3 is about more than progressive enhancement and pretty typography. It can also aid usability by making content easier to read, as well as improve efficiency in development and page performance.

There are too many CSS3 techniques to cover in a single article, let alone an article that isn't just about CSS3! So, we'll go through the basics of the most important or supported CSS3 techniques and point you to some great resources to learn more in-depth.

Semi-transparent Color

Aids in: progressive enhancement, efficiency

RGBA allows you to specify a color by not only setting the values of red, green, and blue that it's comprised of, but also the level of opacity it should have. An alternative to RGBA is HSLA, which works the same way, but allows you to set values of hue, saturation, and lightness, instead of values of red, green, and blue. The article Color in Opera 10 — HSL, RGB and Alpha Transparency explains how HSLA can be more intuitive to use than RGBA.

The 24 Ways web site makes extensive use of RGBA to layer semi-transparent boxes and text over each other.

RGBA or HSLA isn't just about making things look cool; it can also improve your web site's efficiency. You don't have to take time to make alpha-transparent PNGs to use as backgrounds, since you can just use a color in the CSS, and the user agent doesn't have to download those images when loading the site.

Styling Backgrounds and Borders

Aids in: progressive enhancement, efficiency

CSS3 offers a whole host of new ways to style backgrounds and borders, often without having to use images or add extra **`div`**s. Most of these new techniques already have good browser support, and since they're mainly used for purely cosmetic changes, they're a good way to get some progressive enhancement goodness going in your sites right away.

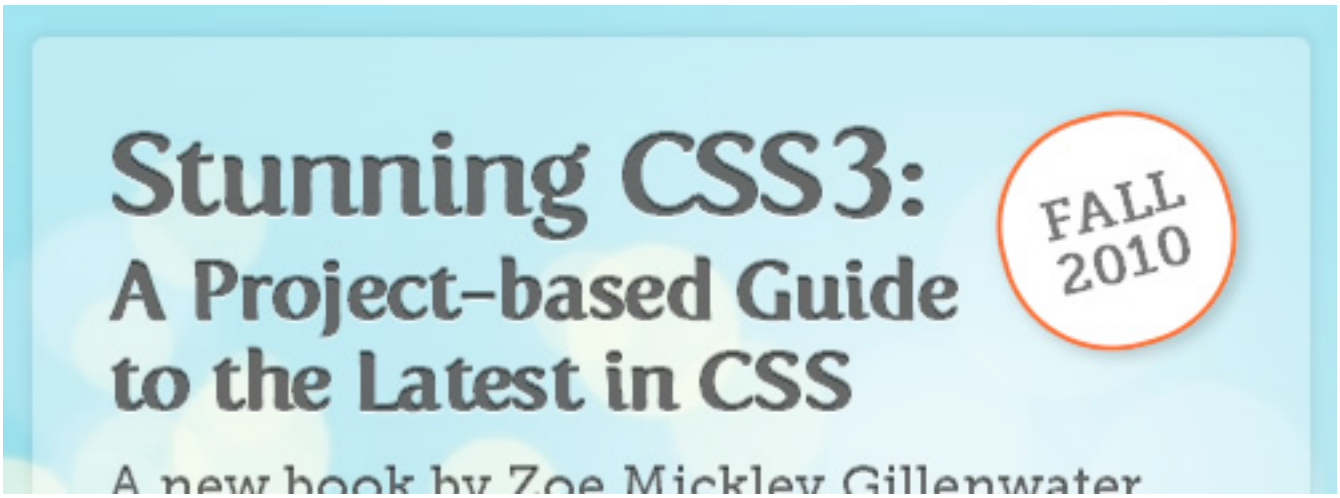
Here are some of the new things CSS3 lets you do with backgrounds:

- **Multiple backgrounds on a single element:** You can now add more than one background image to an element by listing each image, separated by commas, in the **`background-image`** property. No more nesting extra **`div`**s just to have more elements to attach background images onto!
- **More control over where backgrounds are placed:** The new **`background-clip`** and **`background-origin`** properties let you control if backgrounds are displayed under borders, padding, or just content, as well as where the origin point for **`background-position`** should be.

- **Background sizing:** You can scale background images using the new **`background-size property`**. While scaling won't look good on many background images, it could be really handy on abstract, grunge-type backgrounds, where tiling can be difficult and where some image distortion would be unnoticeable.
- **Gradients:** While just part of a CSS3 draft spec, Safari, Chrome and Firefox support declaring multiple color and placement values in the **`background-image`** property to create gradients without images. This allows the gradients to scale with their container — unlike image gradients — and eliminates the need for page users to download yet another image while viewing your site.

CSS3 lets you do the following with borders:

- **Rounded corners:** Use the **`border-radius-property`** to get rounded corners on **`div`**s, buttons, and whatever else your heart desires — all without using images or JavaScript.
- **Images for borders:** With CSS 2.1, the only way to create a graphic border was to fake it with background images, often multiple ones pieced together on multiple **`div`**s. You can now add unique borders without having to use background images by adding the images to the borders directly, using the new **`border-image property`**, which also allows you to control how the images scale and tile.



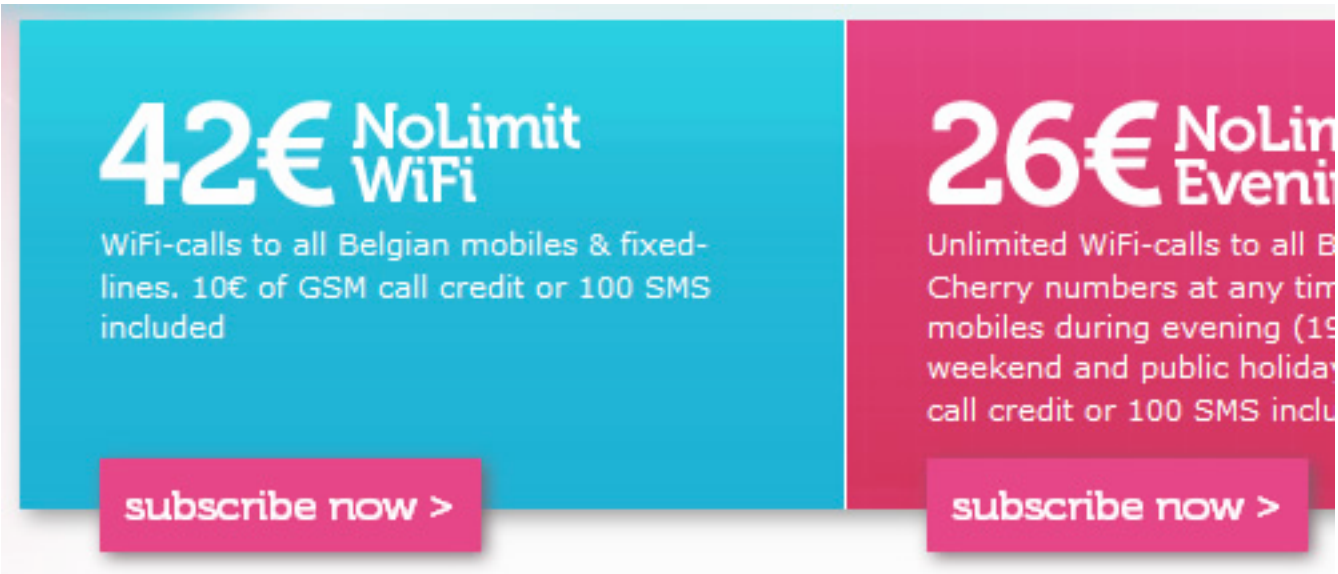
The border-radius property can be used to round corners and even create circles out of pure CSS, with no images needed. (Stunning CSS3 web site)

Drop Shadows

Aids in: progressive enhancement, adaptability, efficiency

Drop shadows can provide some visual polish to your design, and now they're possible to achieve without images, both on boxes and on text.

The **`box-shadow property`** has been temporarily removed from the CSS3 spec, but is supposed to be making its re-appearance soon. In the meantime, it's still possible to get image-free drop shadows on boxes in Firefox and Safari/Chrome using the **`-moz-box-shadow`** and **`-webkit-box-shadow`** properties, respectively, and in Opera 10.5 using the regular **`box-shadow`** property with no prefix. In the property, you set the the shadow's horizontal and vertical offsets from the box, color, and can optionally set blur radius and/or spread radius.



The Cherry web site uses drop shadows created with `box-shadow` on many boxes and buttons.

The **`text-shadow` property** adds drop shadows on — you guessed it — text. It's supported by all the major browsers except — you guessed it — Internet Explorer. This makes it the perfect progressive enhancement candidate — it's simply a visual effect, with no harm done if some users don't see it. Similarly to **`box-shadow`**, it takes a horizontal offset, vertical offset, blur radius and color.

Using **`text-shadow`** keeps you from resorting to Flash or images for your text. This can speed up the time it takes you to develop the site, as well as speed up your pages. Avoiding Flash and image text can also aid accessibility and usability; just make sure your text is still legible with the drop shadow behind it, so you don't inadvertently *hurt* usability instead!

Transforms

Aids in: progressive enhancement, adaptability, efficiency

CSS3 makes it possible to do things like rotate, scale, and skew the objects in your pages without resorting to images, Flash, or JavaScript. All of these effects are called "transforms." They're supported in Firefox, Safari, Chrome, and Opera 10.5.

You apply a transform using the **`transform`** property, naturally (though for now you'll need to use the browser-specific equivalents: **`-moz-transform`**, **`-webkit-transform`**, and **`-o-transform`**). You can also use the **`transform-origin`** property to specify the point of origin from which the transform takes place, such as the center or top right corner of the object.

In the **`transform`** property, you specify the type of transform (called "transform functions"), and then in parentheses write the measurements needed for that particular transform. For instance, a value of **`translate(10px, 20px)`** would move the element 10 pixels to the right and 20 pixels down from its original location in the flow. Other supported transform functions are `scale`, **`rotate`**, and **`skew`**.

The [BeerCamp SXSW 2010 site] scales and rotates the sponsor logos on hover.

Animation and Transitions

Aids in: progressive enhancement, efficiency

Animation is now no longer the solely the domain of Flash or JavaScript — you can now create animation in pure HTML and CSS. Unfortunately, CSS3 animation and transitions do not have very good browser support, but as with most of the effects we've talked about so far, they're great for adding a little non-essential flair.

CSS3 transitions are essentially the simplest type of animation. They smoothly ease the change between one CSS value to another over a specified duration of time. They're triggered by changing element states, such as hovering. They're supported by Safari, Chrome, and Opera 10.5.

To create a transition, all you have to do is specify which elements you want to apply the transition to and which CSS properties will transition, using the **`transition-property`** property. You'll also need to add a

`transition-duration` value in seconds ("s" is the unit), since the default time a transition takes is 0 seconds. You can add them both in the **`transition`** shorthand property. You can also specify a delay or a timing function to more finely tune how the two values switch.

Beyond transitions, full-fledged animations with multiple keyframes are also possible with CSS3 (but currently only supported in Safari/Chrome). First, you give the animation a name and define what the animation will do at different points (keyframes, indicated with percentages) through its duration. Next, you apply this animation to an element using the **`animation-name`**, **`animation-duration`**, and **`animation-interation-count`** properties. You could also set a delay and timing function, just like with transitions.

CSS3 USABILITY / READABILITY ENHANCEMENTS

Most the CSS3 techniques we've gone over so far have been purely cosmetic effects that aid progressive enhancement. But CSS3 can also be used to improve the usability of your pages.

Creating Multiple Columns of Text

Aids in: progressive enhancement, adaptability

Some pieces of text are more readable in narrow, side-by-side columns, similar to traditional newspaper layout. You can tell the browser to arrange your text into columns by either defining a width for each column (the **`column-width`** property) or by defining a number of columns (the **`column-count`** property). Other new properties let you control gutters/gaps, rule lines, breaking between columns and spanning across columns. (For now, you need to use the browser-specific prefixes of **`-moz`** and **`-webkit`**.) This is another one of those techniques that can harm instead of

aid usability if used improperly, as explained in “CSS3 Multi-column layout considered harmful,” so use it judiciously.

Controlling Text Wrapping and Breaking

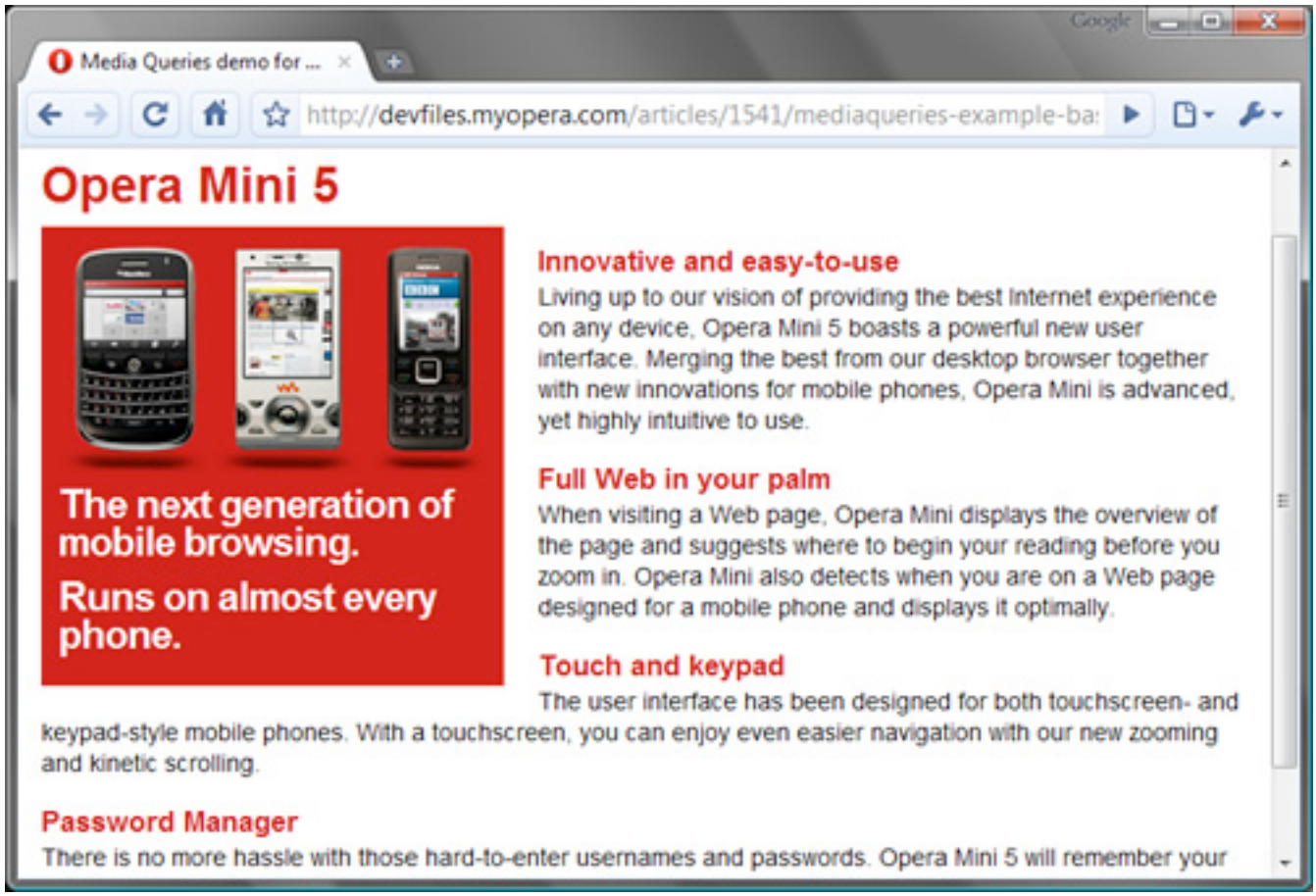
Aids in: adaptability

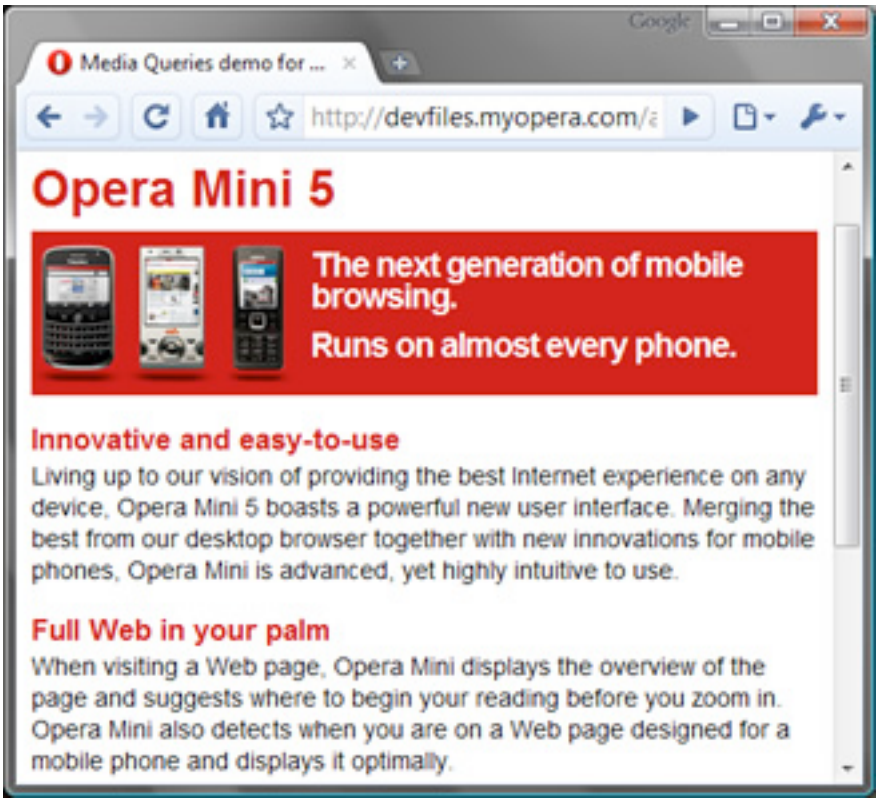
CSS3 gives you more control over how blocks of text and individual words break and wrap if they’re too long to fit in their containers. Setting **`word-wrap`** to **`break-word`** will break a long word and wrap it onto a new line (particularly handy for long URLs in your text). The **`text-wrap`** property gives you a number of options for where breaks may and may not occur between words in your text. The CSS2 **`white-space`** property has now in CSS3 become a shorthand property for the new **`white-space-collapse`** and **`text-wrap`** properties, giving you more control over what spaces and line breaks are preserved from your markup to the rendered page. Another property worth mentioning, even though it’s not currently in the CSS3 specification, is **`text-overflow`**, which allows the browser to add an ellipsis character (…) to the end of a long string of text instead of letting it overflow.

Media Queries

Aids in: adaptability, efficiency

CSS2 let you apply different styles to different media types — screen, print, and so on. CSS3's media queries take this a step further by letting you customize styles based on the user's viewport width, display aspect ratio, whether or not his display shows color, and more. For instance, you could detect the user's viewport width and change a horizontal nav bar into a vertical menu on wide viewports, where there is room for an extra column. Or you could change the colors of your text and backgrounds on non-color displays.





This demo file from Opera uses media queries to rearrange elements and resize text and images based on viewport size.

Media queries couldn't come at a better time — there is more variety in the devices and settings people use to browse the web than ever before. You can now optimize your designs more precisely for these variations to provide a more usable and attractive design, but without having to write completely separate style sheets, use JavaScript redirects, and other less efficient development practices.

Media queries are supported to some degree by all the major browsers except IE. Also, media queries are particularly helpful in serving alternate styles to small-screen mobile devices. IMPROVING EFFICIENCY THROUGH CSS3

Many of the visual effect properties of CSS3 that we've gone over have a great bonus in addition to making your design look great: they can improve efficiency, both in your development process and in the performance of the pages themselves.

Any CSS3 property that keeps you from having to create and add extra images is going to reduce the time it takes you to create new pages as well as re-skin existing ones. Less images also mean less stuff for the server to have to send out and less stuff for the users to download, both of which increase page loading speed.

CSS3 properties that keep you from having to add extra `div`s or extra classes can also reduce your development time as well as file size. We've already gone over some great techniques that help with this, but there are a few more worth mentioning.

The `box-sizing` Property

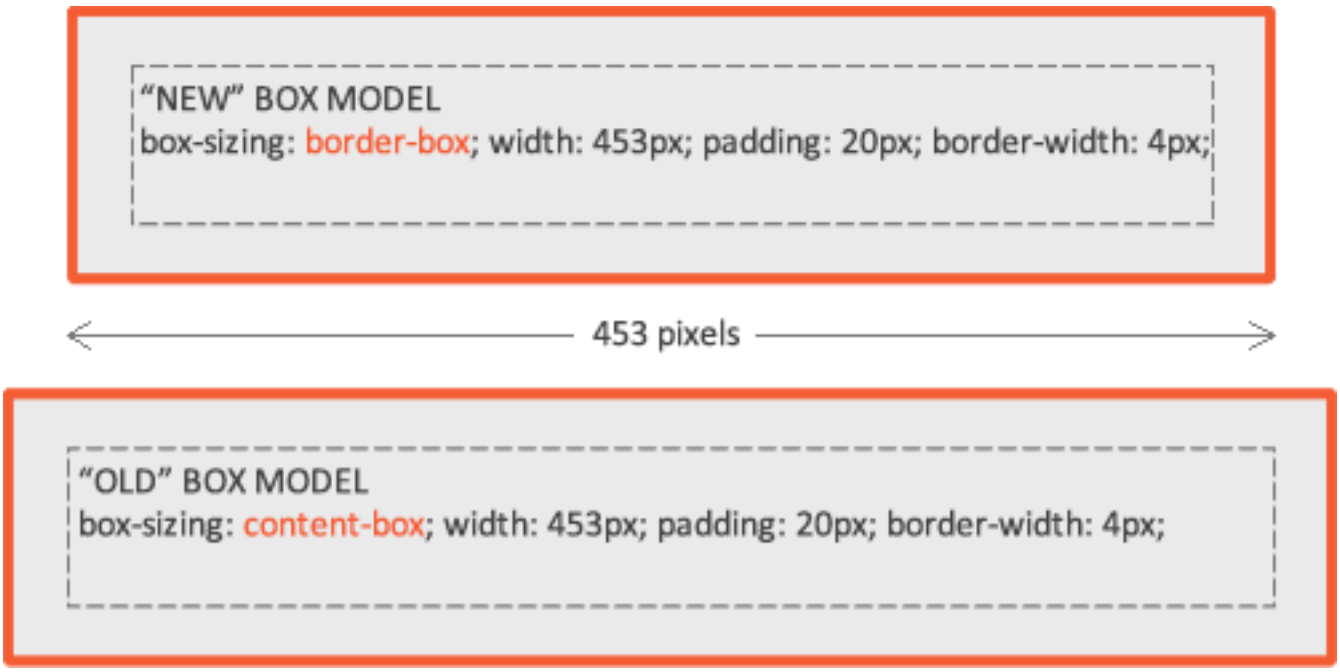
Aids in: efficiency

In addition to the `div`-conserving properties we've already talked about, the **`box-sizing`** property can also help limit your **`div`** use in certain situations.

In the traditional W3C box model of CSS 2.1, the value you declare for a width or height controls the width or height of the *content area* only, and then the padding and border are *added* onto it. (This is called the content-box model.) If you've worked with CSS for a while, you're probably used to the content-box box model and don't really think much about it. But, it can lead you to add extra **`div`**s from time to time. For instance, if you want to set a box's width and padding in different units of measurement from each other, like ems for the width and pixels for the padding, it's often easiest to nest another **`div`** and apply the padding to this instead, to make sure you know how much total space the box will take up. In small doses, nesting

additional **`div`**s simply to add padding or borders is not a great sin. But in complicated designs, the number of extra `div`s can really add up, which adds to both your development time and the file size of the HTML and CSS.

Setting the new **`box-sizing`** property to **`border-box`** instead of **`content-box`** solves this problem so you can get rid of all those extra **`div`**s. When a box is using the border-box box model, the browser will *subtract* the padding and border from the width of the box instead of adding it. You always know that the total space the box takes up equals the width value you've declared.



*In the traditional box model (bottom image), padding and border are added onto the declared width. By setting **`box-sizing`** to **`border-box`** (top image), the padding and border are subtracted from the declared width.*

The **box-sizing** property has good browser support, with the exception of IE 6 and IE 7. Unlike the more decorative CSS3 properties, however, lack of support for **box-sizing** could cause your entire layout to fall apart. You'll have to determine how serious the problem would be in your particular case, whether it's worth living with or hacking, or whether you should avoid using **box-sizing** for now.

CSS3 Pseudo-Classes and Attribute Selectors

Aids in: progressive enhancement, efficiency, modularity, rich typography

CSS has several really useful selectors that are only now coming into common use. Many of these are new in CSS3, but others have been around since CSS2, just not supported by all browsers (read: IE) until recently, and thus largely ignored. IE still doesn't support them all, but they can be used to add non-essential visual effects.

Taking advantage of these newer, more advanced selectors can improve your efficiency and make your pages more modular because they can reduce the need for lots of extra classes, **div**s, and **span**s to create the effects you want to see. Some selectors even make certain effects possible that you can't do with classes, such as styling the first line of a block of text differently. These types of visual effects can improve the typography of your site and aid progressive enhancement.

HTML5

Although this article is focused on modern CSS techniques, you can't have great CSS-based web pages without great markup behind them. Although HTML5 is still in development, and although debate continues about its strengths and weaknesses, some web developers are already using it in

their web pages. While HTML 4.01 and XHTML 1.0 are still great choices for the markup of your pages, it's a good idea to start learning what HTML5 has to offer so you can work with it comfortably in the future and perhaps start taking advantage of some of its features now. So, here is a brief overview of how HTML5 can help with our five modern CSS-based web design characteristics (progressive enrichment, adaptive to diverse users, modular, efficient, typographically rich).

Note: Many of these techniques are not supported in enough browsers yet to make their benefits really tangible, so think of this section as, perhaps, "here's how HTML5 *can* aid these five characteristics in the *future*."

NEW STRUCTURAL MARKUP

Aids: adaptability, modularity, efficiency

HTML5 introduces a number of new semantic elements that can add more structure to your markup to increase modularity. For instance, inside your main content **`div`** you can have several **`article`** elements, each a standalone chunk of content, and each can have its own **`header`**, **`footer`**, and heading hierarchy (**`h1`** through **`h6`**). You can further divide up an **`article`** element with **`section`** elements, again with their own **`header`**s and **`footer`**s. Having clearer, more semantic markup makes it easier to shuffle independent chunks of content around your site if needed, or syndicate them through RSS on other sites and blogs.

In the future, as user agents build features to take advantage of HTML5, these new elements could also make pages more adaptable to different user scenarios. For instance, web pages or browsers could generate table of contents based on the richer hierarchy provided by HTML5, to assist navigation within a page or across a site. Assistive technology like screen

readers could use the elements to help users jump around the page to get straight to the important content without needing “skip nav” links.

Although many of these benefits won’t be realized until some unforeseen time in the future, you can start adding these new elements now, so that as soon as tools pop up that can take full advantage of them, you’ll be ready. Even if your browser doesn’t recognize an element, you can still style it — that’s standard browser behavior. Well, in every browser but IE. Luckily, you can easily trick IE into styling these elements using a very simple piece of JavaScript, handily provided by Remy Sharp.

Of course, you usually can’t depend on all your users having JavaScript enabled, so the very safest and most conservative option is to not use these new structural elements just yet, but use **`div`**s with corresponding **`class`** names as if they were these new elements. For instance, where you would use an **`article`** element, use a **`div`** with a **`class`** name of “article.” You can still use the HTML5 doctype — HTML5 pages work fine in IE, as long as you don’t use the new elements. You can then later convert to the new HTML5 elements easily if desired, and in the meantime, you can take advantage of the more detailed HTML5 validators. Also, using these standardized class names can make updating the styles easier for both you and others in your team, and having consistent naming conventions across sites makes it easier for users with special needs to set up user style sheets that can style certain elements in a needed way.

REDUCING JAVASCRIPT AND PLUG-IN DEPENDENCE

Aids in: adaptability, efficiency

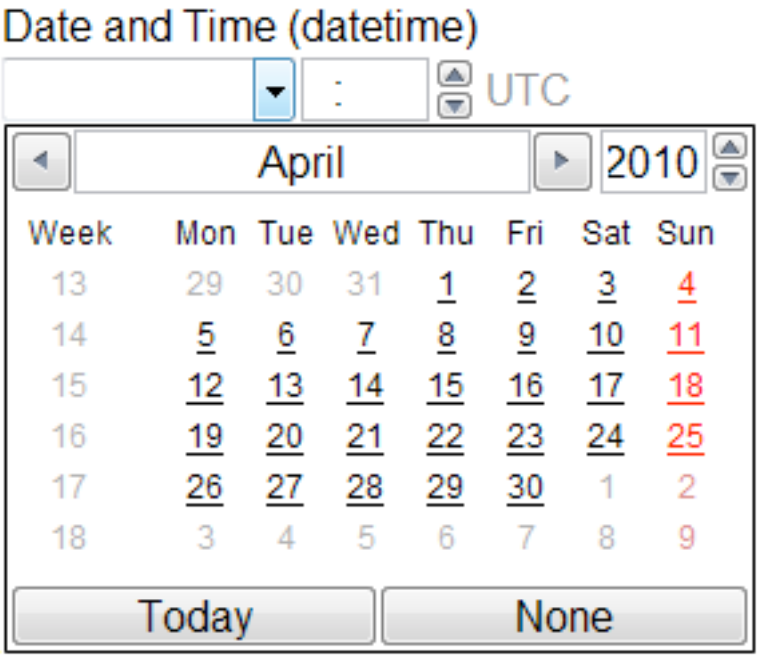
A number of the new elements and features in HTML5 make effects possible with pure markup that used to be possible only with JavaScript or

various third-party plug-ins, like Flash or Java. By removing the need for JavaScript and plug-ins, you can make your pages work on a wider variety of devices and for a wider variety of users. You may also make your development process quicker and more efficient, since you don't have to take the time to find the right script or plug-in and get it all set up. Finally, these techniques may be able to boost the speed of your pages, since extra files don't have to be downloaded by the users. (On the other hand, some may decrease performance, if the built-in browser version is slower than a third-party version. We'll have to wait and see how browsers handle each option now and in the future.)

Some of the features that reduce JavaScript and plug-in dependence are:

- **New form elements and attributes.** HTML5 offers a bunch of new **`input`** types, such as **`email`**, **`url`**, and **`date`**, that come with built-in client-side validation without the need for JavaScript. There are also many new form attributes that can accomplish what JavaScript used to be required for, like **`placeholder`** to add suggestive placeholder text to a field or **`autofocus`** to make the browser jump to a field. The new **`input`** types degrade to regular inputs in browsers that don't support them, and the new attributes are just ignored, so it doesn't hurt unsupporting browsers to start using them now.
  Of course, you'll have to put in fallback JavaScript for unsupporting browsers, negating the "no JavaScript" benefits for the time being. (Or, depend on server-side validation—which you always ought to have in place as a backup behind client-side validation anyway—to catch the submissions from unsupporting browsers.) Still, they offer a nice usability boost for users with the most up to date browsers, so they're good for progressive enhancement.

- **The `canvas` element.** The **`canvas`** element creates a blank area of the screen that you can create drawings on with JavaScript. So, it *does* require the use of JavaScript, *but* it removes the need for Flash or Java plug-ins. It's supported in every major browser but IE, but you can make it work in IE easily using the ExplorerCanvas script.
- **The `video` and `audio` elements.** HTML5 can embed **`video`** and **`audio`** files directly, just as easily as you would add an image to a page, without the need for any additional plug-ins.



Some of the new input types in HTML5 will bring up widgets, such as the calendar date picker seen with the datetime input type in Opera, without needing any JavaScript. (HTML5 input types test page)

IE Filtering

Aids in: progressive enhancement

IE 6 doesn't seem to be going away anytime soon, so if you want to really make sure your pages are progressively enhanced, you're going to have to learn how to handle it. Beyond ignoring the problem or blocking IE 6 altogether, there are a number of stances you can take:

- **Use conditional comments to fix IE's bugs:** You can create separate style sheets for each version of IE you're having problems with and make sure only that version sees its sheet. The IE sheets contain only a few rules with hacks and workarounds that the browser needs.
- **Hide all main styles from IE and feed it very minimal styles only:** This is another conditional comment method, but instead of fixing the bugs, it takes the approach of hiding all the complex CSS from IE 6 to begin with, and only feeding it very simple CSS to style text and the like. Andy Clarke calls this Universal Internet Explorer 6 CSS.
- **Use JavaScript to "fix" IE:** There are a number of scripts out there that can make IE 6 emulate CSS3, alpha-transparent PNGs, and other things that IE 6 doesn't support. Some of the most popular are ie7-js, Modernizr, and ie-css3.js.

Flexible Layouts

Aids in: adaptability

One of the main ways you can make your sites adaptable to your users' preferences is to create flexible instead of fixed-width layouts. We've already gone over how media queries can make your pages more adaptable to different viewport widths, but creating liquid, elastic, or resolution-dependent layouts can be used instead of or in conjunction with media

queries to further optimize the design for as large a segment of your users as possible.

- **Liquid layouts:** Monitor sizes and screen resolutions cover a much larger range than they used to, and mobile devices like the iPhone and iPad let the user switch between portrait and landscape mode, changing their viewport width on the fly. Liquid layouts, also called fluid, change in width based on the user's viewport (eg, window) width so that the entire design always fits on the screen without horizontal scrollbars appearing. The `min-width` and `max-width` properties and/or media queries can and should be used to keep the design from getting too stretched out or too squished at extreme dimensions.
- **Elastic layouts:** If you want to optimize for a particular number of text characters per line, you can use an elastic layout, which changes in width based on the user's text size. Again, you can use `min-` and `max-width` and/or media queries to limit the degree of elasticity.
- **Resolution-dependent layouts:** This type of layout, also called adaptive layout, is similar to media queries, but uses JavaScript to switch between different style sheets and rearrange boxes to accommodate different viewport widths.

Layout Grids

Aids in: modularity, efficiency

Designing on a grid of (usually invisible) consistent horizontal and vertical lines is not new — it goes back for centuries — but its application to web design has gained in popularity in recent years. And for good reason: a

layout grid can create visual rhythm to guide the user's eye, make the design look more clean and ordered, and enforce design consistency.

Grids can also make your designs more modular and your development more efficient because they create a known, consistent structure into which you can easily drop new elements and rearrange existing ones without as much thought and time as it would take in a non-grid layout. For instance, all of your elements must be as wide as your grid's column measurement, or some multiple of it, so you can easily move an element to another spot on the page or to another page and be assured that it will fit and look consistent with the rest of the design. At worst, you'll need to adjust the other elements' widths around it to a different multiple of the column measurements to get the new element to fit, but even this is not too work-intensive, as there is only a handful of pre-determined widths that any element can have.

All of the content of The New York Times site falls into a grid of five columns, plus a thin column on the left for navigation.

Efficient CSS Development Practices

Aids in: modularity, efficiency

Layout grids and many of the CSS3 techniques we've gone over have the side benefit of making your CSS more modular and helping you write and maintain CSS more efficiently. There are also a few CSS development practices that you can use with *any* of the techniques we've already covered in order to reduce the time it takes you to write the CSS for those techniques in the first place, as well as save you time reusing components in your pages.

CSS FRAMEWORKS

A CSS framework is a library of styles that act as building blocks to create the standard pieces you might need in your site. While CSS frameworks differ greatly in depth and breadth, most popular, publicly-distributed frameworks contain some sort of layout grid, as well as standard styles for text, navigation, forms, images, and more. It's a good idea to create your own CSS framework, perhaps based on one of the most popular ones; it can be as simple as standardizing the IDs and classes you tend to use on every project and creating a starter style sheet for yourself.

Good CSS frameworks provide you with a solid starting point for your designs, cutting down your time spent developing, testing, tweaking, and updating. They can also reduce the time others (your team members or those who inherit your sites) spend modifying your CSS, as everyone is working from a standard set of conventions. Frameworks can make your designs more modular by giving you a standard set of classes that can be reused from page to page easily, breaking the styles down into separate sheets that can be applied independently to pages on an as-needed basis, or allowing you to plug in various types of content without needing to invent new classes for it.

But, frameworks have their share of problems too. For instance, publicly-distributed (as opposed to your own private) frameworks tend to have large file sizes, as they need to work for any type of site with any type of content; if they're separated into multiple sheets, they can further damage page speed since every HTTP request takes time. We won't get into the full list of pros and cons here, but there are ways to work around many of them, so check out the following articles for the details.

OBJECT-ORIENTED CSS (OOCSS)

Nicole Sullivan coined the term object-oriented CSS (OOCSS) for her method of creating self-contained chunks of HTML (modules) that can be reused anywhere in the page or site and that any class can be applied to. Some of the main principles of OOCSS are:

- using primarily classes instead of IDs
- creating default classes with multiple, more specific classes added on to elements
- avoiding dependent selectors and class names that are location-specific

- leaving dimensions off module styles so the modules can be moved anywhere and fit
- styling containers separately from content

OOCSS aims to make your CSS development more efficient, as well as to make the CSS itself more modular and less redundant, which reduces file sizes and loading speed.

- Object Oriented CSS (the original blog post, presentation, and framework, at stubbornella.com)
- Object Oriented CSS (OOCSS): The Lowdown (TYPESETT)
- Object-Oriented CSS: What, How, and Why (Nettuts)

CSS GENERATION

When it comes to writing CSS quickly, what could be quicker than having some piece of software write it for you? Now, please don't think that I'm advocating not learning CSS and having a tool write a complete style sheet for you. That is a bad, bad idea. But, there are some quality tools out there that can give you a *headstart* with your CSS, just to shave a *little* time off the front of your CSS development process. Most good CSS generators are focused on creating styles for one particular area of your design, such as the layout structure or type styles, not the whole style sheet.

There are far too many tools to link to individually here, so remember when you're finding your own tools to carefully review the CSS it outputs. If it's invalid, bloated, or just plain ugly, don't use the tool! CSS Performance

Aids in: efficiency

Your efficiently *created* CSS-based web sites also need to *perform* as efficiently as possible for your users. Many of the CSS3 techniques we've covered can reduce file sizes and HTTP requests to increase the speed of your pages. There are some additional CSS techniques you can use to boost performance.

CSS COMPRESSION

Writing clean CSS that takes advantage of shorthand properties, grouped selectors, and other efficient syntax is nothing new, but it remains very important for improving performance. There are also tricks some CSS developers employ to further reduce CSS file sizes, such as writing each rule on one line to reduce all the line breaks. Although you can do some of this manually, there are a number of tools that can optimize and compress your CSS for you.

CSS SPRITES

CSS Sprites is a CSS technique named by Dave Shea of combining many (or all) of your site's images into one big master image and then using **`background-position`** to shift the image around to show only a single image at a time. This greatly improves your pages' performance because it greatly reduces the number of HTTP requests to your server. This is not a new technique, but it's becoming increasingly important in modern CSS-based web sites as page performance becomes more and more important.

The Apple site uses CSS sprites for various states of its navigation bar.

Not everyone is a fan of CSS sprites, however. For some opposing arguments, as well as alternative methods of reducing image HTTP requests, see:

- CSS Sprites are Stupid – Let's Use Archives Instead! (Firefox Demo) (kaioa.com)
- How to reduce the number of HTTP requests (Robert's talk)

Font Embedding and Replacement

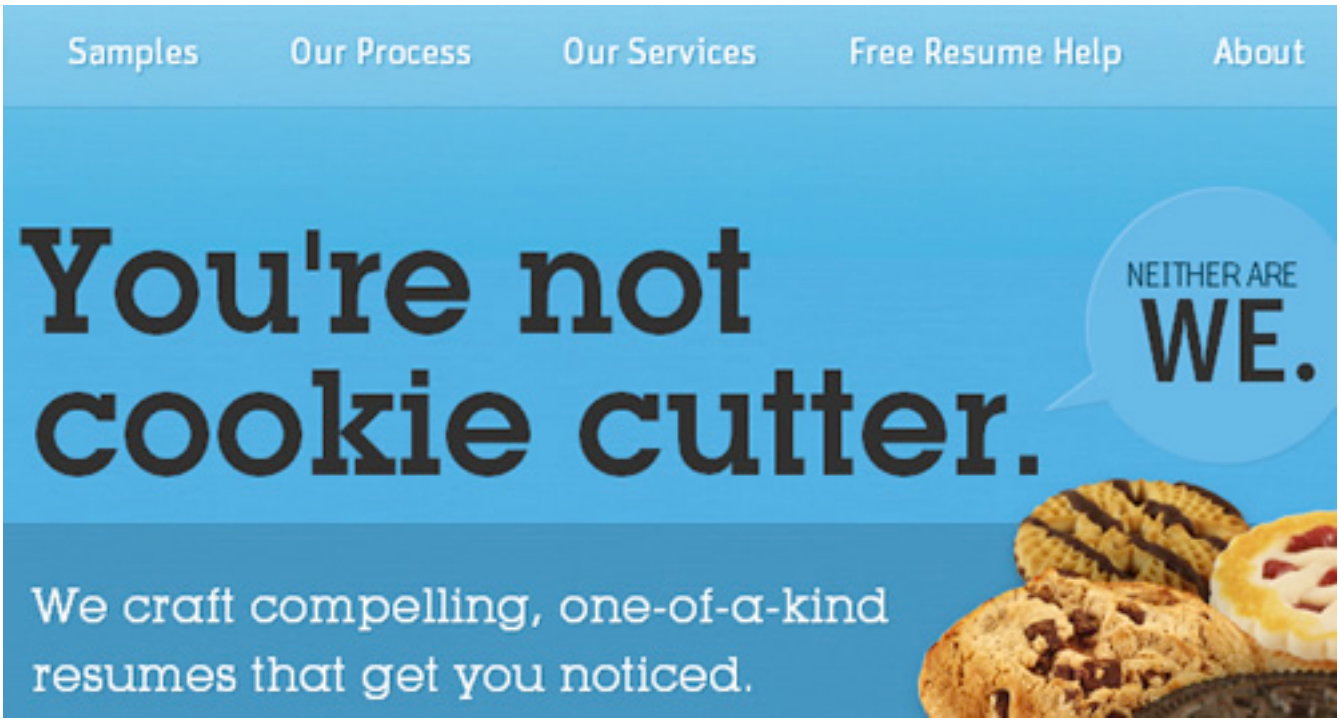
Aids in: progressive enhancement, rich typography

Until recently, web designers were limited to working with the fonts on their end users' machines. We now have a number of techniques and technologies that make unique but still readable and accessible text possible.

THE @FONT-FACE RULE

The **`@font-face rule`**, part of CSS3, allows you to link to a font on your server, called a "web font," just as you can link to images, and displays text on your site in this font. You can now make use of your beautiful, unique fonts instead of just the fonts that most people already have installed on their machines. Fortunately, **`@font-face`** has good browser support. But alas, it's not as simple as that. Different browsers support different types of fonts, different platforms and browsers anti-alias very differently, you can get a flash of unstyled text before the font loads, your font may not allow **`@font-face`** embedding in its license, and on and on it goes.

Sam Howat's site uses @font-face to get attractive non-standard fonts into the headings and intro blocks of text.



Blue Sky Resumes uses @font-face extensively in headings, feature copy, and the main nav bar of the site.

OTHER FONT EMBEDDING AND REPLACEMENT TECHNIQUES

If the pure CSS solution of **`@font-face`** is making your head spin, you can use a font embedding service or font replacement technique.

- **Font embedding services:** There are a number of third-party font embedding services available that make use of **`@font-face`**, such as Typekit and Kernest, but make implementation easier by helping you work around the browser differences. They also all get around the legal issue of font embedding by providing you with a set of fonts that are licensed for this type of use and impossible or difficult for end users to steal. Most of these services are not free, but some have free options that give you access to a limited set of fonts.

- **Font replacement techniques:** These free techniques, such as sIFR and Cufón, do not make use of **`@font-face`,** but instead use scripting and/or Flash to display fonts that are not on the user's machine. None of them directly address the licensing issue, but none of them link directly to ready-to-use fonts, so copyright legality is not clear-cut.

Conclusion

You're now equipped with the basic knowledge and a slew of links to create modern CSS-based web pages that are progressively enriched, adaptive to diverse users, modular, efficient, and typographically rich. Go out and create great, modern work!

Writing CSS For Others

Harry Roberts

I think a lot of us CSS authors are doing it wrong. We are selfish by nature; we get into our little bubbles, writing CSS (as amazing as it may be) with only ourselves in mind. How many times have you inherited a CSS file that's made you say "WTF" at least a dozen times?

(Image: Toca Boca)

HTML has a standard format and syntax that everyone understands. For years, programmers have widely agreed on standards for their respective languages. CSS doesn't seem to be there yet: everyone has their own favorite format, their own preference between single-line and multi-line, their own ideas on organization, and so on.

A New Way of Thinking

Recently, I have begun to think that CSS authors could take a leaf from the programmers' book. We need to write CSS that others can understand and use with ease. Programmers have been writing sharable code since day one, and it's high time that CSS be written with as much organization and openness.

In writing inuit.css and working on a huge front-end framework at my job, it has become more apparent to me that writing code that can be easily picked up by others is extremely important. I wouldn't say that I've nailed everything yet, but I'll share with you some things that I think are vital when writing code, specifically CSS, that will be used by others.

First, the reasoning: my number one tip for developers is to always code like you're working in a team, even when you're not. You may be the only developer on your project right now, but it might not stay that way:

- Your project could be taken to another developer, agency or team. Even though this is not the best situation to find yourself in, handing over your work smoothly and professionally to others is ideal.
- If you're doing enough work to warrant employing someone else or expanding the team at all, then your code ceases to be yours and becomes the team's.

- You could leave the company, take a vacation or be off sick, at which point someone else will inherit your code, even if only temporarily.
- Someone will inevitably poke through your source code, and if they've never met you, this could be the only basis on which they judge your work. First impressions count!

Comments Are King!

One thing I've learned from building a massive front-end framework at work and from producing inuit.css is that comments are vital. Comments, comments, comments. Write one line of code, then write *about* it. N.B. This is not meant to mean write about *every* line of code, as that would be overkill. Only comment where it helps/is useful.

It might seem like overkill at first, but write about everything you do. The code might look simple to you, but there's bound to be someone out there who has no idea what it does. Write it down. I had already gotten into this habit when I realized that this was the same technique that a good friend and incredibly talented developer, Nick Payne, told me about. That technique is called "rubber-duck debugging":

> *... an unnamed expert programmer would keep a rubber duck by his desk at all times, and debug his code by forcing himself to explain it, line by line, to the duck.*

Write comments like you're talking to a rubber duck!

Good comments take care of 99% of what you hand over and — more importantly — take care of your documentation. Your code should *be* the documentation.

Comments are also an excellent way to show off. Ever wanted to tell someone how awesome a bit of your code is but never found the chance? This is that chance! Explain how clever it is, and just wait for people to read it.

Egos aside, though, comments do force you to write nicer code. I've found that writing extensive comments has made me a better developer. I write cleaner code, because writing comments reminds me that I'm intending for others to read the code.

Multi-Line CSS

This issue really divides developers: single-line versus multi-line CSS. I've always written multi-line CSS. I love it and despise single-line notation. But others think the opposite — and they're no more right or wrong than I am. Taste is taste, and consistency is what matters.

Having said that, when working on a team, I firmly believe that multi-line CSS is the way to go. Multi-line ensures that each CSS declaration is accounted for. One line represents one piece of functionality (and can often be attributed to one person).

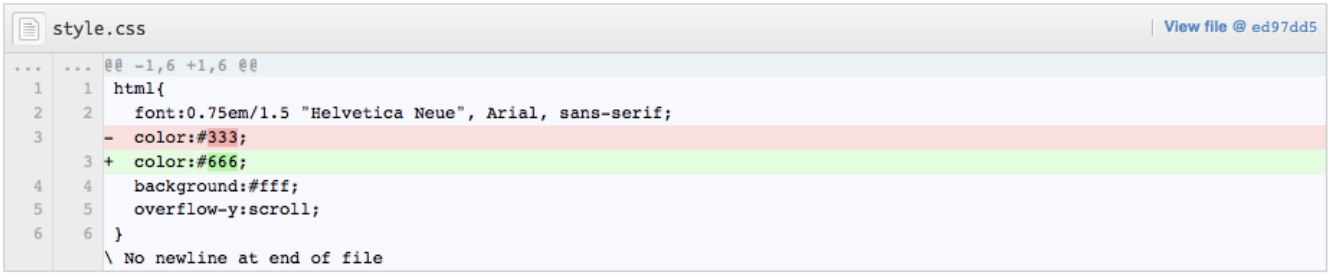
As a result, each line will show up individually on a diff between two versions. If you change, say, only one hex value in a `color` declaration, then that is all that needs to be flagged. A diff on a single-line document would flag an entire rule set as having been changed, even when it hasn't.

Take the following example:

style.css View file @ 55519a1

```
... ... @@ -1 +1 @@
1       -html  { font:0.75em/1.5 "Helvetica Neue", Arial, sans-serif; color:#333; background:#fff; overflow-y:scroll; }
        \ No newline at end of file
    1   +html  { font:0.75em/1.5 "Helvetica Neue", Arial, sans-serif; color:#666; background:#fff; overflow-y:scroll; }
        \ No newline at end of file
```

Above, we just changed a **`color`** value in a rule set, but because it was a single-line CSS file, the entire rule set appears to have changed. This is very misleading, and also not very readable or obvious. At first glance, it appears that a whole rule set has been altered. But look closely and you'll see that only **`#333`** has been changed to **`#666.`** We can make this distinction far more obvious by using multi-line CSS, like so:



Having said all this, I am by no means a version-control expert. I've only just started using GitHub for inuit.css, so I'm very new to it all. Instead, I'll leave you with Jason Cale's excellent article on the subject.

Furthermore, single-line CSS makes commenting harder. Either you end up with one comment per rule set (which means your comments might be less specific than had they been done per line), or you get a messy single line of comment, then code, then comment again, as shown here:

With multi-line CSS, you have a much neater comment structure:

100644 | 6 lines (6 sloc) | 0.281 kb raw | blame | history

```
1  html{
2          font:0.75em/1.5 "Helvetica Neue", Arial, sans-serif; /* 12px sans with an 18px line-height */
3          color:#666; /* Light grey is nicer to read than black */
4          background:#fff; /* Always define a background colour */
5          overflow-y:scroll; /* Force scrollbars to avoid page jumps */
6  }
```

Ordering CSS Properties

Likewise, the order in which people write their CSS properties is very personal.

Many people opt for alphabetized CSS, but this is counter-intuitive. I commented briefly on the subject on GitHub; my reasoning is that ordering something by a meaningless metric makes no sense; the initial letter of a declaration has no bearing on the declaration itself. Ordering CSS alphabetically makes as much sense as ordering CDs by how bright their covers are.

A more sensible approach is to order by type and relevance. That is, group your color declarations together, your box-model declarations together, your font declarations together and so on. Moreover, order each grouping according to its relevance to the selector. If you are styling an `h1`, then put font-related declarations first, followed by the others. For example:

```
#header {
   /* Box model */
   width: 100%;
   padding: 20px 0;
   /* Color */
   color: #fff;
   background: #333;
```

```
}
h1 {
   /* Font */
   font-size: 2em;
   font-weight: bold;
   /* Color */
   color: #c00;
   background: #fff;
   /* Box model */
   padding: 10px;
   margin-bottom: 1em;
}
```

Please note that the comments above are not intended to go into your CSS file, but are just to illustrate my point in the article.

Ordering CSS Files

Ordering CSS files is always tricky, and there is no right or wrong way. A good idea, though, is to section the code into defined groups, with headings, as well as a table of contents at the top of the file. Something like this:

```
/*-------------------------------------*\
   CONTENTS
\*-------------------------------------*/
/*
MAIN
TYPE
IMAGES
TABLES
MISC
RESPONSIVE
*/
/*-------------------------------------*\
```

```
   $MAIN
\*-------------------------------------*/
html {
   styles
}
body {
   styles
}
/*-------------------------------------*\
   $TYPE
\*-------------------------------------*/
```

And so on.

This way, you can easily read the contents and jump straight to a particular section by performing a quick search (Command/Control + F). Prepending each heading with a dollar sign makes it unique, so that a search will yield only headings.

THE “SHARED” SECTION

All CSS files should have a section for sharing, where you tether selectors to a single declaration, rather than write the same declaration over and over.

So, instead of writing this…

```
/*-------------------------------------*\
   $TYPE
\*-------------------------------------*/
h1 {
   font-size: 2em;
   color: #c00;
}
h2 {
   font-size: 1.5em;
   color: #c00;
```

```
}
a {
   color: #c00;
   font-weight: bold;
}
#tagline {
   font-style: italic;
   color: #c00;
}
```

... you would write this:

```
/*-------------------------------------*\
   $TYPE
\*-------------------------------------*/
h1 {
   font-size: 2em;
}
h2 {
   font-size: 1.5em;
}
a {
   font-weight: bold;
}
#tagline {
   font-style: italic;
}
/*-------------------------------------*\
   $SHARED
\*-------------------------------------*/
h1, h2, a, #tagline {
   color:#c00;
}
```

This way, if the brand color of **#c00** ever changes, a developer would only ever need to change that value once. This is essentially using variables in CSS.

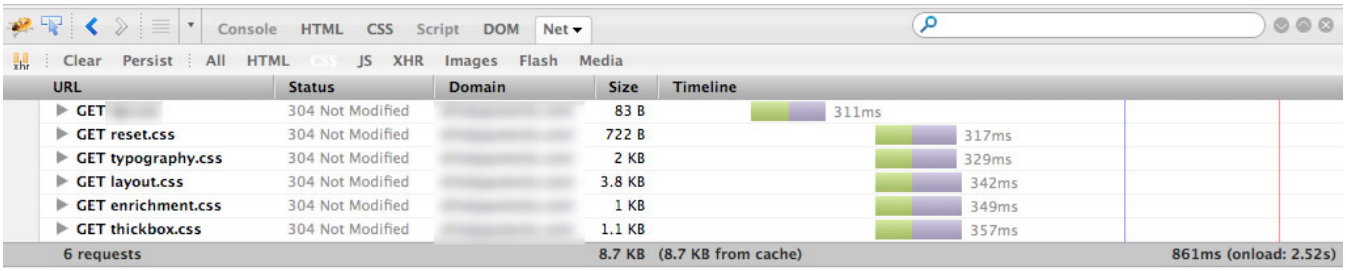
MULTIPLE CSS FILES FOR SECTIONS, OR ONE BIG FILE WITH ALL SECTIONS?

A lot of people separate their sections into multiple files, and then use the **@import** rule to put them all back together in one meta file. For example:

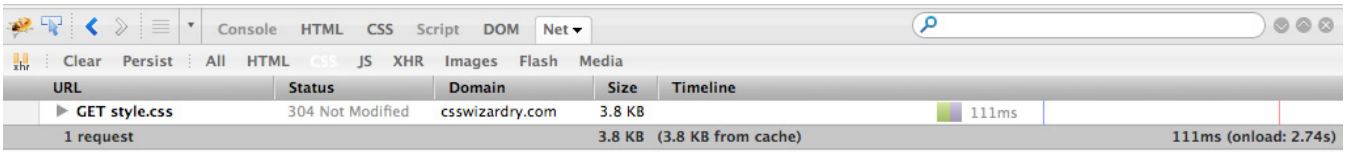
```
@import url(main.css)
@import url(type.css)
@import url(images.css)
@import url(tables.css)
@import url(misc.css)
@import url(responsive.css)
```

This is fine, and it does keep everything in sections, but it does lead to a lot more HTTP requests than is necessary; and minimizing requests is one of the most important rules for a high-performance website.

Compare this...



... to this:



If you section and comment your CSS well enough, using a table of contents and so forth, then you avoid the need to split up your CSS files, thus keeping those requests down.

If you really want to break up your CSS into multiple style sheets, you can do that—just combine them into one at build time. This way, your developers can work across multiple files, but your users will download one concatenated file.

Learning From Programmers

Programmers have been doing this for ages, and doing it well. Their job is to write code that is as readable as it is functional. We front-end developers could learn a lot from how programmers deal with code.

The code of my good friend (and absolutely awesome chap) Dan Bentley really struck a chord with me. It's beautiful. I don't understand what it does most of the time, but it's so clean and lovely to look at. So much white space, so neat and tidy, all commented and properly looked after. His PHP, Ruby, Python or whatever-he-decides-to-use-that-day always looks so nice. It made me want to write my CSS the same way.

White space is your friend. You can remove it before you go live if you like, but the gains in cleanliness and readability are worth the few extra bytes (which you could always cut back down on by Gzip'ing your files anyway).

Make the code readable and maintainable first and foremost, then worry about file size later. Happy developers are worth more than a few kilobytes in saved weight.

Code Should Take Care Of Itself

So far, we've talked about people maintaining your code, but what about actually using it?

You can do a number of things to make the life of whoever inherits your code much easier—and to make you look like a saint. I can't think of many generic examples, but I have a few specific ones, mainly from inuit.css.

INTERNATIONALIZE YOUR SELECTORS

Inuit.css has a class named **`.centered`**, which is spelt in US English. CSS is written in US English anyway, so we're used to this; but as an English developer, I always end up typing UK English at least once in a project. Here is the way I have accounted for this:

```
.centred, .centered {
   [style]
}
```

Both classes do the same thing, but the next developer doesn't have to remember to be American!

If your selectors include words that have US and UK spelling variants, include both.

LET THE CODE DO THE HEAVY LIFTING

Also in inuit.css, I devised a method to not need **`class="end"`** for the last column in a row of grids. Most frameworks require this class, or something similar, to omit the **`margin-right`** on the last item in a list and thus prevent the grid system from breaking.

Remembering to add this class isn't a big deal, but it's still one more thing to remember. So, I worked out a way to remove it.

In another major inuit.css update, I removed a **`.grid`** class that used to be required for every single grid element. This was a purely functional class that developers had to add to any **`<div>`** that they wanted to behave like a grid column. For example:

```
<div class="grid grid-4">
    …
</div>
```

This **`.grid`** class, in conjunction with the **`.grid-4`** class, basically says, "I want this **`<div>`** to be a grid item and to span four columns." This isn't a huge burden on developers, but again, it's one more thing that could be removed to make their lives easier.

The solution was to use a regex CSS attribute selector: **`[class^="grid-"]{}`**. This says, “Select any element whose class begins with **`.grid-`**,” and it allows the developer’s mark-up to now read as follows:

```
<div class="grid-4">
    …
</div>
```

CSS attribute selectors may be less efficient than, say, classes, but not to the point that you’d ever notice it (unless you were working on a website with massive traffic, like Google). The benefits of making the mark-up leaner and the developer’s life easier far outweigh the performance costs.

Do the hard work once and reap the benefits later. Plus, get brownie points from whoever picks up your project from you.

BE PRE-EMPTIVE, THINK ABOUT EDGE CASES

An example of being pre-emptive in inuit.css is the grid system. The grids are meant to be used in a series of sibling columns that are all contained in one parent, with a class of **5**.

This is their intended use. But what if someone wants a standalone grid? That’s not their intended use, but let’s account for it should it happen.

Note: inuit.css has changed since this was written, but the following is true as of version 2.5.

Another and perhaps better example is the 12-column grid system in inuit.css. By default, the framework uses a 16-column grid, with classes **`.grid-1`** through **`.grid-16`** for each size of column.

A problem arises, though, when you attempt to switch from the 16-column system to the 12-column system. The **`.grid-15`** class, for example, doesn't exist in a 12-column layout; and even if it did, it would be too big.

What I did here was to make **`.grid-13`** through **`.grid-16`** use the exact same properties as **`.grid-12`**. So, if you switch from a 16-column layout to a 12-column one, your **`.grid-13`** through **`.grid-16`** would't break it — they would all just become **`.grid-12`**s.

This means that developers can switch between them, and inuit.css will take care of everything else.

Pre-empt the developer's next problem, and solve it for them.

Addendum

A few people have mentioned that some of these practices lead to a bloated CSS file. Bloat is where *unnecessary* code makes its way into a file, and your comments should be anything but unnecessary. It *will* lead to a larger CSS file, but not a bloated one.

If you are worried about increased file size then you should minify and gzip your stylesheets, but this is best practice (particularly for highly trafficked websites) anyway.

That's It

There you have it: a few humble suggestions on how CSS authors can write code that is perfect for other developers to inherit, understand, maintain, extend and enjoy.

Decoupling HTML From CSS

Jonathan Snook

For years, the Web standards community has talked about the separation of concerns. Separate your CSS from your JavaScript from your HTML. We all do that, right? CSS goes into its own file; JavaScript goes in another; HTML is left by itself, nice and clean.

CSS Zen Garden proved that we can alter a design into a myriad of permutations simply by changing the CSS. However, we've rarely seen the flip side of this — the side that is more likely to occur in a project: the HTML changes. We modify the HTML and then have to go back and revise any CSS that goes with it.

In this way, we haven't really separated the two, have we? We have to make our changes in two places.

Exploring Approaches

Over the course of my career, I've had the pleasure and privilege to work on hundreds of different websites and Web applications. For the vast majority of these projects, I was the sole developer building out the HTML and CSS. I developed a way of coding websites that worked well for me.

Most recently, I spent two years at Yahoo working on Mail, Messenger, Calendar and other projects. Working on a much larger project with a much larger team was a great experience. A small team of prototypers worked with a larger team of designers to build out all of the HTML and CSS for multiple teams of engineers.

It was the largest-scale project I had worked on in many aspects:

- Yahoo's user base is massive. Mail alone has about 300 million users.
- Hundreds of people spread across multiple teams were working with the HTML and CSS.
- We were developing a system of components to work across multiple projects.

It was during my time at Yahoo that I began to really examine how I and the team at Yahoo build websites. What pain points did we keep running into, and how could we avoid them?

I looked to see what everyone else was doing. I looked at Nicole Sullivan's Object-Oriented CSS, Jina Bolton's presentation on "CSS Workflow" and Natalie Downe's "Practical, Maintainable CSS," to name just a few.

I ended up writing my thoughts as a long-form style guide named "Scalable and Modular Architecture for CSS." That sounds wordy, so you can just call it SMACSS (pronounced "smacks") for short. It's a guide that continues to evolve as I refine and expand on ways to approach CSS development.

As a result of this exploration, I've noticed that designers (including me) traditionally write CSS that is deeply tied to the HTML that it is designed to style. How do we begin to decouple the two for more flexible development with less refactoring?

In other words, how do we avoid throwing `!important` at everything or falling into selector hell?

Reusing Styles

In the old days, we wrapped **`font`** tags and applied **`background`** attributes to every HTML element that needed styling. This was, of course, very impractical, and thus CSS was born. CSS enabled us to reuse styles from one part of the page on another.

For example, a navigation menu has a list of items that all look the same. Repeating inline styles on each item wouldn't be practical. As a result, we begin to see CSS like this:

```
#nav {
   margin: 0;
   padding: 0;
   list-style: none;
}
#nav li {
   float: left;
}
#nav li a {
   display: block;
   padding: 5px 10px;
   background-color: blue;
}
```

Sure beats adding float:left to every list item and display:block; padding:5px 10px; to every link. Efficiency, for the win! Just looking at this, you can see the HTML structure that is expected:

```
<ul id="nav">
   <li><a href="/">Home</a></li>
   <li><a href="/products">Products</a></li>
   <li><a href="/contact">Contact Us</a></li>
</ul>
```

Now, the client comes back and says, “I want a drop-down menu to appear when the user clicks on ‘Products.’ Give them easy access to each of the pages!” As a result, our HTML changes.

```
<ul id="nav">
   <li><a href="/">Home</a></li>
   <li><a href="/products">Products</a>
      <ul>
         <li><a href="/products/shoes">Shoes</a></li>
         <li><a href="/products/jackets">Jackets</a></li>
      </ul>
   </li>
   <li><a href="/contact">Contact Us</a></li>
</ul>
```

We now have a list item within a list item, and links within it. Our menu has a horizontal navigation when the client wants a vertical list, so we add some rules to style the inner list to match what the client wants.

```
#nav ul {
   margin: 0;
   padding:0;
   list-style:none;
}
#nav li li {
   float: none;
}
#nav li li a {
   padding: 2px;
   background-color: red;
}
```

Problem solved! Sort of.

Reducing The Depth Of Applicability

Probably the most common problem with CSS is managing specificity. Multiple CSS rules compete in styling a particular element on the page. With our menu, our initial rules were styling the list items and the links in the navigation and the menu. Not good.

By adding more element selectors, we were able to increase specificity and have our menu styles win out over the navigation styles.

However, this can become a game of cat and mouse as a project increases in complexity. Instead, we should be limiting the impact of CSS. Navigation styles should apply to and affect only the elements that pertain to it, and menu styles should apply to and affect only the elements that pertain to it.

I refer to this impact in SMACSS as the "depth of applicability." It's the depth at which a particular rule set impacts the elements around it. For example, a style like **`#nav li a`**, when given an HTML structure that includes our menus, has a depth of 5: from the **`ul`** to the **`li`** to the **`ul`** to the **`li`** to the **`a`.**

The deeper the level of applicability, the more impact the styles can have on the HTML and the more tightly coupled the HTML is to the CSS.

The goal of more manageable CSS — especially in larger projects — is to limit the depth of applicability. In other words, write CSS to affect only the elements that we want them to affect.

CHILD SELECTORS

One tool for limiting the scope of CSS is the child selector (>). If you no longer have to worry about Internet Explorer 6 (and, thankfully, many of us don't), then the child selector should be a regular part of your CSS diet.

Child selectors limit the scope of selectors. Going back to our navigation example, we can use the child selector to limit the scope of the navigation so that it does not affect the menu.

```
#nav {
   margin: 0;
   padding: 0;
   list-style: none;
}
#nav > li {
   float: left;
}
#nav > li > a {
   display: block;
   padding: 5px 10px;
   background-color: blue;
}
```

For the menu, let's add a class name to it. This will make it more descriptive and provide a base for the rest of our styles.

```
.menu {
   margin: 0;
   padding: 0;
   list-style: none
}
.menu > li > a {
   display: block;
   padding: 2px;
   background-color: red;
}
```

What we've done is limited the scope of our CSS and isolated two visual patterns into separate blocks of CSS: one for our navigation list and one for our menu list. We've taken a small step towards modularizing our code and a step towards decoupling the HTML from the CSS.

Classifying Code

Limiting the depth of applicability helps to minimize the impact that a style might have on a set of elements much deeper in the HTML. However, the other problem is that as soon as we use an element selector in our CSS, we are depending on that HTML structure never to change. In the case of our navigation and menu, it's always a list with a bunch of list items, with a link inside each of those. There's no flexibility to these modules.

Let's look at an example of something else we see in many designs: a box with a heading and a block of content after it.

```
<div class="box">
   <h2>Sites I Like</h2>
   <ul>
      <li><a href="http://smashingmagazine.com/">Smashing
Magazine</a></li>
      <li><a href="http://smacss.com/">SMACSS</a></li>
   </ul>
</div>
Let's give it some styles.
.box {
   border: 1px solid #333;
}
.box h2 {
   margin: 0;
   padding: 5px 10px;
   border-bottom: 1px solid #333;
   background-color: #CCC;
```

```
}
.box ul {
   margin: 10px;
}
```

The client comes back and says, “That box is great, but can you add another one with a little blurb about the website?”

```
<div class="box">
   <h2>About the Site</h2>
   <p>This is my blog where I talk about only the bestest
things in the whole wide world.</p>
</div>
```

In the previous example, a margin was given to the list to make it line up with the heading above it. With the new code example, we need to give it similar styling.

```
.box ul, .box p {
   margin: 10px;
}
```

That’ll do, assuming that the content never changes. However, the point of this exercise is to recognize that websites can and do change. Therefore, we have to be proactive and recognize that there are alternative elements we might want to use.

For greater flexibility, let’s use classes to define the different parts of this module:

```
.box .hd { }  /* this is our box heading */
.box .bd { }  /* this is our box body */
```

When applied to the HTML, it looks like this:

```
<div class="box">
   <h2 class="hd">About the Site</h2>
```

```
   <p class="bd">This is my blog where I talk about only the
bestest things in the whole wide world.</p>
</div>
```

CLARIFYING INTENT

Different elements on the page could have a heading and a body. They're "protected" in that they're a child selector of **box**. But this isn't always as evident when we're looking at the HTML. We should clarify that these particular **hd** and **bd** classes are for the **box** module.

```
.box .box-hd {}
.box .box-bd {}
```

With this improved naming convention, we don't need to combine the selectors anymore in an attempt to namespace our CSS. Our final CSS looks like this:

```
.box {
border: 1px solid #333;
}
.box-hd {
margin: 0;
padding: 5px 10px;
border-bottom: 1px solid #333;
background-color: #CCC;
}
.box-bd {
   margin: 10px;
}
```

The bonus of this is that each of these rules affects only a single element: the element that the class is applied to. Our CSS is easier to read and easier to debug, and it's clear what belongs to what and what it does.

It's All Coming Undone

We've just seen two ways to decouple HTML from CSS:

1. Using child selectors,
2. Using class selectors.

In addition, we've seen how naming conventions allow for clearer, faster, simpler and more understandable code.

These are just a couple of the concepts that I cover in "Scalable and Modular Architecture for CSS," and I invite you to read more.

POSTSCRIPT

In addition to the resources linked to above, you may wish to look into BEM, an alternative approach to and framework for building maintainable CSS. Mark Otto has also been documenting the development of Twitter Bootstrap, including the recent article "Stop the Cascade," which similarly discusses the need to limit the scope of styles.

CSS Specificity And Inheritance

Inayaili de Leon

CSS' barrier to entry is extremely low, mainly due to the nature of its syntax. Being clear and easy to understand, the syntax makes sense even to the inexperienced Web designer. It's so simple, in fact, that you could style a simple CSS-based website within a few hours of learning it.

But this apparent simplicity is deceitful. If after a few hours of work, your perfectly crafted website looks great in Safari, all hell might break loose if you haven't taken the necessary measures to make it work in Internet Explorer. In a panic, you add hacks and filters where only a few tweaks or a different approach might do. Knowing how to deal with these issues comes with experience, with trial and error and with failing massively and then learning the correct way.

Understanding a few often overlooked concepts is also important. The concepts may be hard to grasp and look boring at first, but understanding them and knowing how to take advantage of them is important.

Two of these concepts are *specificity* and *inheritance*. Not very common words among Web designers, are they? Talking about **`border-radius`** and **`text-shadow`** is a lot more fun; but specificity and inheritance are fundamental concepts that any person who wants to be good at CSS should understand. They will help you create clean, maintainable and flexible style sheets. Let's look at what they mean and how they work.

The notion of a *“cascade”* is at the heart of CSS (just look at its name). It ultimately determines which properties will modify a given element. The cascade is tied to three main concepts: importance, specificity and source order. The cascade follows these three steps to determine which properties to assign to an element. By the end of this process, the cascade has assigned a *weight* to each rule, and this weight determines which rule takes precedence, when more than one applies.

1. Importance

Style sheets can have a few different sources:

1. **User agent**
   For example, the browser’s default style sheet.
2. **User**
   Such as the user’s browser options.
3. **Author**
   This is the CSS provided by the page (whether inline, embedded or external)

By default, this is the order in which the different sources are processed, so the author’s rules will override those of the user and user agent, and so on.

There is also the `!important` declaration to consider in the cascade. This declaration is used to balance the relative priority of user and author style sheets. While author style sheets take precedence over user ones, if a user rule has `!important` applied to it, it will override even an author rule that also has `!important` applied to it.

Knowing this, let's look at the final order, in ascending order of importance:

1. User agent declarations,
2. User declarations,
3. Author declarations,
4. Author `!important` declarations,
5. User `!important` declarations.

This flexibility in priority is key because it allows users to override styles that could hamper the accessibility of a website. (A user might want a larger font or a different color, for example.)

2. Specificity

Every CSS rule has a particular weight (as mentioned in the introduction), meaning it could be more or less important than the others or equally important. This weight defines which properties will be applied to an element when there are conflicting rules.

Upon assessing a rule's importance, the cascade attributes a specificity to it; if one rule is more specific than another, it overrides it.

If two rules share the same weight, source and specificity, the later one is applied.

2.1 HOW TO CALCULATE SPECIFICITY?

There are several ways to calculate a selector's specificity.

The quickest way is to do the following. Add 1 for each element and pseudo-element (for example, **`:before`** and **`:after`**); add 10 for each attribute (for example, **`[type="text"]`**), class and pseudo-class (for example, **`:link`** or **`:hover`**); add 100 for each ID; and add 1000 for an inline style.

Let's calculate the specificity of the following selectors using this method:

- **p.note**
  1 class + 1 element = 11
- **#sidebar p[lang="en"]**
  1 ID + 1 attribute + 1 element = 111
- **body #main .post ul li:last-child**
  1 ID + 1 class + 1 pseudo-class + 3 elements = 123

A similar method, described in the W3C's specifications, is to start with a=0, b=0, c=0 and d=0 and replace the numbers accordingly:

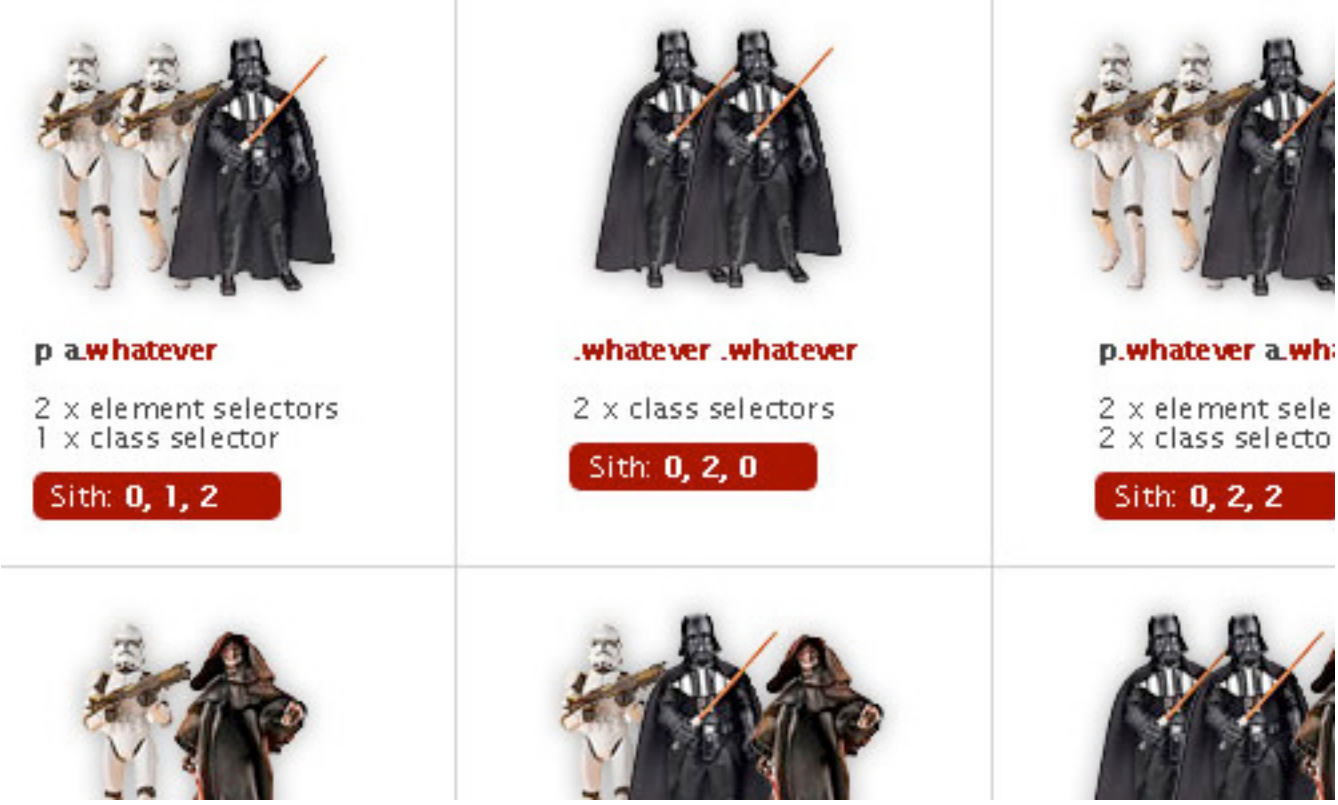
- a = 1 if the style is inline,
- b = the number of IDs,
- c = the number of attribute selectors, classes and pseudo-classes,
- d = the number of element names and pseudo-elements.

Let's calculate the specificity of another set of selectors:

- <p style="color:#000000;">
  a=1, b=0, c=0, d=0 → 1000

- footer nav li:last-child
  a=0, b=0, c=1, d=3 → 0013
- **#sidebar input:not([type="submit"])**
  a=0, b=1, c=1, d=1 → 0111
  (Note that the negation pseudo-class doesn't count, but the selector inside it does.)

If you'd rather learn this in a more fun way, Andy Clarke drew a clever analogy between specificity and Star Wars back in 2005, which certainly made it easier for Star Wars fans to understand specificity. Another good explanation is "CSS Specificity for Poker Players," though slightly more complicated.



Andy Clarke's CSS Specificity Wars chart.

Remember that non-CSS presentational markup is attributed with a specificity of 0, which would apply, for example, to the `font` tag.

Getting back to the **`!important`** declaration, keep in mind that using it on a shorthand property is the same as declaring all of its sub-properties as **`!important`** (even if that would revert them to the default values).

If you are using imported style sheets (**`@import`**) in your CSS, you have to declare them before all other rules. Thus, they would be considered as coming before all the other rules in the CSS file.

Finally, if two selectors turn out to have the same specificity, the last one will override the previous one(s).

2.2 MAKING SPECIFICITY WORK FOR YOU

If not carefully considered, specificity can come back to haunt you and lead you to unwittingly transform your style sheets into a complex hierarchy of unnecessarily complicated rules.

You can follow a few guidelines to avoid major issues:

- When starting work on the CSS, use generic selectors, and add specificity as you go along;
- Using advanced selectors doesn't mean using unnecessarily complicated ones;
- Rely more on specificity than on the order of selectors, so that your style sheets are easier to edit and maintain (especially by others).

A good rule of thumb can be found in Jim Jeffers' article, "The Art and Zen of Writing CSS":

> *Refactoring CSS selectors to be less specific is exponentially more difficult than simply adding specific rules as situations arise.*

3. Inheritance

A succinct and clear explanation of inheritance is in the CSS3 Cascading and Inheritance module specifications (still in "Working draft" mode):

> *Inheritance is a way of propagating property values from parent elements to their children.*

Some CSS properties are inherited by the children of elements by default. For example, if you set the **`body`** tag of a page to a specific font, that font will be inherited by other elements, such as headings and paragraphs, without you having to specifically write as much. This is the magic of inheritance at work.

The CSS specification determines whether each property is inherited by default or not. Not all properties are inherited, but you can force ones to be by using the **`inherit`** value.

3.1 OBJECT-ORIENTED PROGRAMMING INHERITANCE

Though beyond the scope of this article, CSS inheritance shouldn't be confused with object-oriented programming (OOP) inheritance. Here is the definition of OOP inheritance from Wikipedia, and it makes clear that we are *not* talking about the same thing:

> *In object-oriented programming (OOP), inheritance is a way to form new classes [...] using classes that have already been defined. Inheritance is employed to help reuse existing code with little or no modification. The new classes [...] inherit attributes and behavior of the pre-existing classes. ...*

3.2 HOW INHERITANCE WORKS

When an element inherits a value from its parent, it is inheriting its computed value. What does this mean? Every CSS property goes through a four-step process when its value is being determined. Here's an excerpt from the W3C specification:

> *The final value of a property is the result of a four-step calculation: the value is determined through specification (the "specified value"), then resolved into a value that is used for inheritance (the "computed value"), then converted into an absolute value if necessary (the "used value"), and finally transformed according to the limitations of the local environment (the "actual value").*

IN OTHER WORDS:

1. **Specified value**
   The user agent determines whether the value of the property comes from a style sheet, is inherited or should take its initial value.

2. **Computed value**
   The specified value is resolved to a computed value and exists even when a property doesn't apply. The document doesn't have to be laid out for the computed value to be determined.

3. **Used value**
   The used value takes the computed value and resolves any dependencies that can only be calculated after the document has been laid out (like percentages).

4. **Actual value**
   This is the value used for the final rendering, after any approximations have been applied (for example, converting a decimal to an integer).

If you look at any CSS property's specification, you will see that it defines its initial (or default) value, the elements it applies to, its inheritance status and its computed value (among others). For example, the **`background-color`** specification states the following:

> *Name: background-color*
> *Value: <color>*
> *Initial: transparent*
> *Applies to: all elements*
> *Inherited: no*
> *Percentages: N/A*

Media: visual
Computed value: the computed color(s)

Confusing? It can be. So, what do we need to understand from all this? And why is it relevant to inheritance?

Let's go back to the first sentence of this section, which should make more sense now. *When an element inherits a value from its parent, it inherits its computed value*. Because the computed value exists even if it isn't specified in the style sheet, a property can be inherited even then: the initial value will be used. So, you can make use of inheritance even if the parent doesn't have a specified property.

3.3 USING INHERITANCE

The most important thing to know about inheritance is that it's there and how it works. If you ignore the jargon, inheritance is actually very straightforward.

Imagine you had to specify the **`font-size`** or **`font-family`** of every element, instead of simply adding it to the **`body`** element? That would cumbersome, which is why inheritance is so helpful.

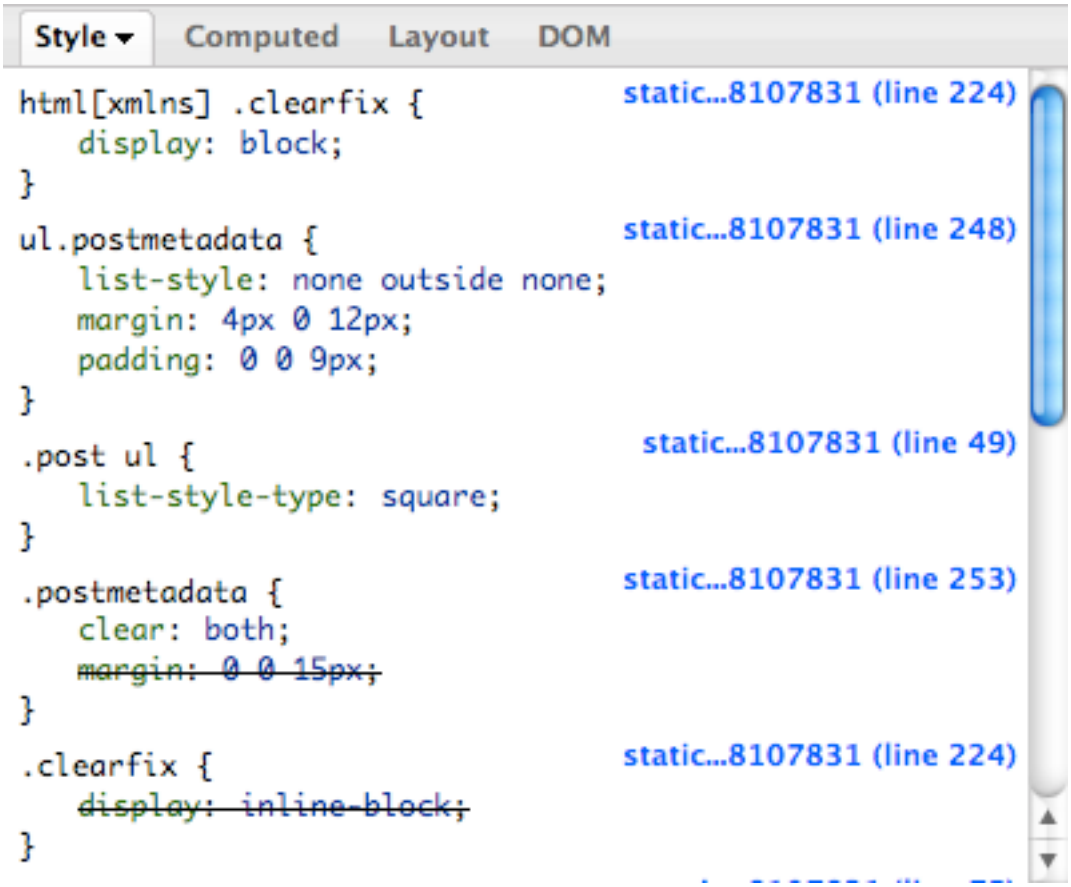
Don't break it by using the universal selector (`*`) with properties that inherit by default. Bobby Jack wrote an interesting post about this on his Five-Minute Argument blog. You don't have to remember all of the properties that inherit, but you will in time.

Rarely does a CSS-related article not bring some kind of bad news about Internet Explorer. This article is no exception. IE supports the `inherit` value only from version 8, except for the **`direction`** and **`visibility`** properties. Great.

4. Using Your Tools

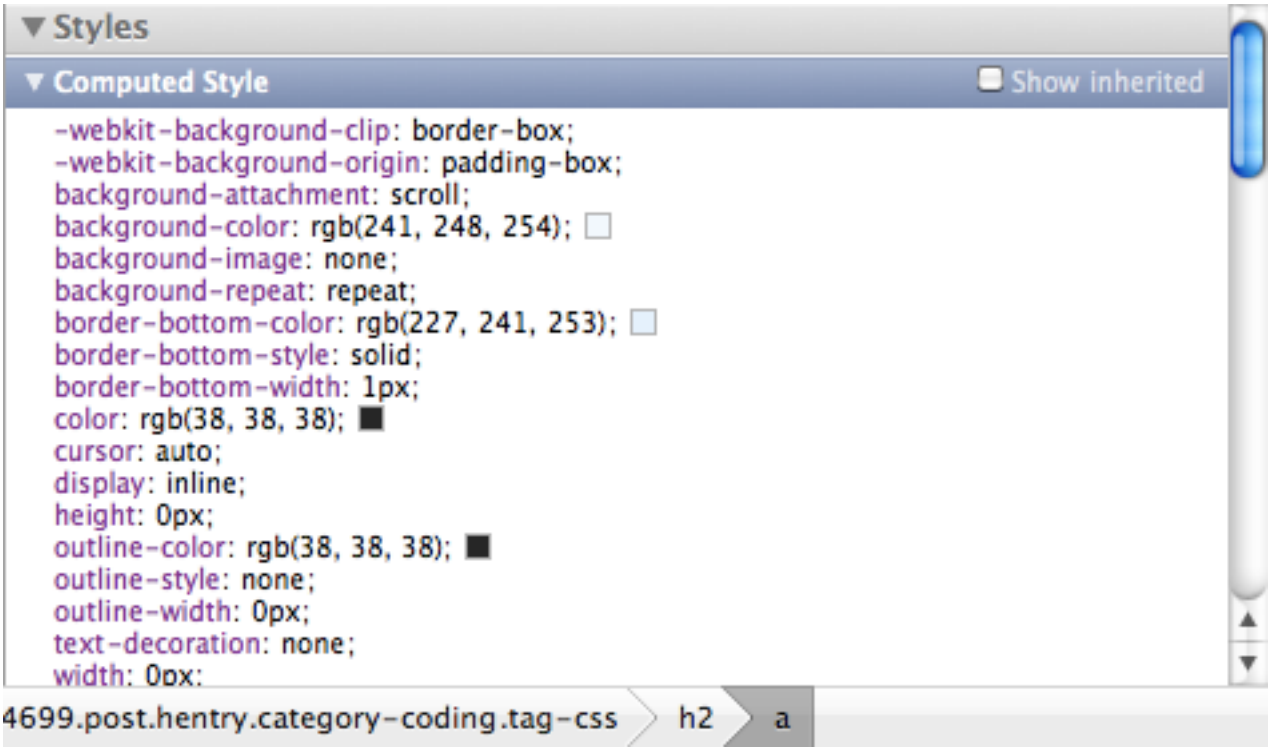
If you use tools like Firebug or Safari's Web Inspector, you can see how a given cascade works, which selectors have higher specificity and how inheritance is working on a particular element.

For example, here below is *Firebug* in action, inspecting an element on the page. You can see that some properties are overridden (i.e. crossed out) by other more specific rules:



Firebug in action, informing you how specificity is working.

In the next shot, *Safari's Web Inspector* shows the computed values of an element. This way, you can see the values even though they haven't been explicitly added to the style sheet:



With Safari's Web Inspector (and Firebug), you can view the computed values of a particular element.

5. Conclusion

Hopefully this article has opened your eyes to (or has refreshed your knowledge of) CSS inheritance and specificity. We encourage you to read the articles cited below, as well as Smashing Magazine's previous article on the topic.

Even if you don't think about them, these issues are present in your daily work as a CSS author. Especially in the case of specificity, it's important to know how they affect your style sheets and how to plan for them so that they cause only minimal (or no) problems.

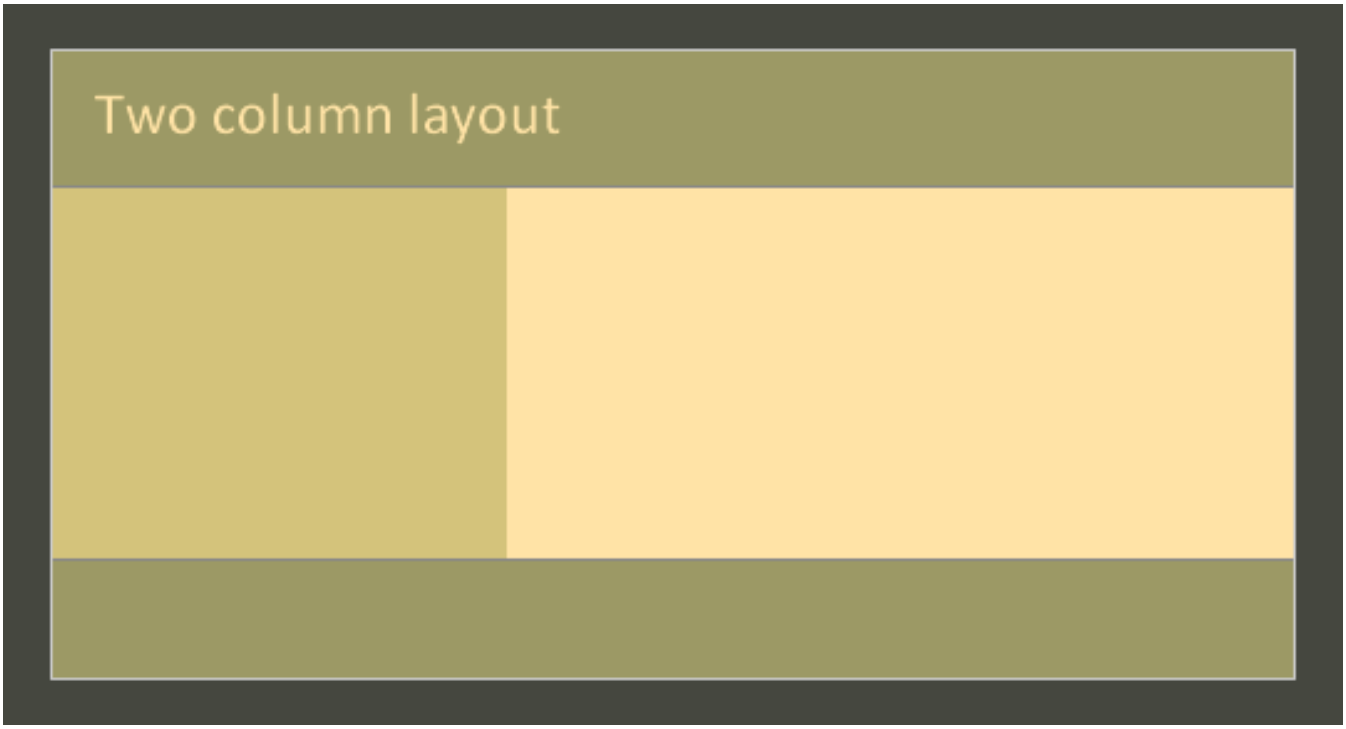
Equal Height Column Layouts With Borders And Negative Margins In CSS

Thierry Koblentz

"What? Another "Equal Height Columns" article? Enough already!" If this is what you think, then think again because this solution is *different*. It does not rely on any of the usual tricks. It does not use images, nor extra markup, nor CSS3, nor pseudo-classes, nor Javascript, nor absolute positioning. All it uses is **`border`** and **`negative margin`**. Please note that this article will also demonstrate different construct techniques and will brush up on a few concepts.

1. Centering columns without a wrapper `div`

This layout will be built without a wrapper **`div`**:



THE BASICS

We use the background of **body** and the border of one of the columns to create background colors that vertically fill the “row”.

THE MARKUP

```
<div id="header">
  <h2><a href="#">Header</a></h2>
  <p>Lorem ipsum...</p>
</div>
<div id="sidebar">
  <h2><a href="#">Sidebar</a></h2>
  <p>Lorem ipsum...</p>
</div>
<div id="main">
  <h2><a href="#">Main</a></h2>
  <p>Lorem ipsum...</p>
</div>
```

```
<div id="footer">
  <h2><a href="#">Footer</a></h2>
  <p>Lorem ipsum...</p>
</div>
```

Tip: always include links within your dummy text to spot stacking context issues.

About using body as a wrapper:

- always style **`html`** with a background to prevent the background of body from extending beyond its boundaries and be painted across the viewport.
- never style **`html`** with **`height: 100%`** or the background of body will be painted no taller than the viewport.

THE CSS

```
html {
  background: #9c9965;
}
body {
  width: 80%;
  margin: 20px auto;
  background: #ffe3a6;
}
#header {
  background: #9c9965;
}
#sidebar {
  float: left;
  width: 200px;
  background: #d4c37b;
}
#main {
```

```
  border-left: 200px solid #d4c37b;
}
#footer {
  clear: left;
  background: #9c9965;
}
```

What do these rules do exactly?

- We style **`html`** with a background to prevent the browser from painting the background color of **`body`** *outside* our layout.
- We style **`body`** with auto margin to center the layout; the width is set using percentage. The background declaration is for **`#main`.**
- We style the background of **`#header`** to mask the background color of body (the same goes for **`#footer`**).
- The background color of **`#sidebar`** matches the border color of **`#main`.** This is the trick we use to make our columns appear as being of equal height.
- The footer clears any previous float so it stays below the columns, at the bottom of the layout.

If you take a look at this first step, you'll notice that the heading in **`#sidebar`** is not vertically aligned with the heading in **`#main`** and that we have some gap at the top and bottom of the **`#sidebar`.** This is because out of these two containers, only one is a block formatting context. So margins do not collapse in **`#sidebar`** while they do in **`#main`.** This also means that **`#main`** will not contain floats and that applying **`clear:left`** to any nested element in there will clear **`#sidebar`** as well.

So to prevent any float and margin collapsing issues we make all the main boxes block formatting contexts.

```
#header,
#footer {
  overflow: hidden;
  zoom: 1;
}
#main {
  float: left;
}
#sidebar {
  margin-right: -200px;
}
```

Note: if things look a bit different in IE 6 and 7 it is because in these browsers *default* margins do collapse inside block-formatting contexts. The following should also be considered:

- **`overflow`** and **`zoom`** on **`#header`** and **`#footer`** make these elements new block formatting contexts.
- For **`#main`** we use **`float`** rather than **`overflow`** to prevent potential issues if we had to offset some nested content.
- The negative margin keeps **`#main`** into place because now that it is a float, its border box comes *next* to the margin box of **`#sidebar`** (the negative value must match the dimension of the said box).

If you look at the second step, you'll see that the border of **`#main`** hides **`#sidebar.`** This is because of the stacking context. In the flow (tree order), **`#main`** comes after **`#sidebar`** so the former overlaps the latter.

Positioning **`#sidebar`** brings it up in the stack.

```
#sidebar {
```

```
  position: relative;
}
```

Note: if you make **`#main`** a new containing block you'll revert to the original stack order. In this case, you'll need to use **`z-index`** to keep **`#sidebar`** on top of **`#main.`**

If you look at step three, you'll see that we are almost there. The last things to do are mostly cosmetic. I've inserted a base styles sheet:

```
<link rel="stylesheet" type="text/css" href="http://
tjkdesign.com/ez-css/css/base.css">
```

and then added these rules:

```
html {
  height: auto;
}

body {
  border: 1px solid #efefef;
}

#header,
#main,
#sidebar,
#footer {
  padding-bottom: 2em;
}
```

Why do we need these rules?

- We can reset the height on **`html`** so the background of **`#main`** is not cut-off at the fold (this styling is inherited from the base styles sheet).
- We can draw a border all around the layout.

- Because the base styles sheet only sets top margins, we can create gaps at the bottom of the main boxes via padding.

Note: The rule for **`html`** is shown here, but it makes more sense to *remove* that rule from the base styles sheet rather than overwriting the declaration here.

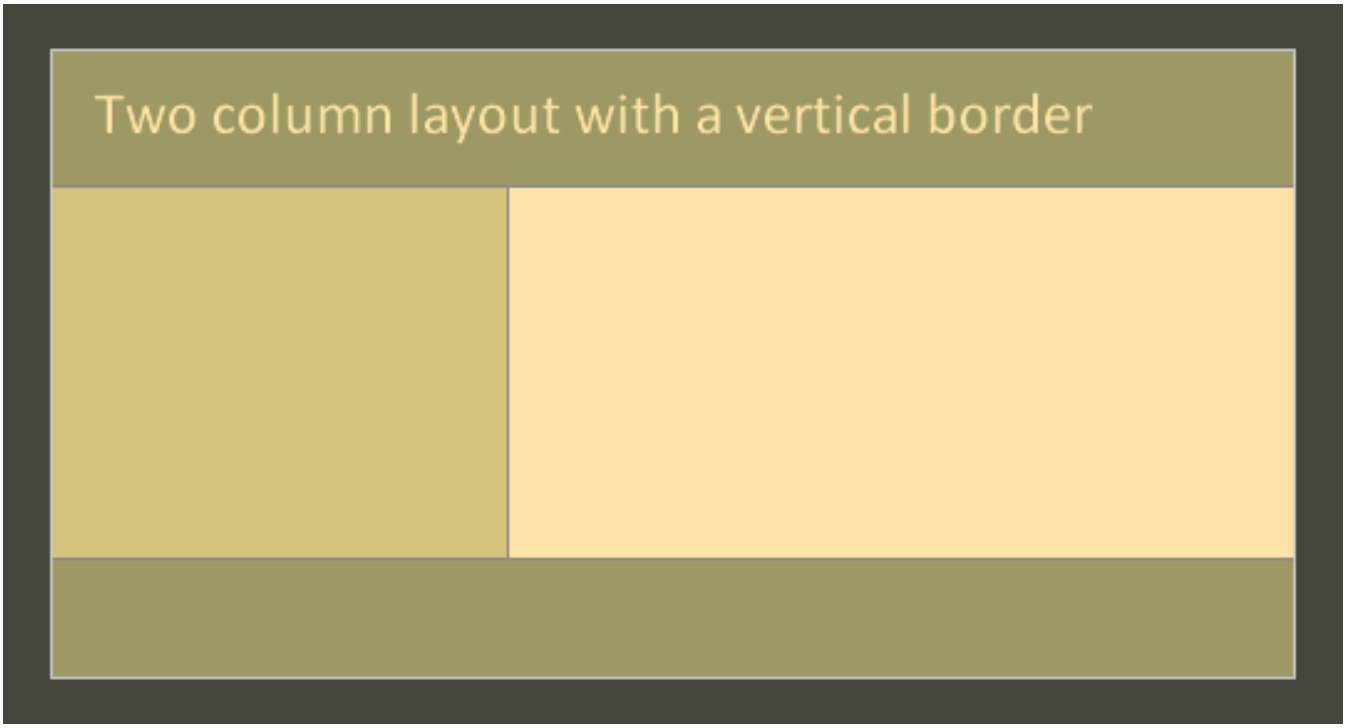
This is the last step for this first layout. It relies on these simple rules:

```
html {
  height: auto;
  background: #45473f;
}
body {
  width: 80%;
  margin: 20px auto;
  background: #ffe3a6;
  border: 1px solid #efefef;
}
#sidebar {
  float: left;
  position: relative;
  width: 200px;
  margin-right: -200px;
  background: #d4c37b; /* color must match #main's left border
*/
}
#main {
  float: left;
  border-left: 200px solid #d4c37b; /* color must match
#sidebar's background */
}
#header,
#footer {
  clear: left;
  overflow: hidden;
```

```
  zoom: 1;
  background: #9c9965;
}
#header,
#main,
#sidebar,
#footer {
  padding-bottom:2em;
}
```

2. Creating a 2-col-layout with two borders in between the columns

We'll build this one with a single wrapper for our two columns, and we want to paint a vertical border between the two columns.



THE BASICS

The wrapper **`div`** allows us to be a bit more creative here. The background of the wrapper is used to paint the background of one column, while its left border is used to paint the background color of the other column. The vertical border will be done by overlapping the right border of the left column with the left border of the right column.

Note: if you have use a fixed width layout (vs. fluid like here), then the wrapper can be used to create the two background colors as well as the vertical border at the same time. This is done by using the left border for the left column, the right border for the right column and the background for the vertical border. Yes, this means the content box is one pixel wide and that negative margins are used to pull the columns into place.

MARKUP

```
<div id="header">
  <h2><a href="#">Header</a></h2>
  <p>Lorem ipsum...</p>
</div>
<div id="wrapper">
  <div id="sidebar">
      <h2><a href="#">Sidebar</a></h2>
      <p>Lorem ipsum...</p>
  </div>
  <div id="main">
      <h2><a href="#">Main</a></h2>
      <p>Lorem ipsum...</p>
  </div>
</div>
<div id="footer">
  <h2><a href="#">Footer</a></h2>
  <p>Lorem ipsum...</p>
</div>
```

CSS

We start with the generic rules from the previous demo:

```
html {
  background: #45473f;
}
body {
  width: 80%;
  margin: 20px auto;
  background: #ffe3a6;
}
#header,
#footer {
  overflow: hidden;
  zoom: 1;
  background: #9c9965;
}
#sidebar {
  float: left;
  width: 200px;
}
#main {
  float: left;
}
```

To which we add **`position: relative`**:

```
#wrapper {
  display: inline-block;
  border-left: 200px solid #d4c37b;
  position: relative;
}
#sidebar {
  margin-left: -200px;
  position: relative;
}
```

Note: there is no need to use **`clear`** on the footer since **`#wrapper`** *contains* both floats.

- Rather than using **`overflow/zoom`**, we use **`inline-block`** to create a new block formatting context (this declaration also triggers hasLayout). The left border will paint a background color behind **`#sidebar`**.
- Negative margin is used to bring **`#sidebar`** outside the content box of the parent's container (to make it overlap the border of **`#wrapper`**).

The case of IE6: If the above rules use **`position: relative`** (twice), it is because of IE 6. It is applied on **`#wrapper`** to prevent **`#sidebar`** from being clipped outside of its content box. It is also applied on **`#sidebar`** to make sure that the elements are "always" painted with the proper offset.

If you look at this first step, you'll see that we have everything working, but the vertical border is in between the columns. You should also notice that in browsers other than IE 6 and 7, there is a small gap at the bottom of **`#sidebar`** (at the bottom **`#wrapper`** actually). This is because **`#wrapper`** is styled with **`inline-block`** so it is sitting on the baseline of the line box. The gap you see is the "descender space" (the space reserved for descenders in lowercase letters).

So these are the rules to remove the gap and create the vertical border:

```
#wrapper {
  vertical-align: bottom;
}
#sidebar {
  margin-right: -1px;
  border-right: 1px solid #888;
}
#main {
  border-left:1px solid #888;
}
```

What do these rules do?

`vertical-align: bottom` makes **`#wrapper`** sit at the bottom of the line box rather than the baseline.

the two borders (for **`#sidebar`** and **`#main`**) overlap because of the negative *right* margin set on **`#sidebar`**. This overlap guarantees that this "common" border will be as tall as the tallest column.

If you look at step two, things look much better. The last things to do is to add the base styles sheet and the same rules we used at the end of the first demo:

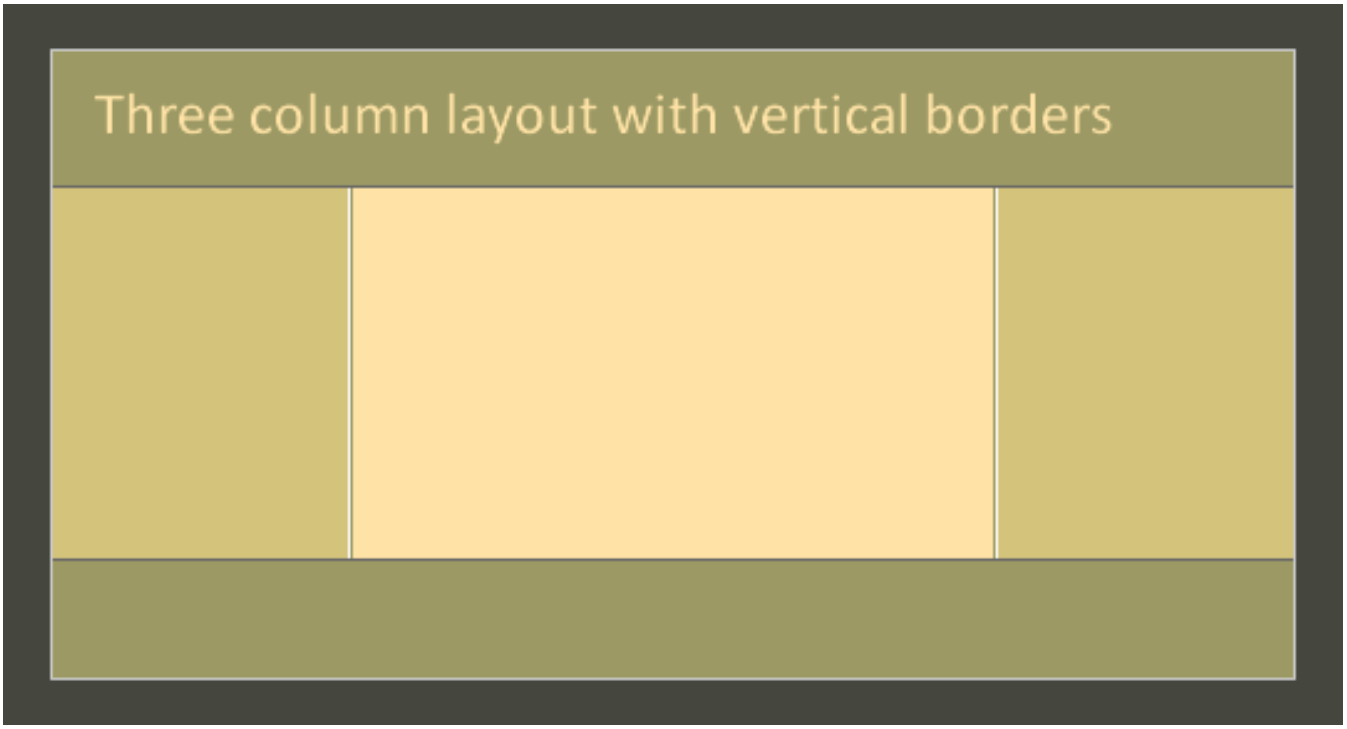
```
<link rel="stylesheet" type="text/css" href="http://
tjkdesign.com/ez-css/css/base.css">
```

and then add these rules:

```
html {
  height: auto;
}
body {
  border: 1px solid #efefef;
}
#header,
#main,
#sidebar,
#footer {
  padding-bottom: 2em;
}
```

3. Creating a three column layout with a border in between the columns

We'll build a layout with a single **`#main`-**wrapper, one containing **all the divs**. This approach complicates things a bit, but it also allows to tackle new challenges. Please note that with this layout, the vertical borders will not show in IE 6 and 7.



THE BASICS

We use the wrapper to create the background of the three columns. The left and right borders of the wrapper will be used for the two side bars while its background will be used for the main content.

THE MARKUP

```
<div id="wrapper">
  <div id="header"><h2><a href="#">Header</a></h2><p>Lorem
ipsum...</p></div>
  <div id="sidebar"><h2><a href="#">Sidebar</a></h2><p>Lorem
ipsum...</p></div>
  <div id="aside"><h2><a href="#">Aside</a></h2><p>Lorem
ipsum...</p></div>
  <div id="main"><h2><a href="#">Main</a></h2><p>Lorem
ipsum...</p></div>
  <div id="footer"><h2><a href="#">Footer </a></h2><p>Lorem
ipsum...</p></div>
</div>
```

CSS

We start with the generic rules from the previous demos:

```
html {
  background: #45473f;
}

body {
  width: 80%;
  margin: 20px auto;
  background: #ffe3a6;
}
#header,
#footer {
  overflow: hidden;
  zoom: 1;
  background: #9c9965;
}
#sidebar {
  float: left;
  width: 200px;
```

```
}
#main {
  float: left;
}
To which we add:
#wrapper {
  border-left: 200px solid #D4C37B;
  background-color: #ffe3a6;
  border-right: 200px solid #D4C37B;
}
```

This code sets the background color for the three columns. In the same sequence as the above declarations.

If you look at this first step, you'll see that we have achieved the background effect we are looking for, but things look pretty broken. Everything shows inside the wrapper's content box.

These next rules should fix the display of the three columns (**`zoom: 1`** for the **`#wrapper`** and **`position: relative`** for **`#sidebar`** and **`#aside`**):

`#aside` is given a width and floated to the right. The negative margins pull each side bar over the wrapper's border — outside of the content box.

Note:IE 6 and 7 needs **`#wrapper`** to have a layout, hence the use of **`zoom`**. IE 6 needs the two **`position`** declarations for the same reason as in the previous demos.

If you look at step two, you'll see that **`#header`** does not stretch across the entire layout and that **`#footer`** is nowhere to be found.

These two rules should take care of everything:

```
#header,
#footer {
  margin-left: -200px;
```

```
  margin-right: -200px;
  position: relative;
}
#footer {
  clear: both;
}
```

The negative margin on both sides of **#header** and **#footer** stretches the two boxes outside of the wrapper's content box. **clear:both** makes the footer clears all the columns. This is step three.

Once again, the **position** declaration is for IE 6. Just remember to always position elements that you offset.

WHAT'S NEXT?

You know the drill. We insert a base styles sheet in the document:

```
<link rel="stylesheet" type="text/css" href="http://
tjkdesign.com/ez-css/css/base.css">
```

and add the usual:

```
html {
  height: auto;
}
body {
  border: 1px solid #efefef;
}
#header,
#main,
#sidebar,
#footer {
  padding-bottom: 2em;
}
```

Step four shows how things look before we tackle the vertical borders.

ADDING VERTICAL BORDERS

The following technique is inspired from the companion columns technique (Ingo Chao) and the Nicolas Gallagher method.

To get the effect we want (two borders touching each other), we use generated content to which we apply a background color and a border.

THE CSS

```
html:before {
  content: ".";
  position: absolute;
  height: 20px;
  background: #45473f;
  left: 0;
  right: 0;
  z-index: 2;
}
body {
  border-top: 0;
}
#header {
  border-top: 1px solid #fff;
}
#wrapper {
  position: relative;
}
#header,
#footer {
  z-index: 1;
}
#wrapper:before,
```

```
#wrapper:after {
  content: ".";
  position: absolute;
  width: 1px;
  height: 2000px;
  background: #9c9965;
  bottom: 0;
}
#wrapper:before {
  left: 0;
  border-left: 1px solid #fff;
}
#wrapper:after {
  right: 0;
  border-right: 1px solid #fff;
}
body {
  position: relative\9;
  z-index: -1;
}
```

OK, so what's going on here?

- The fake borders get out of the container (at the top), so the first rule paints generated content on top of them. Note that we would not need this rule if the color of the fake borders was the same as the page's background (**`html`**), or if there was no gap between the top of the viewport and the layout.
- Because these borders are painted over the border around **`body`**, we move the top border from **`body`** to **`#header`**.
- To properly position the fake borders, we need to make the wrapper the containing block for the generated content.

- We bring **`#header`** and **`#footer`** above the stack so they hide the fake borders which are painted inside the wrapper (from bottom to top).
- This is the generated content we use to create the columns.

The case of IE 8: The last rule is for IE 8. Without this, IE 8 would not paint the generated content over the borders that escape the wrapper (at the top). If this declaration is sandboxed via the "\9" hack, it is because Gecko browsers would make everything unclickable/unselectable.

Note: these pseudo-classes are not supported by IE 6 and 7, so in these browsers, there are no borders between the columns.

Things to consider

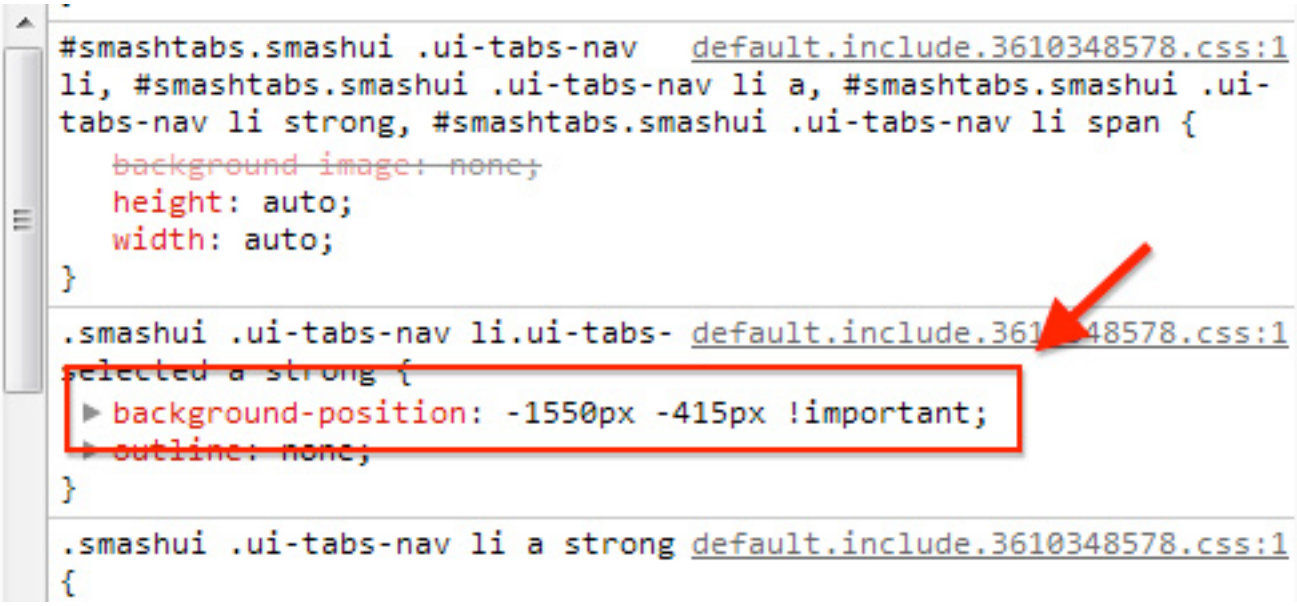
The third layout uses one *main* wrapper, but it would make more sense to use a *inner* wrapper instead to hold only the columns. In case this route was taken here, then it was only for those of you who are stuck with this type of construct, but want to implement this solution for equal height columns.

When absolutely positioning elements inside a containing block with wide columns like in the last two demos, remember that the reference is the padding box, so **`0`** for **`right`** or **`left`** may not be the value you would want to use.

!important CSS Declarations: How And When To Use Them

Louis Lazaris

When the CSS1 specification was drafted in the mid to late 90s, it introduced **`!important`** declarations that would help developers and users easily override normal specificity when making changes to their stylesheets. For the most part, **`!important`** declarations have remained the same, with only one change in CSS2.1 and nothing new added or altered in the CSS3 spec in connection with this unique declaration.



Let's take a look at what exactly these kinds of declarations are all about, and when, if ever, you should use them.

A Brief Primer on the Cascade

Before we get into `!important` declarations and exactly how they work, let's give this discussion a bit of context. In the past, Smashing Magazine has covered CSS specificity in-depth, so please take a look at that article if you want a detailed discussion on the CSS cascade and how specificity ties in.

Below is a basic outline of how any given CSS-styled document will decide how much weight to give to different styles it encounters. This is a general summary of the cascade as discussed in the spec:

- Find all declarations that apply to the element and property
- Apply the styling to the element based on importance and origin using the following order, with the first item in the list having the least weight:
    - Declarations from the user agent
    - Declarations from the user
    - Declarations from the author
    - Declarations from the author with `!important` added
    - Declarations from the user with `!important` added
- Apply styling based on specificity, with the more specific selector "winning" over more general ones
- Apply styling based on the order in which they appear in the stylesheet (i.e., in the event of a tie, last one "wins")

With that basic outline, you can probably already see how **`!important`** declarations weigh in, and what role they play in the cascade. Let's look at **`!important`** in more detail.

Syntax and Description

An **`!important`** declaration provides a way for a stylesheet author to give a CSS value more weight than it naturally has. It should be noted here that the phrase "!important declaration" is a reference to an entire CSS declaration, including property and value, with **`!important`** added (thanks to Brad Czerniak for pointing out this discrepancy). Here is a simple code example that clearly illustrates how **`!important`** affects the natural way that styles are applied:

```
#example {
  font-size: 14px !important;
}
#container #example {
  font-size: 10px;
}
```

In the above code sample, the element with the id of "example" will have text sized at 14px, due to the addition of **`!important`**.

Without the use of **`!important`**, there are two reasons why the second declaration block should naturally have more weight than the first: The second block is later in the stylesheet (i.e. it's listed second). Also, the second block has more specificity (**`#container`** followed by **`#example`** instead of just **`#example`**). But with the inclusion of **`!important`**, the first **`font-size`** rule now has more weight.

Some things to note about **!important** declarations:

When **!important** was first introduced in CSS1, an author rule with an **!important** declaration held more weight than a user rule with an **!important** declaration; to improve accessibility, this was reversed in CSS2

- If **!important** is used on a shorthand property, this adds "importance" to all the sub-properties that the shorthand property represents
- The **!important** keyword (or statement) must be placed at the end of the line, immediately before the semicolon, otherwise it will have no effect (although a space before the semicolon won't break it)
- If for some particular reason you have to write the same property twice in the same declaration block, then add **!important** to the end of the first one, the first one will have more weight in every browser except IE6 (this works as an IE6-only hack, but doesn't invalidate your CSS)
- In IE6 and IE7, if you use a different word in place of **!important** (like **!hotdog**), the CSS rule will still be given extra weight, while other browsers will ignore it

When Should !important Be Used?

As with any technique, there are pros and cons depending on the circumstances. So when should it be used, if ever? Here's my subjective overview of potential valid uses.

NEVER

!important declarations should not be used unless they are absolutely necessary after all other avenues have been exhausted. If you use **!important** out of laziness, to avoid proper debugging, or to rush a project

to completion, then you're abusing it, and you (or those that inherit your projects) will suffer the consequences.

If you include it even sparingly in your stylesheets, you will soon find that certain parts of your stylesheet will be harder to maintain. As discussed above, CSS property importance happens naturally through the cascade and specificity. When you use **`!important`**, you're disrupting the natural flow of your rules, giving more weight to rules that are undeserving of such weight.

If you never use **`!important`**, then that's a sign that you understand CSS and give proper forethought to your code before writing it.

That being said, the old adage "never say never" would certainly apply here. So below are some legitimate uses for **`!important.`**

TO AID OR TEST ACCESSIBILITY

As mentioned, user stylesheets can include **`!important`** declarations, allowing users with special needs to give weight to specific CSS rules that will aid their ability to read and access content.

A special needs user can add **`!important`** to typographic properties like **`font-size`** to make text larger, or to color-related rules in order to increase the contrast of web pages.

In the screen grab below, Smashing Magazine's home page is shown with a user-defined stylesheet overriding the normal text size, which can be done using Firefox's Developer Toolbar:

In this case, the text size was adjustable without using **`!important`**, because a user-defined stylesheet will override an author stylesheet regardless of specificity. If, however, the text size for body copy was set in the author stylesheet using an **`!important`** declaration, the user stylesheet could not override the text-size setting, even with a more specific selector. The inclusion of **`!important`** resolves this problem and keeps the adjustability of text size within the user's power, even if the author has abused **`!important`**.

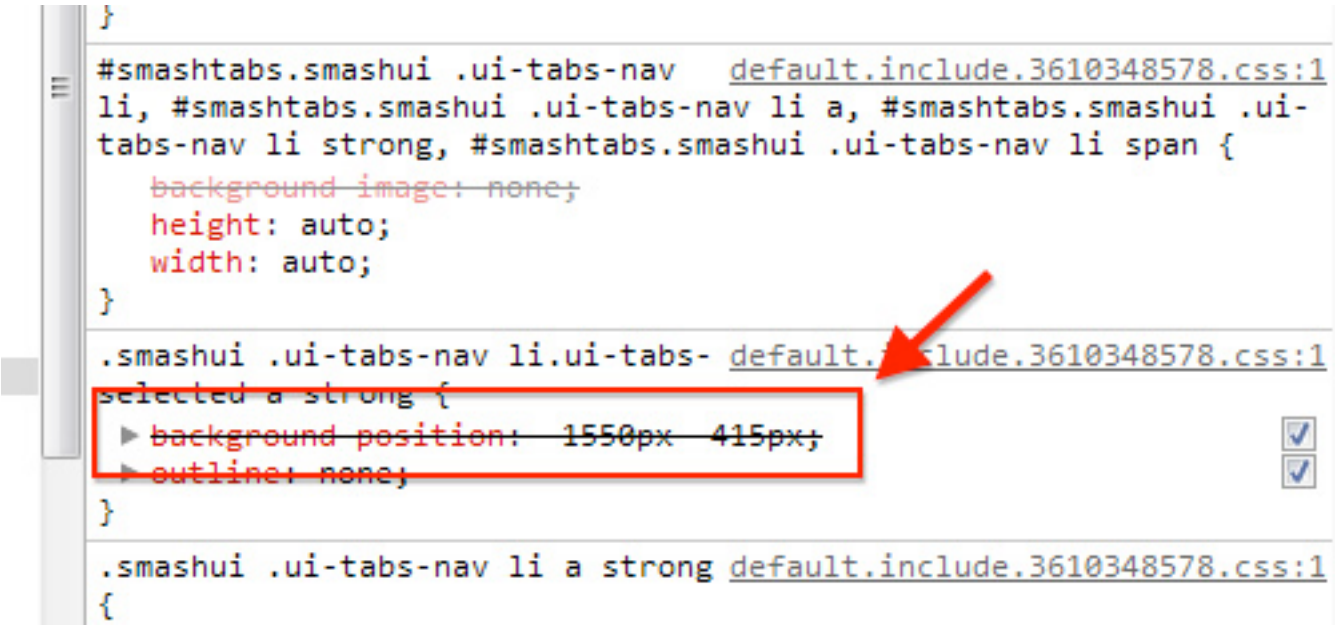
TO TEMPORARILY FIX AN URGENT PROBLEM

There will be times when something bugs out in your CSS on a live client site, and you need to apply a fix very quickly. In most cases, you should be able to use Firebug or another developer tool to track down the CSS code that needs to be fixed. But if the problem is occurring on IE6 or another browser that doesn't have access to debugging tools, you may need to do a quick fix using **`!important.`**

After you move the temporary fix to production (thus making the client happy), you can work on fixing the issue locally using a more maintainable method that doesn't muck up the cascade. When you've figured out a better solution, you can add it to the project and remove **`!important`** — and the client will be none the wiser.

TO OVERRIDE STYLES WITHIN FIREBUG OR ANOTHER DEVELOPER TOOL

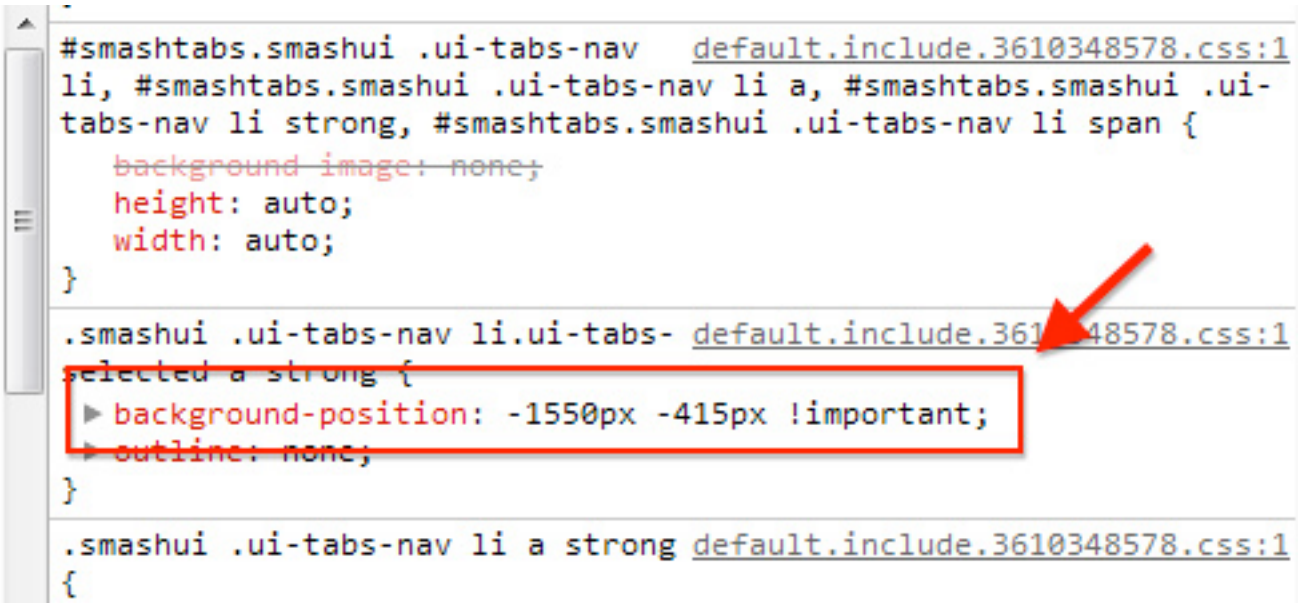
Inspecting an element in Firebug or Chrome's developer tools allows you to edit styles on the fly, to test things out, debug, and so on — without affecting the real stylesheet. Take a look at the screen grab below, showing some of Smashing Magazine's styles in Chrome's developer tools:



The highlighted background style rule has a line through it, indicating that this rule has been overridden by a later rule. In order to reapply this rule, you could find the later rule and disable it. You could alternatively edit the selector to make it more specific, but this would give the entire declaration block more specificity, which might not be desired.

`!important` could be added to a single line to give weight back to the overridden rule, thus allowing you to test or debug a CSS issue without making major changes to your actual stylesheet until you resolve the issue.

Here's the same style rule with **`!important`** added. You'll notice the line-through is now gone, because this rule now has more weight than the rule that was previously overriding it:



TO OVERRIDE INLINE STYLES IN USER-GENERATED CONTENT

One frustrating aspect of CSS development is when user-generated content includes inline styles, as would occur with some WYSIWYG editors in CMSs. In the CSS cascade, inline styles will override regular styles, so any undesirable element styling that occurs through generated content will be difficult, if not impossible, to change using customary CSS rules. You can circumvent this problem using an **`!important`** declaration, because a CSS rule with **`!important`** in an author stylesheet will override inline CSS.

FOR PRINT STYLESHEETS

Although this wouldn't be necessary in all cases, and might be discouraged in some cases for the same reasons mentioned earlier, you could add **`!important`** declarations to your print-only stylesheets to help override specific styles without having to repeat selector specificity.

FOR UNIQUELY-DESIGNED BLOG POSTS

If you've dabbled in uniquely-designed blog posts (many designers take issue with using "art direction" for this technique, and rightly so), as showcased on Heart Directed, you'll know that such an undertaking requires each separately-designed article to have its own stylesheet, or else you need to use inline styles. You can give an individual page its own styles using the code presented in this article on the Digging Into WordPress blog.

The use of **`!important`** could come in handy in such an instance, allowing you to easily override the default styles in order to create a unique experience for a single blog post or page on your site, without having to worry about natural CSS specificity.

Conclusion

`!important` declarations are best reserved for special needs and users who want to make web content more accessible by easily overriding default user agent or author stylesheets. So you should do your best to give your CSS proper forethought and avoid using **`!important`** wherever possible. Even in many of the uses described above, the inclusion of **`!important`** is not always necessary.

Nonetheless, **`!important`** is valid CSS. You might inherit a project wherein the previous developers used it, or you might have to patch something up quickly — so it could come in handy. It's certainly beneficial to understand it better and be prepared to use it should the need arise.

Do you ever use **`!important`** in your stylesheets? When do you do so? Are there any other circumstances you can think of that would require its use?

CSS Sprites Revisited

Niels Matthijs

I'm pretty confident that I won't surprise anyone here by saying that CSS sprites have been around for quite a while now, rearing their somewhat controversial heads in the Web development sphere as early as 2003.

Still, the CSS sprite hasn't truly found its way into the everyday toolkit of the common Web developer. While the theory behind CSS sprites is easy enough and its advantages are clear, they still prove to be too bothersome to implement, especially when time is short and deadlines are looming. Barriers exist to be breached, though, and we're not going to let a couple of tiny bumps in the road spoil the greater perks of the CSS sprite.

If you want more background information on best practices and practical use cases, definitely read "The Mystery of CSS Sprites: Techniques, Tools and Resources." If you're the defensive type, I would recommend "CSS Sprites: Useful Technique, or Potential Nuisance?," which discusses possible caveats.

I won't take a stance on the validity of CSS sprites. The aim of this article is to find out why people still find it difficult to use CSS sprites. Also, we'll come up with a couple of substantial improvements to current techniques. So, start up Photoshop (or your CSS sprite tool of choice), put on your LESS and Sass hats, and brush up your CSS pseudo-element skills, because we'll be mixing and matching our way to easier CSS sprite implementation.

The Problem With CSS Sprites

While Photoshop is booting, take a moment to consider why CSS sprites have had such a hard time getting widespread acceptance and support. I'm always struggling to find the correct starting position for each image within a sprite. I'll usually forget the exact coordinates by the time I need to write them down in the style sheet, especially when the image is located at **`x: 259, y: 182`** and measures 17×13 pixels.

I'm always switching back and forth between Photoshop and my CSS editor to write down the correct values, getting more and more frustrated with every switch. I know software is out there specifically tailored to help us deal with this, but I've never found an application that lets me properly structure its CSS output so that I can just copy and paste it into my CSS file.

Another important thing to realize is that CSS sprites are meant to be one part of our website optimization toolkit. Many of the people who write and blog about CSS sprites are comfortably wearing their optimization hats while laying out best practices, often going a little overboard and losing track of how much effort it requires to implement their methods of choice.

This is fine if you're working on an optimization project, but it becomes counterproductive when you need to build a website from scratch while fighting tight deadlines. If you have time to truly focus on implementing a CSS sprite, it's really not all that hard; but if you have to juggle 50 other priorities at the same time, it does turn into a nuisance for many developers. With these two factors in mind, we'll try to find a way to make image targeting easier, while coming to terms with the fact that sometimes we have to sacrifice full optimization in the name of ease of development.

CSS sprites in the wild: Amazon, Google and Facebook.

Preparing The Sprite

If you look online for examples of CSS sprites, you'll see that most are optimized for an ideal use of real estate—gaps between images are kept to a minimum in order to keep the load of the entire sprite as low as possible. This reduces loading time considerably when the image is first downloaded, but it also introduces those horrible image coordinates that we mentioned earlier. So, why not take a slightly different approach? Let's look for an easier way to target images while making sure the resulting sprite is not substantially bigger than the sum of its individual images.

Rather than try to stack the images in such a way that makes the resulting sprite as small as possible, let's try a different layout. Instead of a random grid, we're going to build a nice square grid, reserving the space of each square for one image (although larger images might cover multiple squares). The goal is to be able to target the starting point of every image (top-left corner) with a very simple mathematical function.

The size of the grid squares will depend on the average dimension of the images that you'll be spriting. For my website, I used a 32×32 pixel grid (because only one image exceeds this dimension), but grid sizes will vary according to the situation. Another thing to take into account is the image bleeding that can occur when zooming a page; if you want to play it safe, add a little padding to each image within the sprite. For extra optimization, you could also use a rectangular grid instead of a square grid, further reducing any wasted space; while a tad more complex, this isn't much different from what we'll be doing. For this article, though, we'll stick with the square grid.

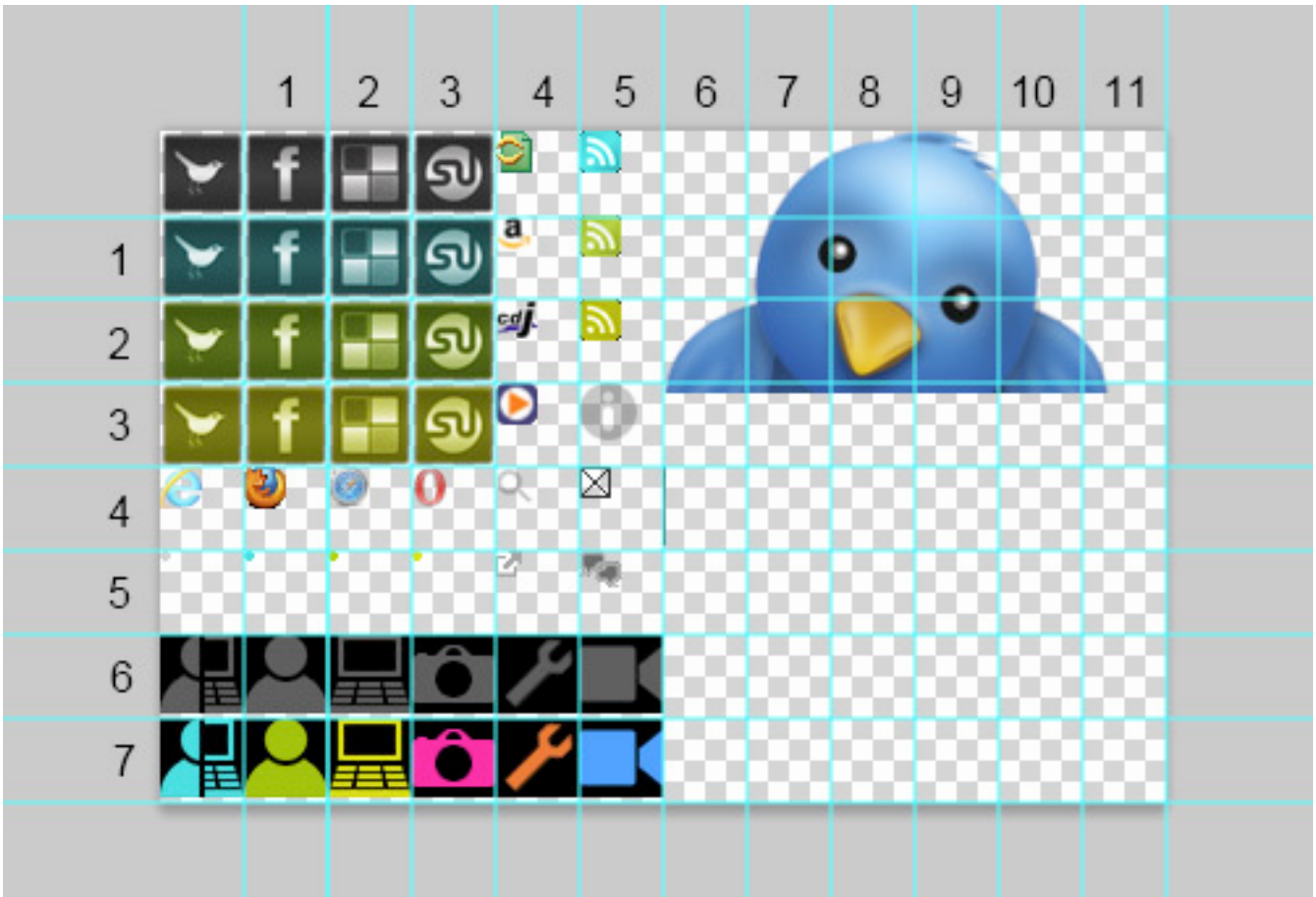
Preparing The Sprite: A Photoshop Bonus

I'm no designer, so I'm not sure how applicable this section is beyond the realm of Photoshop, but there are still some noteworthy improvements that can be made before we get to the actual code. First of all, visualizing the grid can be of great help. Photoshop has guides to do this (so you don't actually have to include the markers in your sprite), with the added bonus that layers will snap to these guides (making pixel-perfect positioning of each individual image much easier).

Manually adding these guides can be somewhat of a drag, though. Luckily, a colleague of mine (hats off to Filip Van Tendeloo) was kind enough to write and share a little Photoshop script that builds this grid of guides

automatically based on the base size of your square. Pretty convenient, no? Just download the script, save it in the **`Presets\Scripts`** directory, and choose **`File → Scripts → Sprite Grid`** from the Photoshop menu.

Finally, to really finish things off, you can add numbers to the x and y axes of your grid so that you can easily pinpoint every square in the grid. Don't add these numbers to the actual sprite, though; just copy the sprite into a new image and add them there. These numbers are just for reference—but trust me, they will definitely come in handy in a minute.



Example of a reference image.

If you can't (or don't want to) use preprocessing languages such as LESS and Sass, just add the coordinates of each square instead.

LESS And Sass To The Rescue

With all of the images on the same square grid, we can now easily compute the top-left coordinate of each image (this coordinate is the exact match needed for the **`background-position`** values to reposition the sprite). Just take the base size of the grid and multiply it by the numbers that you just added to your reference image. Say you need to target the image at a position of (5,3), and your grid size is 32 pixels; the top-left position of your image will be 32 × (5,3) = (160,96). To add these values to your CSS file, just use their negative equivalents, like so:

```
background-position: -160px -96px;
```

Leave your calculator where it is for the time being, because it would be too much of a hassle. If computers are good at anything, it's doing calculations, so we're going to make good use of that power. Plain CSS doesn't allow us to do calculations (yet), but languages like LESS and Sass were made to do just that. If you need more background information, check out the article "An Introduction to LESS and Comparison to Sass." While both languages offer similar functionality, their syntaxes are slightly different. I'm a LESS man myself, but if you're used to Sass, converting the following examples to Sass code shouldn't be too hard. Now for some advanced CSS magic:

```
@spriteGrid: 32px;
.sprite(@x, @y) {
   background: url(img/sprite.png) no-repeat;
   background-position: -(@x*@spriteGrid) -(@y*@spriteGrid);
}
```

Nothing too exciting so far. First, we've defined the grid size using a LESS variable **`(@spriteGrid)`**. Then, we made a mixin that accepts the numbers we added to the reference image earlier. This mixin uses the grid variable and the image's coordinates to calculate the base position of the image that

we want to target. It might not look fancy, but it sure makes the basic use of sprites a lot easier already. From now on, you can just use **`.sprite(1,5)`**, and that's all there is to it. No more annoying calculations or finding random start coordinates.

Of course, the mixin above only works in our somewhat simplified example (using a square grid and just one sprite). Some websites out there are more complex and might require different sprites using rectangular grids for additional optimization. This isn't exactly impossible to fix with LESS, though:

```
.spriteHelper(@image, @x, @y, @spriteX, @spriteY) {
   background: url("img/@{image}.png") no-repeat;
   background-position: -(@x*@spriteX) -(@y*@spriteY);
}
.sprite(@image, @x, @y) when (@image = sprite1), (@image =
sprite3){
   @spriteX: 32px;
   @spriteY: 16px;
   .spriteHelper(@image, @x, @y, @spriteX, @spriteY);
}
.sprite(@image, @x, @y) when (@image = sprite2){
   @spriteX: 64px;
   @spriteY: 32px;
   .spriteHelper(@image, @x, @y, @spriteX, @spriteY);
}
```

Yes, this looks a bit more daunting, but it's still pretty basic if you take a minute to understand what's going on. Most importantly, the added complexity is safely hidden away inside the LESS mixins, and we're still able to use the same sprite mixin as before, only now with three parameters in total. We've just added an image parameter so that we can easily determine which sprite to use each time we call our mixin.

Different sprites might have different grid dimensions, though; so, for each grid dimension, we need to write a separate LESS mixin. We match each

sprite to its specific dimension (the example above handles three different sprites and matches it to two different grid dimensions), and we define the dimensions (`@spriteX` and `@spriteY`) locally in each mixin. Once that is done, we offload the rest of the work to a little `.spriteHelper` mixin that does all of the necessary calculations and gives us the same result as before.

The “guards” (as LESS calls them—characterized by the keyword “when”) are quite new to the LESS vocabulary, so do make sure you’re using the latest version of the LESS engine if you attempt to run this yourself. It does offer a great way to keep the sprite mixin as clear and concise as possible, though. Just pass the sprite you want to use to the sprite mixin, and LESS will figure out the rest for you.

Common CSS Sprite Use Cases

With these mixins at our disposal, let’s see what different CSS sprite use cases we can identify and find out whether capturing these use cases in additional LESS mixins is possible. Once again, in order to minimize complexity in the upcoming sections, we’ll assume you’re working with a square grid sprite. If you need a rectangular grid, just add the image’s parameter to the LESS mixins we’ll be working with, and everything should be fine.

For each use case, we’ll define two mixins: one with and one without height and width parameters. If you’re like me and you keep misspelling the height and width keywords, the second mixin might come in handy. It does leave you the responsibility of learning the correct order of the parameters, though. LESS allows you to define these mixins in the same document, however, so there’s really no harm in listing them both, picking whichever is easiest for you.

1. REPLACED TEXT

This is probably the easiest use case, occurring when we have an **`html`** element at our disposal that can be given a fixed dimension (in order to ensure we can hide the unwanted parts of the sprite). We'll also be hiding the text inside the **`html`** element, replacing it with an image from our sprite. Typical examples of this use case are action links (think delete icons, print links, RSS icons, etc.) and headings (although CSS3 and Web fonts are quickly making this technique obsolete).

```
.hideText{
   text-indent: -999em;
   letter-spacing: -999em;
   overflow: hidden;
}
.spriteReplace(@x, @y) {
   .sprite(@x, @y);
   .hideText;
}
.spriteReplace (@x, @y, @width, @height) {
   .sprite(@x, @y);
   .hideText;
   width: @width;
   height: @height;
}
```

The **`spriteReplace`** mixin simply wraps our former sprite mixin and adds a small helper mixin to hide the text from view. It's pretty basic, but it does save us the trouble of adding the **`.hideText`** mixin manually for every instance of this use case.

Using sprites for replaced elements.

In the example above, we have a list of sharing options. For whatever reason (theming or just personal preference), let's say we decide to use CSS backgrounds instead of HTML images. Nothing to it with the mixins we just created. Here's the code (assuming we're using the reference image shown earlier):

```
<ul class="sharing">
   <li class="twitter"><a href="#">Share [article's title] on
Twitter</a></li>
   …
</ul>
.sharing .twitter a {
   .spriteReplace(0, 0, 32px, 32px); display:block;
}
```

2. INLINE IMAGES

For the second use case, we'll tackle inline images. The main problem we're facing here is that we won't be able to put fixed dimensions on the **`html`** element itself because we're dealing with variable-sized content. Typical uses of inline images are for icons next to text links (for example, external links, download links, etc.), but this method can also be used for any other item that allows text to be wrapped around a sprite image.

```
.spriteInline(@x, @y) {
   .sprite(@x, @y);
   display: inline-block;
   content: "";
}
.spriteInline(@x, @y, @width, @height) {
   .sprite(@x, @y);
   display: inline-block;
   content: "";
   width: @width;
   height: @height;}
```

We might be lacking a structural element to visualize our image, but 2011 taught us that pseudo-elements are the perfect hack to overcome this problem (Nicolas Gallagher's eye-opening article on using pseudo-elements for background images explains everything in detail). This is why the **`spriteInline`** mixin was especially developed to be used within a **`:before`** or **`:after`** selector (depending on where the image needs to appear). We'll add an **`inline-block`** declaration to the pseudo-element so that it behaves like an inline element while still allowing us to add fixed dimensions to it, and we'll add the `content` property, which is needed just for visualizing the pseudo-element.

Using sprites for inline images.

The example above shows a list of affiliate links. The design demands that each link be limited to one line, so we can safely use the **`.spriteInline`** mixin to display the icons in front of each link:

```
<ul class="affiliates">
   <li class="amazon"><a href="#">Buy on Amazon.com</a></li>
   …
</ul>
.affiliates .amazon a:before {
   .spriteInline(4, 1, 22px, 16px);
}
```

3. PADDED IMAGES

The third and final use case comes up when text isn't allowed to wrap around a sprite image. Typical examples are list items that span multiple lines, and all kinds of visual notifications that bare icons. Whenever you want to reserve space on a multi-line element to make sure the text lines up neatly next to the image, this is the method you'll want to use.

```
.spritePadded(@x, @y) {
   .sprite(@x, @y);
   position: absolute;
   content: "";
}
.spritePadded(@x, @y, @width, @height) {
   .sprite(@x, @y);
   position: absolute;
   content: "";
   width: @width;
   height: @height;
}
```

Again we'll try our luck with pseudo-elements; this time, though, we'll be performing some positioning tricks. By applying a **`position: absolute`** to the pseudo-element, we can place it right inside the space reserved by its parent element (usually by applying padding to the parent—hence, the name of the mixin). The actual position of the image (the **`left, right, top`** and **`bottom`** properties) is not added to the **`spritePadded`** mixin and should be done in the selector itself to keep things as clean and simple as possible (in this case, by maintaining a low parameter count).

Because we're using absolute positioning, either the **`:before`** or **`:after`** pseudo-element will do the job. Keep in mind, though, that the **`:after`** element is a likely candidate for CSS clearing methods, so if you want to avoid future conflicts (because the clearing fix won't work if you add a

`position: absolute` to the **`:after`** pseudo-element), you'd be safest applying the sprite style to the **`:before`** element.

Using sprites for padded elements.

Let's assume we need to indicate that our article is available in other languages (probably on a different page or even website). We'll add a little notification box, listing the different translations of the current article. If the text breaks onto a second line, though, we do not want it to crawl below our little icon, so we'll use the **`spritePadded`** mixin:

```
<section class="notification translated">
   <p>Translation available…</p>
</section>
.translated p {
   padding-left: 22px;
   position: relative;
}
.translated p:before {
   .spritePadded(5, 5, 16px, 14px);
   left: 0;
   top: 0;
}
```

The Ideal Sprite

So, have we achieved CSS sprite wizardry here? Heck, no! If you've been paying close attention, you might have noticed that the use cases above offer no solution for adding repeating backgrounds to a sprite. While there

are some ways around this problem, none of them are very efficient or even worthy of recommendation.

What CSS sprites need in order to truly flourish is a CSS solution for cropping an image. Such a property should define a crop operation on a given background image before the image is used for further CSS processing. This way, it would be easy to crop the desired image from the sprite and position it accordingly, repeating it as needed; we'd be done with these `:before` and `:after` hacks because there wouldn't be anything left to hide. A solution has been proposed in the Editor's Draft of the CSS Image Values and Replaced Content Module Level 3 (section 3.2.1), but it could take months, even years, for this to become readily available.

For now, the LESS mixins above should prove pretty helpful if you plan to use CSS sprites. Just remember to prepare your sprite well; if you do, things should move ahead pretty smoothly, even when deadlines are weighing on you.

Learning To Use The :before And :after Pseudo-Elements In CSS

Louis Lazaris

If you've been keeping tabs on various Web design blogs, you've probably noticed that the `:before` and `:after` pseudo-elements have been getting quite a bit of attention in the front-end development scene — and for good reason. In particular, the experiments of one blogger — namely, London-based developer Nicolas Gallagher — have given pseudo-elements quite a bit of exposure of late.

Nicolas Gallagher used pseudo-elements to create 84 GUI icons created from semantic HTML.

To complement this exposure (and take advantage of a growing trend), I've put together what I hope is a fairly comprehensive run-down of pseudo-elements. This article is aimed primarily at those of you who have seen some of the cool things done with pseudo-elements but want to know what this CSS technique is all about before trying it yourself.

Although the CSS specification contains other pseudo-elements, I'll focus on **`:before`** and **`:after`**. So, for brevity, I'll say "pseudo-elements" to refer generally to these particular two.

What Does A Pseudo-Element Do?

A pseudo-element does exactly what the word implies. It creates a phoney element and inserts it before or after the content of the element that you've targeted.

The word "pseudo" is a transliteration of a Greek word that basically means "lying, deceitful, false." So, calling them *pseudo*-elements is appropriate, because they don't actually change anything in the document. Rather, they insert ghost-like elements that are visible to the user and that are style-able in the CSS.

Basic Syntax

The **`:before`** and **`:after`** pseudo-elements are very easy to code (as are most CSS properties that don't require a ton of vendor prefixes). Here is a simple example:

```
#example:before {
   content: "#";
}
#example:after {
   content: ".";
}
```

There are two things to note about this example. First, we're targeting the same element using **`#example:before`** and **`#example:after`**. Strictly speaking, they are the pseudo-elements in the code.

Secondly, without the **`content`** property, which is part of the generated content module in the specification, pseudo-elements are useless. So, while the pseudo-element selector itself is needed to target the element, you won't be able to insert anything without adding the **`content`** property.

In this example, the element with the id **example** will have a hash symbol placed “before” its content, and a period (or full stop) placed “after” its content.

SOME NOTES ON THE SYNTAX

You could leave the **content** property empty and just treat the pseudo-element like a content-less box, like this:

```
#example:before {
   content: "";
   display: block;
   width: 100px;
   height: 100px;
}
```

However, you can’t remove the **content** property altogether. If you did, the pseudo-element wouldn’t work. At the very least, the **content** property needs empty quotes as its value.

You may have noticed that you can also code pseudo-elements using the double-colon syntax (**::before** and **::after**), which I’ve discussed before. The short explanation is that there is no difference between the two syntaxes; it’s just a way to differentiate pseudo-elements (double colon) from pseudo-classes (single colon) in CSS3.

One final point regarding the syntax. Technically, you could implement a pseudo-element universally, without targeting any element, like this:

```
:before {
   content: "#";
}
```

While the above is valid, it’s pretty useless. The code will insert a hash symbol before the content in each element in the DOM. Even if you

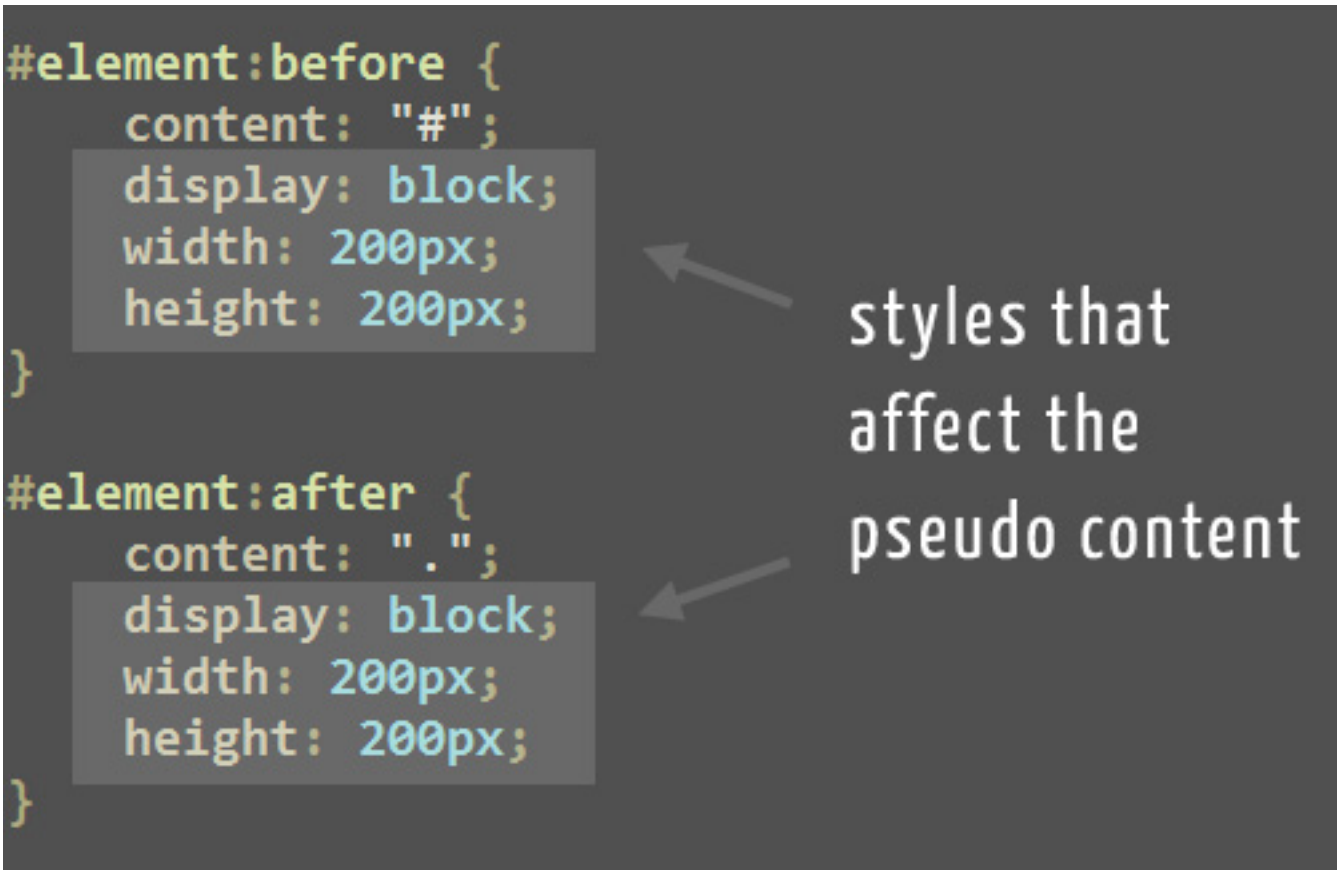
removed the **<body>** tag and all of its content, you'd still see two hash symbols on the page: one in the **<html>** element, and one in the **<body>** tag, which the browser automatically constructs.

Characteristics Of Inserted Content

As mentioned, the content that is inserted is not visible in the page's source. It's visible only in the CSS.

Also, the inserted element is by default an inline element (or, in HTML5 terms, in the category of text-level semantics). So, to give the inserted element a height, padding, margins and so forth, you'll usually have to define it explicitly as a block-level element.

This leads well into a brief description of how to style pseudo-elements. Look at this graphic from my text editor:



In this example, I've highlighted the styles that will be applied to the elements inserted before and after the targeted element's content. Pseudo-elements are somewhat unique in this way, because you insert the content *and* the styles in the same declaration block.

Also note that typical CSS inheritance rules apply to the inserted elements. If you had, for example, a font stack of **`Helvetica, Arial, sans-serif`** applied to the **`<body>`** element of the document, then the pseudo-element would inherit that font stack the same as any other element would.

Likewise, pseudo-elements don't inherit styles that aren't naturally inherited from parent elements (such as padding and margins).

Before Or After What?

Your hunch on seeing the **`:before`** and **`:after`** pseudo-elements might be that the inserted content will be injected before and after the targeted element. But, as alluded to above, that's not the case.

The content that's injected will be child content in relation to the targeted element, but it will be placed "before" or "after" any other content in that element.

To demonstrate this, look at the following code. First, the HTML:

```
<p class="box">Other content.</p>
```

And here's the CSS that inserts a pseudo-element:

```
p.box {
   width: 300px;
   border: solid 1px white;
   padding: 20px;
}

p.box:before {
   content: "#";
   border: solid 1px white;
   padding: 2px;
   margin: 0 10px 0 0;
}
```

In the HTML, all you would see is a paragraph with a class of **box**, with the words "Other content" inside it (the same as what you would see if you viewed the source on the live page). In the CSS, the paragraph is given a set width, along with some padding and a visible border.

Then we have the pseudo-element. In this case, it's a hash symbol inserted "before" the paragraph's content. The subsequent CSS gives it a border, along with some padding and margins.

Here's the result viewed in the browser:

The outer box is the paragraph. The border around the hash symbol denotes the boundary of the pseudo-element. So, instead of being inserted "before" the paragraph, the pseudo-element is placed before the "Other content" in the paragraph.

Inserting Non-Text Content

I mentioned briefly that you can leave the **`content`** property's value as an empty string or insert text content. You basically have two additional options of what to include as the value of the **`content`** property.

First, you can include a URL that points to an image, just as you would do when including a background image in the CSS:

```
p:before {
    content: url(image.jpg);
}
```

Notice that the quotes are missing. If you wrapped the URL reference in quotes, then it would become a literal string and insert the text "url(image.jpg)" as the content, instead of inserting the image itself.

Naturally, you could include a Data URI in place of the image reference, just as you can with a CSS background.

You also have the option to include a function in the form of **`attr(X)`**. This function, according to the spec, "returns as a string the value of attribute X for the subject of the selector."

Here's an example:

```
a:after {
    content: attr(href);
}
```

What does the **`attr()`** function do? It takes the value of the specified attribute and places it as text content to be inserted as a pseudo-element.

The code above would cause the `href` value of every **<a>** element on the page to be placed immediately after each respective **<a>** element. This could be used in a print style sheet to include full URLs next to all links when a document is printed.

You could also use this function to grab the value of an element's **`title`** attribute, or even micro data values. Of course, not all of these examples would be practical in and of themselves; but depending on the situation, a specific attribute value could be practical as a pseudo-element.

While being able to grab the **`title`** or **`alt`** text of an image and display it on the page as a pseudo-element would be practical, this isn't possible. Remember that the pseudo-element must be a child of the element to which it is being applied. Images, which are void (or empty) elements, don't have child elements, so it wouldn't work in this case. The same would apply to other void elements, such as **`<input>`**.

Dreaded Browser Support

As with any front-end technology that is gaining momentum, one of the first concerns is browser support. In this case, that's not as much of a problem.

Browser support for **`:before`** and **`:after`** pseudo-elements stacks up like this:

- Chrome 2+,
- Firefox 3.5+ (3.0 had partial support),
- Safari 1.3+,
- Opera 9.2+,
- IE8+ (with some minor bugs),
- Pretty much all mobile browsers.

The only real problem (no surprise) is IE6 and IE7, which have no support. So, if your audience is in the Web development niche (or another market that has low IE numbers), you can probably go ahead and use pseudo-elements freely.

PSEUDO-ELEMENTS AREN'T CRITICAL

Fortunately, a lack of pseudo-elements will not cause huge usability issues. For the most part, pseudo-elements are generally decorative (or helper-like) content that will not cause problems in unsupported browsers. So, even if your audience has high IE numbers, you can still use them to some degree.

A Couple Of Reminders

As mentioned, pseudo-element content does not appear in the DOM. These elements are not real elements. As such, they are not accessible to most assistive devices. So, never use pseudo-elements to generate content that is critical to the usability or accessibility of your pages.

Another thing to keep in mind is that developer tools such as Firebug do not show the content generated by pseudo-elements. So, if overused, pseudo-elements could cause maintainability headaches and make debugging a much slower process.

(**Update:** you can use Chrome's developer tools to view the styles associated with a pseudo-element, but the element will not appear in the DOM. Also, Firebug is adding pseudo-element support in version 1.8.)

That covers all of the concepts you need in order to create something practical with this technique. In the meantime, for further reading on CSS pseudo-elements, be sure to check out some of the articles that we've linked to in this piece.

Taming Advanced CSS Selectors

Inayaili de Leon

CSS is one of the most powerful tools that is available to web designers (if not the most powerful). With it we can completely transform the look of a website in just a couple of minutes, and without even having to touch the markup. But despite the fact that we are all well aware of its usefulness, CSS selectors are still not used to their full potential and we sometimes have the tendency to litter our HTML with excessive and unnecessary classes and ids, divs and spans.

The best way to avoid these plagues spreading in your markup and keep it clean and semantic, is by using more complex CSS selectors, ones that can target specific elements without the need of a class or an id, and by doing that keep our code and our stylesheets flexible.

CSS Specificity

Before delving into the realms of advanced CSS selectors, it's important to understand how CSS specificity works, so that we know how to properly use our selectors and to avoid us spending hours debugging for a CSS issue that could be easily fixed if we had only payed attention to the specificity.

When we are writing our CSS we have to keep in mind that some selectors will rank higher than others in the cascade, the latest selector that we wrote will not always override the previous ones that we wrote for the same elements.

So how do you calculate the specificity of a particular selector? It's fairly straightforward if you take into account that specificity will be represented as four numbers separated by commas, like: 1, 1, 1, 1 or 0, 2, 0, 1

1. The first digit (a) is always zero, unless there is a style attribute applied to that element within the markup itself
2. The second digit (b) is the sum of the number of IDs in that selector
3. The third digit (c) is the sum of other attribute selectors and pseudo-classes in that selector. Classes (**`.example`**) and attribute selectors (eg. **`li[id=red]`**) are included here.
4. The fourth digit (d) counts the elements (like **`table`, `p`, `div`,** etc.) and pseudo-elements (like **`:first-line`**)
5. The universal selector (*) has a specificity of zero
6. If two selectors have the same specificity, the one that comes last on the stylesheet will be applied

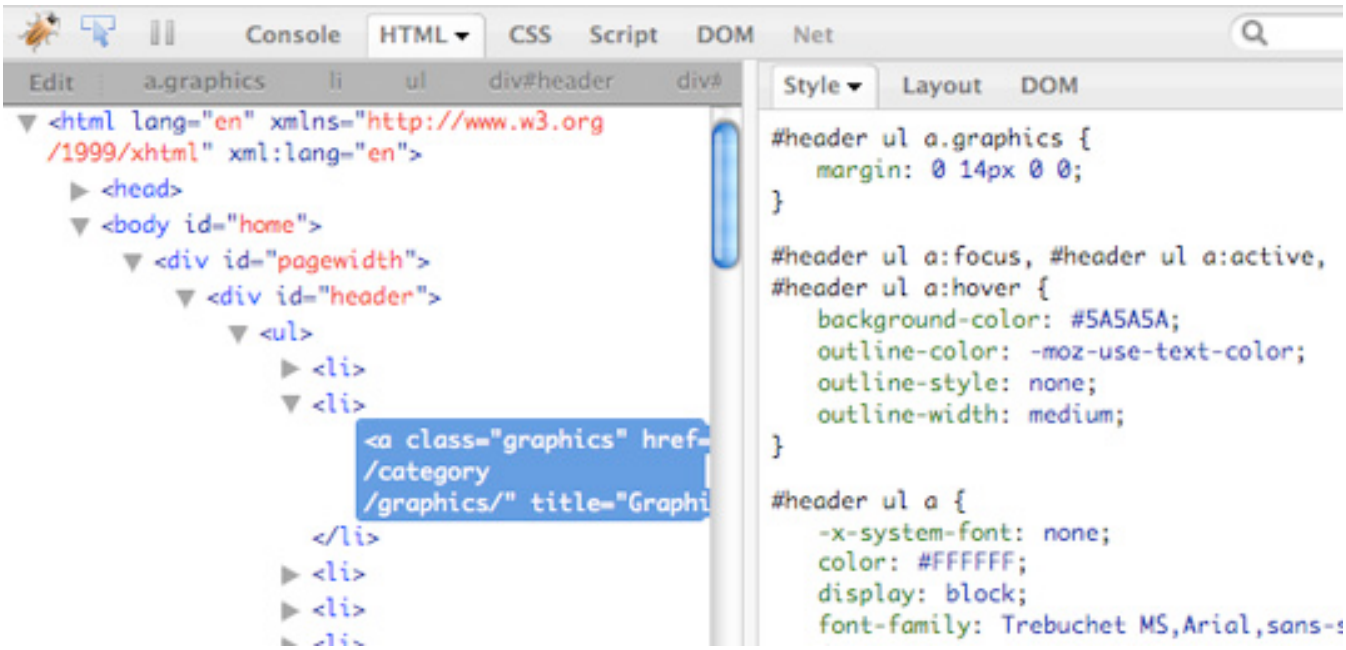
Let's take a look at a few examples, to make it easier to understand:

- **`#sidebar h2`** — $0, 1, 0, 1$
- **`h2.title`** — 0, 0, 1, 1
- **`h2 + p`** — 0, 0, 0, 2
- **`#sidebar p:first-line`** — $0, 1, 0, 2$

From the following selectors, the first one is the one who will be applied to the element, because it has the higher specificity:

- **`#sidebar p#first { color: red; }`** — $0, 2, 0, 1$
- **`#sidebar p:first-line { color: blue; }`** — $0, 1, 0, 2$

It's important to have at least a basic understanding of how specificity works, but tools like *Firebug* are useful to let us know which selector is being applied to a particular element by listing all the CSS selectors in order of their specificity when you are inspecting an element.



Firebug lets you easily see which selector is being applied to an element.

1. Attribute selectors

Attribute selectors let you target an element based on its attributes. You can specify the element's attribute only, so all the elements that *have* that attribute — whatever the value — within the HTML will be targeted, or be more specific and target elements that have particular values on their attributes — and this is where attribute selectors show their power.

There are 6 different types of attribute selectors:

- **`[att=value]`**
  The attribute has to have the exact value specified.

- **`[att~=value]`**
  The attribute's value needs to be a whitespace separated list of words (for example, class="title featured home"), and one of the words is exactly the specified value.
- **`[att|=value]`**
  The attribute's value is exactly "value" or starts with the word "value" and is immediately followed by "-", so it would be "value-".
- **`[att^=value]`**
  The attribute's value starts with the specified value.
- **`[att$=value]`**
  The attribute's value ends with the specified value.
- **`[att*=value]`**
  The attribute's value contains the specified value.

For example, if you want to change the background color of all the **`div`** elements that are posts on your blog, you can use the an attribute selector that targets every **`div`** whose **`class`** attribute starts with "post-"**:**

```
div[class*="post"] {
  background-color: #333;
  }
```

This will match all the **`div`** elements whose class attribute contains the words "posts", in any position.

Another useful usage of attribute selectors is to target different types of **`input`** elements**.** For example, if you want your text inputs to have a different width from the others, you can use a simple attribute selector:

```
input[type="text"] {
  width: 200px;
  }
```

This will target all the **input** elements whose **type** attribute is exactly "text".

Now let's say you want to add a different icon next to each different type of file your website is linking to, so your website's visitors know when they'll get an image, a PDF file, a Word document, etc. This can be done by using an attribute selector:

```
a[href$=".jpg"] {
  background: url(jpeg.gif) no-repeat left 50%;
  padding: 2px 0 2px 20px;
  }

a[href$=".pdf"] {
  background: url(pdf.gif) no-repeat left 50%;
  padding: 2px 0 2px 20px;
  }

a[href$=".doc"] {
  background: url(word.gif) no-repeat left 50%;
  padding: 2px 0 2px 20px;
  }
```

In this example, we've used an attribute selector that will target all the links (**a**) whose **href** attribute ends (**$**) with .jpg, .pdf or .doc.

Notes on browser support

Apart from Internet Explorer 6, all major browsers support attribute selectors. This means that when you are using attribute selectors on your stylesheets, you should make sure that IE6 users will still be provided with a usable site. Take our third example: adding an icon to your links adds

another level of usability to your site, but the site will still be usable if the links don't show any icons.

2. Child selector

The child selector is represented by the sign "**>**". It allows you to **target** elements that are direct children of a particular element.

For example, if you want to match all the **h2** elements that are a direct child of your sidebar **div**, but not the **h2** elements that may be also within the **div**, but that are grandchildren (or later descendants) of your element, you can use this selector:

```
div#sidebar > h2 {
  font-size: 20px;
  }
```

You can also use both child and descendant selectors combined. For example, if you want to target only the **blockquote** elements that are within **div**s that are direct grandchildren of the **body** element (you may want to match blockquotes inside the main content **div**, but not if they are outside it):

```
body > div > div blockquote {
  margin-left: 30px;
  }
```

Notes on browser support
Like the attribute selectors, the child selector is not supported by Internet Explorer 6. If the effect you are trying to achieve by using it is crucial for the website's usability or overall aesthetics, you can consider using a class selector with it, or on a IE-only stylesheet, but that would detract from the purpose of using child selectors.

3. Sibling combinators

There are two types of sibling combinators: adjacent sibling combinators and general sibling combinators.

ADJACENT SIBLING COMBINATOR

This selector uses the plus sign, "**+**", to combine two sequences of simple selectors. The elements in the selector have the same parent, and the second one must come immediately after the first.

The adjacent sibling combinator can be very useful, for example, when dealing with text. Lets say you want to add a top margin to all the **`h2`** tags that follow a paragraph (you don't need to add a top margin if the heading comes after an **`h1`** tag or if it's the first element on that page):

```
p + h2 {
  margin-top: 10px;
  }
```

You can be even more specific and say that you only want this rule applied if the elements are within a particular **`div`**:

```
div.post p + h2 {
  margin-top: 10px;
  }
```

Or you can add another level of complexity: say you want the first line of the paragraphs of every page to be in small caps.

```
.post h1 + p:first-line {
  font-variant: small-caps;
  }
```

Because you know that the first paragraph of every post immediately follows an **h1** tag, you can refer to the **h1** on your selector.

GENERAL SIBLING COMBINATOR

The general sibling combinator works pretty much the same as the adjacent sibling combinator, but with the difference that the second selector doesn't have to immediately follow the first one.

So if you need to target all the **p** tags that are within a particular **div** and that follow the **h1** tag (you may want those **p** tags to be larger than the ones that come before the title of your post), you can use this selector:

```
.post h1 ~ p {
  font-size: 13px;
  }
```

Notes on browser support
Internet Explorer 6 doesn't understand sibling combinators, but, as for the other cases, if your audience includes a small percentage of IE6 users, and if the website's layout isn't broken or severely affected by its lack of support, this is a much easier way of achieving lots of cool effects without the need of cluttering your HTML with useless classes and ids.

4. Pseudo-classes

DYNAMIC PSEUDO-CLASSES

These are called dynamic pseudo-classes because they actually do not exist within the HTML: they are only present when the user is or has interacted with the website.

There are two types of dynamic pseudo-classes: link and user action ones. The link are **`:link`** and **`:visited`**, while the user action ones are **`:hover, :active`** and **`:focus.`**

From all the CSS selectors mentioned in this post, these will probably be the ones that are most commonly used.

The **`:link`** pseudo-class applies to links that haven't been visited by the user, while the **`:visited`** pseudo-class applies to links that have been visited, so they are mutually exclusive.

The **`:hover`** pseudo-class applies when the user moves the cursor over the element, without having to activate or click on it. The **`:active`** pseudo-class applies when the user actually clicks on the element. And finally the **`:focus`** pseudo-class applies when that element is on focus — the most common application is on form elements.

You can use more than one user action dynamic pseudo-class in your stylesheets, so you can have, for example, a different background color for an input field depending on whether the user's cursor is only hovering over it or hovering over it while in focus:

```
input:focus {
  background: #D2D2D2;
  border: 1px solid #5E5E5E;
  }

input:focus:hover {
  background: #C7C7C7;
  }
```

Notes on browser support

The dynamic pseudo-classes are supported by all modern browsers, even IE6. But bear in mind that IE6 only allows the **`:hover`** pseudo-class to be

applied to link elements **(a)** and only IE8 accepts the **`:active`** state on elements other than links.

:FIRST-CHILD

The **`:first-child`** pseudo-class allows you to target an element that is the first child of another element. For example, if you want to add a top margin to the first `li` element of your unordered lists, you can have this:

```
ul > li:first-child {
  margin-top: 10px;
  }
```

Let's take another example: you want all your **`h2`** tags in your sidebar to have a top margin, to separate them from whatever comes before them, but the first one doesn't need a margin. You can use the following code:

```
#sidebar > h2 {
  margin-top: 10px;
  }
#sidebar > h2:first-child {
  margin-top: 0;
  }
```

Notes on browser support

IE6 doesn't support the **`:first-child`** pseudo-class. Depending on the design that the pseudo-class is being applied to, it may not be a major cause for concern. For example, if you are using the **`:first-child`** selector to remove top or bottom margins from headings or paragraphs, your layout will probably not break in IE6, it will only look sightly different. But if you are using the **`:first-child`** selector to remove left and right margins from, for example, a floated sequence of **`div`**s, that may cause more disruption to your designs.

THE LANGUAGE PSEUDO-CLASS

The language pseudo-class, **`:lang()`**, allows you to match an element based on its language.

For example, lets say you want a specific link on your site to have a different background color, depending on that page's language:

```
:lang(en) > a#flag {
  background-image: url(english.gif);
  }
:lang(fr) > a#flag {
  background-image: url(french.gif);
  }
```

The selectors will match that particular link if the page's language is either equal to "en" or "fr" or if it starts with "en" or "fr" and is immediately followed by an "-".

Notes on browser support
Not surprisingly, the only version of Internet Explorer that supports this selector is 8. All other major browsers support the language pseudo-selector.

5. CSS 3 Pseudo-classes

:TARGET

When you're using links with fragment identifiers (for example, http://www.smashingmagazine.com/2009/08/02/bauhaus-ninety-years-of-inspiration/#comments, where "**`#comments`**" is the fragment identifier), you can style the target by using the :target pseudo-class.

For example, lets imagine you have a long page with lots of text and **h2** headings, and there is an index of those headings at the top of the page. It will be much easier for the user if, when clicking on a particular link within the index, that heading would become highlighted in some way, when the page scrolls down. Easy:

```
h2:target {
  background: #F2EBD6;
  }
```

Notes on browser support

This time, Internet Explorer is really annoying and has no support at all for the **:target** pseudo-class. Another glitch is that Opera doesn't support this selector when using the back and forward buttons. Other than that, it has support from the other major browsers.

THE UI ELEMENT STATES PSEUDO-CLASSES

Some HTML elements have an enable or disabled state (for example, input fields) and checked or unchecked states (radio buttons and checkboxes). These states can be targeted by the **:enabled**, **:disabled** or **:checked** pseudo-classes, respectively.

So you can say that any **input** that is disabled should have a light grey background and dotted border:

```
input:disabled {
  border:1px dotted #999;
  background:#F2F2F2;
  }
```

You can also say that all checkboxes that are checked should have a left margin (to be easily seen within a long list of checkboxes):

```
input[type="checkbox"]:checked {
  margin-left: 15px;
  }
```

Notes on browser support

All major browsers, except our usual suspect, Internet Explorer, support the UI element states pseudo-classes. If you consider that you are only adding an extra level of detail and improved usability to your visitors, this can still be an option.

6. CSS 3 structural pseudo-classes

:NTH-CHILD

The **`:nth-child()`** pseudo-class allows you to target one or more specific children of a parent element.

You can target a single child, by defining its value as an integer:

```
ul li:nth-child(3) {
  color: red;
  }
```

This will turn the text on the third **`li`** item within the **`ul`** element red. Bear in mind that if a different element is inside the **`ul`** (not a **`li`**), it will also be counted as its child.

You can target a parent's children using expressions. For example, the following expression will match every third **`li`** element starting from the fourth:

```
ul li:nth-child(3n+4) {
  color: yellow;
  }
```

In the previous case, the first yellow **`li`** element will be the fourth. If you just want to start counting from the first **`li`** element, you can use a simpler expression:

```
ul li:nth-child(3n) {
  color: yellow;
  }
```

In this case, the first yellow **`li`** element will be the third, and every other third after it. Now imagine you want to target only the first four **`li`** elements within the list:

```
ul li:nth-child(-n+4) {
  color: green;
  }
```

The value of **`:nth-child`** can also be defined as “even” or “odd”, which are the same as using “2n” (every second child) or “2n+1” (every second child starting from the first), respectively.

:NTH-LAST-CHILD

The **`:nth-last-child`** pseudo-class works basically as the **`:nth-child`** pseudo-class, but it starts counting the elements from the last one.

Using one of the examples above:

```
ul li:nth-child(-n+4) {
  color: green;
  }
```

Instead of matching the first four **`li`** elements in the list, this selector will match the *last* four elements.

You can also use the values “even” or “odd”, with the difference that in this case they will count the children starting from the last one:

```
ul li:nth-last-child(odd) {
  color: grey;
  }
```

:NTH-OF-TYPE

The **`:nth-of-type`** pseudo-class works just like the **`:nth-child`**, with the difference that it only counts children that match the element in the selector.

This can be very useful if we want to target elements that may contain different elements within them. For example, let’s imagine we want to turn every second paragraph in a block of text blue, but we want to ignore other elements such as images or quotations:

```
p:nth-of-type(even) {
  color: blue;
  }
```

You can use the same values as you would use for the **`:nth-child`** pseudo-class.

:NTH-LAST-OF-TYPE

You guessed it! The **`:nth-last-of-type`** pseudo-class can be used exactly like the aforementioned **`:nth-last-child`**, but this time, it will only target the elements that match our selector:

```
ul li:nth-last-of-type(-n+4) {
  color: green;
  }
```

We can be even more clever, and combine more than one of these pseudo-classes together on a massive selector. Let's say all images within a post **`div`** to be floated left, except for the first and last one (let's image these would full width, so they shouldn't be floated):

```
.post img:nth-of-type(n+2):nth-last-of-type(n+2) {
  float: left;
  }
```

So in the first part of this selector, we are targeting every image starting from the second one. In the second part, we are targeting every image except for the last one. Because the selectors aren't mutually exclusive, we can use them both on one selector thus excluding both the first and last element at once!

:LAST-CHILD

The **`:last-child`** pseudo-class works just as the **`:first-child`** pseudo-class, but instead targets the last child of a parent element.

Let's image you don't want the last paragraph within your post **`div`** to have a bottom margin:

```
.post > p:last-child {
  margin-bottom: 0;
  }
```

This selector will target the last paragraph that is a direct *and* the last child of an element with the class of "post".

:FIRST-OF-TYPE AND :LAST-OF-TYPE

The **`:first-of-type`** pseudo-class is used to target an element that is the first of its type within its parent.

For example, you can target the first paragraph that is a direct child of a particular **div**, and capitalize its first line:

```
.post > p:first-of-type:first-line {
  font-variant: small-caps;
  }
```

With this selector you make sure that you are targeting only paragraphs that are direct children of the "post" **div**, and that are the first to match our **p** element.

The **:last-of-type** pseudo-class works exactly the same, but targets the *last* child of its type instead.

:ONLY-CHILD

The **:only-child** pseudo-class represents an element that is the only child of its parent.

Let's say you have several boxes ("news") with paragraphs of text inside them. When you have more than one paragraph, you want the text to be smaller than when you have only one:

```
div.news > p {
  font-size: 1.2em;
  }
div.news > p:only-child {
  font-size: 1.5em;
  }
```

In the first selector, we are defining the overall size of the **p** elements that are direct children of a "news" **div.** On the second one, we are overriding the previous font-size by saying, if the **p** element is the only child of the "news" **div**, its font size should be bigger.

:ONLY-OF-TYPE

The **`:only-of-type`** pseudo-class represents an element that is the only child of its parent with the same element.

How can this be useful? Image you have a sequence of posts, each one represented by a **`div`** with the class of "post". Some of them have more than one image, but others have only one image. You want the image within the later ones to be aligned to the center, while the images on posts with more than one image to be floated. That would be quite easy to accomplish with this selector:

```
.post > img {
  float: left;
  }
.post > img:only-of-type {
  float: none;
  margin: auto;}
```

:EMPTY

The **`:empty`** pseudo-class represents an element that has no content within it.

It can be useful in a number of ways. For example, if you have multiple boxes in your "sidebar" **`div`**, but don't want the empty ones to appear on the page:

```
#sidebar .box:empty {
  display: none;
  }
```

Beware that even if there is a single space in the "box" **`div`**, it will not be treated as empty by the CSS, and therefore will not match the selector.

Notes on browser support

Internet Explorer (up until version 8) has no support for structural pseudo-classes. Firefox, Safari and Opera support these pseudo-classes on their latest releases. This means that if what it's being accomplished with these selectors is fundamental for the website's usability and accessibility, or if the larger part of the website's audience is using IE and you don't want to deprive them of some design details, it's be wise to keep using regular classes and simpler selectors to cater for those browsers. If not, you can just go crazy!

7. The negation pseudo-class

The negation pseudo-class, **`:not()`**, lets you target elements that do not match the selector that is represented by its argument.

For example, this can be useful if you need to style all the **`input`** elements within a form, but you don't want your input elements with the type submit to be styled — you want them to be styled in a different way —, to look more like buttons:

```
input:not([type="submit"]) {
  width: 200px;
  padding: 3px;
  border: 1px solid #000000;
  }
```

Another example: you want all the paragraphs within your post **`div`** to have a larger font-size, except for the one that indicates the time and date:

```
.post p:not(.date) {
  font-size: 13px;
  }
```

Can you image the number of possibilities this selector brings with it, and the amount of useless selectors you could strip out off your CSS files were it widely supported?

Notes on browser support
Internet Explorer is our usual party pooper here: no support at all, not even on IE8. This probably means that this selector will still have to wait a while before some developers lose the fear of adding them to their stylesheets.

8. Pseudo-elements

Pseudo-elements allow you to access elements that don't actually exist in the HTML, like the first line of a text block or its first letter.

Pseudo-elements exist in CSS 2.1, but the CSS 3 specifications state that they should be used with the double colon "**::**", to distinguish them from pseudo-classes. In CSS 2.1, they are used with only one colon, "**:**". Browsers should be able accept both formats, except in the case of pseudo-elements that may be introduced only in CSS 3.

::FIRST-LINE

The **`::first-line`** pseudo-element will match the first line of a block, inline-block, table-caption or table-cell level element.

This is particularly useful to add subtle typographical details to your text blocks, like, for example, transforming the first line of an article into small caps:

```
h1 + p::first-line {
  font-variant: small-caps;
  }
```

If you've been paying attention, you'll know that this means the paragraph that comes *immediately after* an **h1** tag ("+") should have its first line in small caps.

You could also refer to the first line of a particular **div,** without having to refer to the actual paragraph tag:

```
div.post p::first-line { font-variant: small-caps; }
```

Or go one step farther and target specifically the *first* paragraph within a particular **div**:

```
div.post > p:first-child::first-line {
  font-variant: small-caps;
  }
```

Here, the "**>**" symbol indicates that you are targeting a direct child the post **div**, so if the paragraph were to be inside a second **div**, it wouldn't match this selector.

::FIRST-LETTER

The **::first-letter** pseudo-element will match the first letter of a block, unless it's preceded by some other content, like an image, on the same line.

Like the **::first-line** pseudo-element, **::first-letter** is commonly used to add typographical details to text elements, like drop caps or initials.

Here is how you could use the **::first-letter** pseudo-element to create a drop cap:

```
p {
  font-size: 12px;
  }
```

```
p::first-letter {
  font-size: 24px;
  float: left;
  }
```

Bear in mind that if you use both **`::first-line`** and **`::first-letter`** in the same element, the **`::first-letter`** properties will override the same properties inherited from **`::first-line.`**

This element can sometimes produce unexpected results, if you're not aware of the W3C specs: it's actually the CSS selector with the longest spec! So it's a good idea to read them carefully if you're planning on using it (as it is for all the other selectors).

::BEFORE AND ::AFTER

The **`::before`** and **`::after`** pseudo-elements are used to insert content before or after an element's content, purely via CSS.

These elements will inherit many of the properties of the elements that they are being attached to.

Image you want to the words "Graphic number x:" before the descriptions of graphs and charts on your page. You could achieve this without having to write the words "Graphic number", or the number itself yourself:

```
.post {
  counter-reset: image;
  }
p.description::before {
  content: "Figure number " counter(image) ": ";
  counter-increment: image;
  }
```

What just happened here?

First, we tell the HTML to create the “image” counter. We could have added this property to the body of the page, for example. Also, we can call this counter whatever name we want to, as long as we always reference it by the same name: try it for yourself!

Then we say that we want to add, before every paragraph with the class “description”, this piece of content: “Figure number ” — notice that only what we wrote between quotes will be created on the page, so we need to add the spaces as well!

After that, we have **`counter(image)`**: this will pick up the property we’ve already defined in the **`.post`** selector. It will by default start with the number one (1).

The next property is there so that the counter knows that for each **`p.description`**, it needs to increment the image counter by 1 (**`counter-increment: image`**).

It’s not as complicated as it looks, and it can be quite useful.

The **`::before`** and **`::after`** pseudo-elements are often only used with the content property, to add small sentences or typographical elements, but here it’s shown how we can use it in a more powerful way in conjunction with the **`counter-reset`** and **`counter-increment`** properties.

Fun fact: the **`::first-line`** and **`::first-letter`** pseudo-elements will match the content added by the **`::before`** pseudo-element, if present.

Notes on browser support

These pseudo-elements are supported by IE8 (not IE7 or 6), if the single colon format is used (for example, **`:first-letter,`** not **`::first-letter`**). All the other major browsers support these selectors.

Conclusion

Enough with the boring talk, now it's time for *you* to grab the information on this post and go try it for yourself: start by creating an experimental page and test all of these selectors, come back here when in doubt and make sure to always refer to the W3C specs, but don't just sit there thinking that because these selectors aren't yet widely supported you might as well ignore them.

If you're a bit more adventurous, or if you're not afraid of letting go of the past filled with useless and non-semantic classes and ids, why not sneak one or two of these powerful CSS selectors into your next project? We promise you'll never look back.

Six CSS Layout Features To Look Forward To

Divya Manian

A few concerns keep bobbing up now and then for Web developers, one of which relates to how to lay out a given design. Developers have made numerous attempts to do so with existing solutions. Several articles have been written on finding the holy grail of CSS layouts, but to date, not a single solution works without major caveats. At the W3Conf, I gave a talk on how the CSS Working Group is attempting to solve the concerns of Web developers with multiple proposals. There are six layout proposals that are relevant to us, all of which I described in the talk:

Here is a little more about these proposals and how they will help you in developing websites in the future.

Generated Content For Paged Media

- W3C Editor's Draft
- Demo
- Browser support: Opera Labs Build only

This proposal outlines a set of features that would modify the contents of any element to flow as pages, like in a book. A video demonstration shows how to use paged media to generate HTML5 slideshows (look at the demos for GCPM in the Opera Labs Build to play with the features more). To make the content of an element be paged, you would use this syntax:

```
@media paged {
  html {
    width: 100%;
    overflow-style: paged-x;
    padding: 5%;
    height: 100%;
    box-sizing: border-box;
  }
}
```

This would make the content look something like this:

Here, **`@media paged`** indicates that the browser understands paged media and that all of the selectors specified for it should have their styles applied when paged media is supported. Then, you indicate which selector you want to be paged (in the above example, the selector is the **`html`** element itself) by specifying the property **`overflow-style: paged-x.`** This will simply make the content paged; if you want paged controls to be visible, you need to specify **`overflow-style: paged-x-controls`**.

The properties **`break-before, break-after break-inside`** can be used to control where the content falls within the pages. For example, if you want headings to only appear with their corresponding content and never at the end of the page or standing alone, you can specify that:

```
h3, h2 {
break-after: avoid;
}
```

This ensures that if a heading occurs on the last line of a page, it will be pushed to the next page with the content that it introduces.

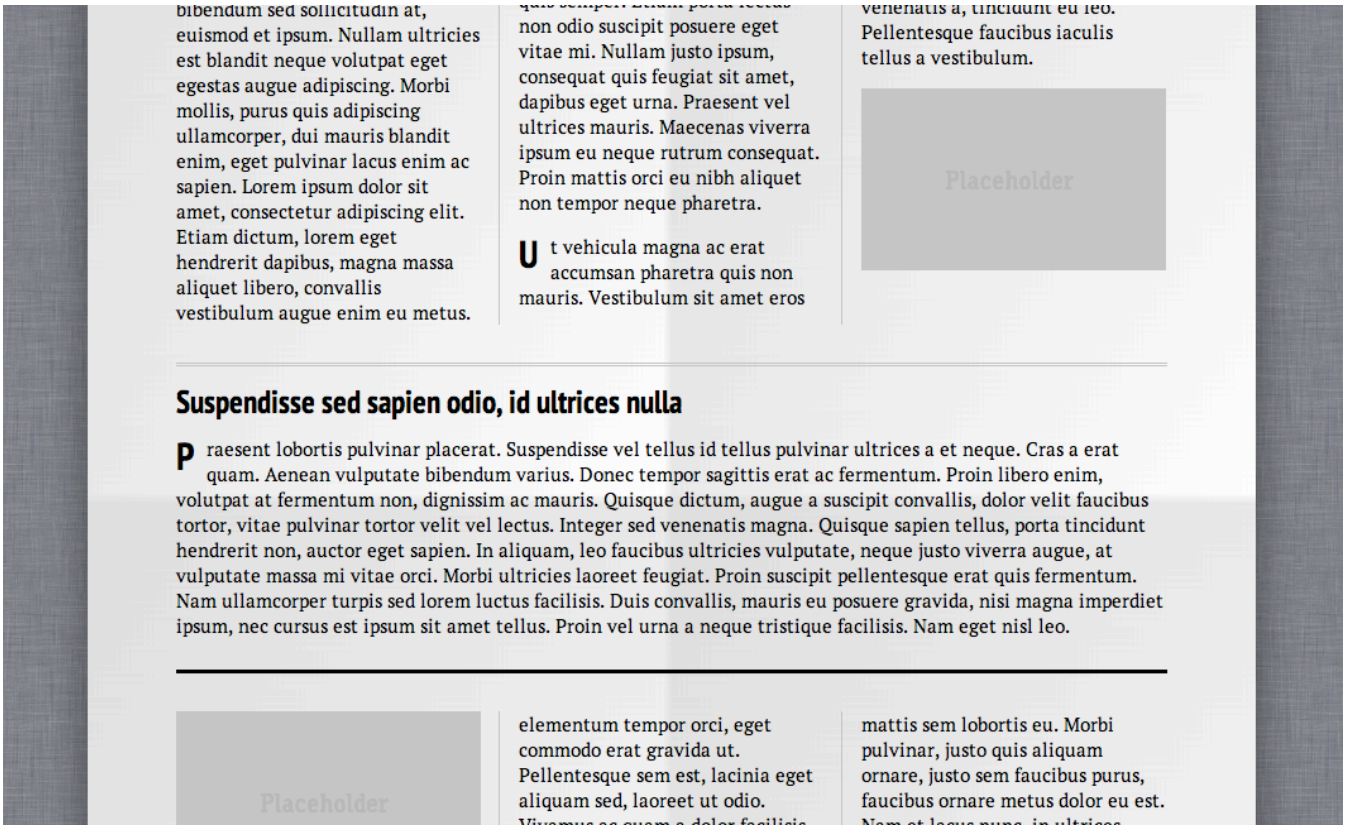
API

Two properties are available on an element whose content is paged: **`currentPage`** and **`pageCount`**. You can set the **`currentPage`** property via JavaScript, which would trigger a page change on that element. This would then trigger an **`onpagechange`** event on that element, which you could use to run other scripts that are required when the page changes. The **`pageCount`** property stores the total number of pages in the paged element. These properties are useful for implementing callbacks that should be triggered on page change; for example, to render notes for a particular slide in a slide deck.

Multiple Columns

- W3C Editor's Draft
- Demo
- Browser support: Opera 11.1+, Firefox 2+, Chrome 4+, Safari 3.1+, IE 10+

Multiple columns are now available in most browsers (including IE10!), which makes them pretty much ready to use on production websites. You can render the content of any element into multiple columns simply by using **`column-width: <length unit>`** or **`column-count: <number>`**. As with paged content, you can use **`break-before`**, **`break-after`** or **`break-inside`** to control how the content displays within each column. You can also make one of the child elements span the whole set of columns by using **`column-span: all`**. Here is how that would look:



Columns are balanced out with content by default. If you would prefer that columns not be balanced, you can set that by using **`column-fill: auto`** property. Here is an example of the default behavior (i.e. **`column-fill: balanced`**):

Excerpt from the Saga Of Hakon Herdebreid

Hakon, King Sigurd's son, was chosen chief of the troop which had followed King Eystein, and his adherents gave him the title of king. He was ten years old. At that time he had with him Sigurd, a son of Halvard Hauld of Reyr, and Andreas and Onund, the sons of Simon, his foster-brothers, and many chiefs, friends of King Sigurd and King Eystein; and they went first up to Gautland.

King Inge took possession of all the estates they had left behind, and declared them banished. Thereafter King Inge went to Viken, and was sometimes also in the north of the country. Gregorius Dagson was in Konungahella, where the danger was greatest, and had beside him a strong and handsome body of men, with which he defended the country.

And here is an example of the **`column-fill: auto`** behavior:

Excerpt from the Saga Of Hakon Herdebreid

Hakon, King Sigurd's son, was chosen chief of the troop which had followed King Eystein, and his adherents gave him the title of king. He was ten years old. At that time he had with him Sigurd, a son of Halvard Hauld of Reyr, and Andreas and Onund, the sons of Simon, his foster-brothers, and many chiefs, friends of King Sigurd and King Eystein; and they went first up to Gautland.

King Inge took possession of all the estates they had left behind, and declared them banished. Thereafter King Inge went to Viken, and was sometimes also in the north of the country. Gregorius Dagson was in Konungahella, where the danger was greatest, and had beside him a strong and handsome body of men, with which he defended the country.

Note that the last column is empty, and each column is filled one after the other.

Regions

- W3C Editor's Draft
- Demo
- Browser support: IE 10+, Chrome 15+, Safari 6+

The closest equivalent to regions would be InDesign's linking of text frames. With the properties in this proposal, you can make the content of selected elements flow through another set of elements. In other words, your content need not be tied to the document flow any longer.

To begin, you need to select elements whose content will be part of a "named flow," like this:

```
article.news { flow-into: article_flow; }
```

Here, all of the content in the **`article`** element with the class name **`news`** will belong to the flow named **`article_flow`**.

Then, you select elements that will render the contents that are part of this named flow:

```
#main {
flow-from: article_flow;
}
```

Here, the element with the ID **`main`** will be used to display the content in the flow named **`article_flow`**. This element has now become a region that renders the content of a named flow. Note that any element that is a region establishes new "block-formatting contexts" and "stacking contexts." For example, if a child element is part of a flow and is absolutely positioned, it will now only be absolutely positioned with respect to the region it belongs to, and not to the whole viewport.

You can also tweak the styles of content that flows through a region:

```
@region #main {
  p { color: indianred; }
}
```

Introduction **This is an example that shows how content can be made to flow between multiple regions. The image is displayed in**

box 'A'. The 'article' is flowed from region '1', to region '2', to region '3' and finally to region '4'. Note how the content that is laid out in 'region 1' can be subject to a different style.

More Details

This illustrates some of the features of CSS Regions. First, the ability to associate a flow of content to a set of regions and effectively getting that content to be threaded from one region to the next.

Then, the example illustrates the concept of 'region styling', where the content that falls into a specific region is subject to additional style rule, in the same spirit as inline content falling in the 'first line' can be subject to additional styling as defined by the ::first-line pseudo element selector.

Finally, this initial example shows that the concept of region is orthogonal to the layout of regions. This means that regions can be created and positioned using existing CSS and HTML layout (such as multi-column, flex box or grid layout). The CSS Regions specification defines how these regions can be the recipients of a 'named flow'.

API

An interface named **`getNamedFlow`** and an event named **`regionLayoutUpdate`** are available for elements that are regions.

GETNAMEDFLOW

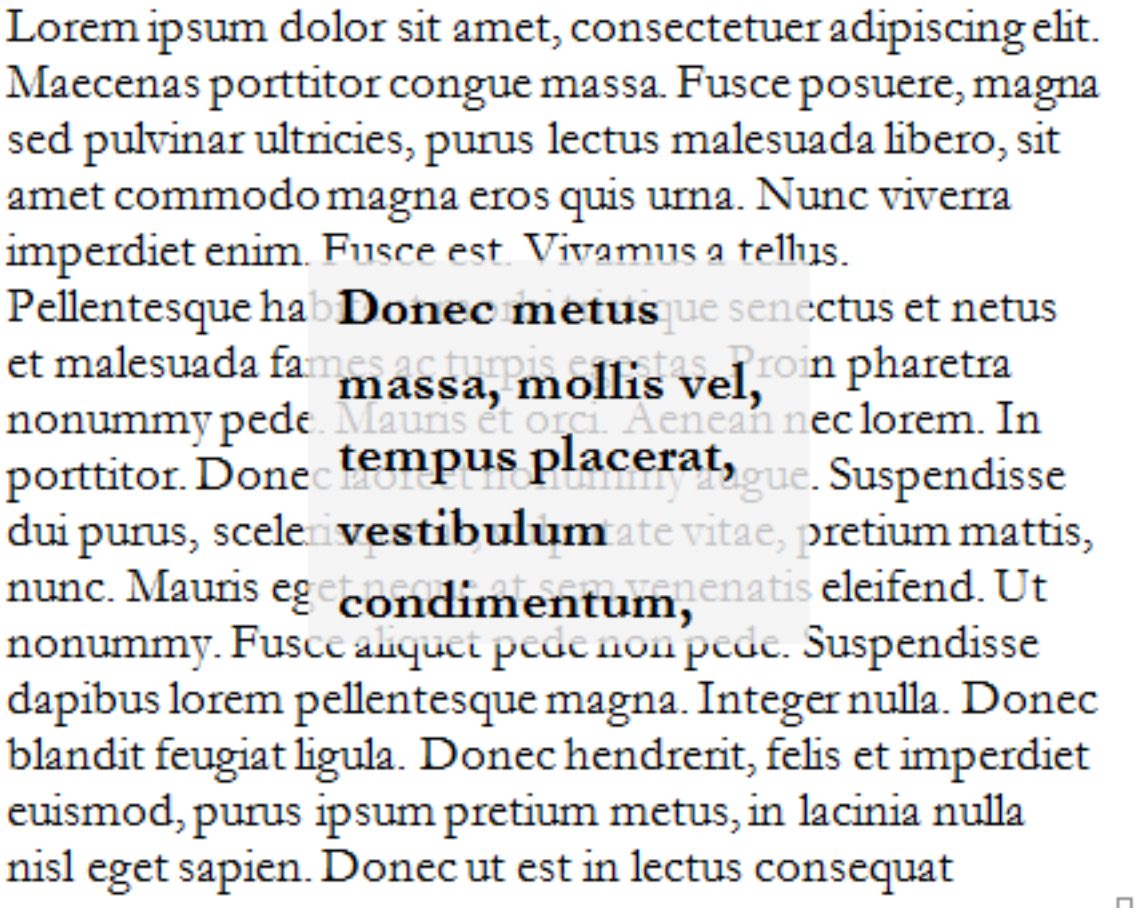
This returns the flow that that particular region is associated with. The properties available are:

- **`overflow`**
  A read-only boolean that tells you whether all of the content of the named flow fits within the regions that are part of the flow or whether it overflows.
- **`contentNodes`**
  A NodeList of all the content elements that belong to the flow.
- **`getRegionsByContentNode`**
  This returns all of the regions that a particular content element would flow through. A very long paragraph might flow through more than one region; with this method, you can retrieve all of the regions that that paragraph element flows through.
- **`regionLayoutUpdate`**
  This event gets triggered every time an update is made to the layout of a region. If the region's dimensions are altered, then the child content elements that are part of that region might alter, too (for example, a few might move to another region, or more child elements might become part of the region).

Exclusions

- Draft specification (a combination of two proposals: "Exclusions" and "Positioned Floats")
- Demo
- Browser support: IE 10+

Exclusions allow inline content to be wrapped around or within custom shapes using CSS properties. An element becomes an “exclusion element” when **`wrap-flow`** is set to a value that is not **`auto`**. It can then set the “wrapping area” for inline text outside or within it, according to various CSS properties. The **`wrap-flow`** can take the following values: **`left, right, maximum,both, clear`** or the default value of **`auto.`** Here is how each of these values would affect the inline content around the exclusion element:



wrap-flow: auto

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas porttitor congue massa. Fusce posuere, magna sed pulvinar ultricies, purus lectus malesuada libero, sit amet commodo magna eros quis urna. Nunc viverra imperdiet enim. Fusce est. Vivamus a tellus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Proin pharetra nonummy pede. Mauris et orci. Aenean nec lorem. In porttitor. Donec laoreet nonummy augue. Suspendisse dui purus, scelerisque at, vulputate vitae,

Donec metus massa, mollis vel, tempus placerat, vestibulum condimentum,

wrap-flow: right

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas porttitor congue massa. Fusce posuere, magna sed pulvinar ultricies, purus lectus malesuada libero, sit amet commodo magna eros quis urna. Nunc viverra imperdiet enim. Fusce est. Vivamus a tellus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Proin pharetra nonummy pede. Mauris et orci. Aenean nec lorem. In porttitor. Donec laoreet nonummy augue. Suspendisse dui purus, scelerisque at, vulputate vitae, pretium mattis, nunc. Mauris eget neque at sem venenatis eleifend. Ut nonummy. Fusce aliquet pede

Donec metus massa, mollis vel, tempus placerat, vestibulum condimentum,

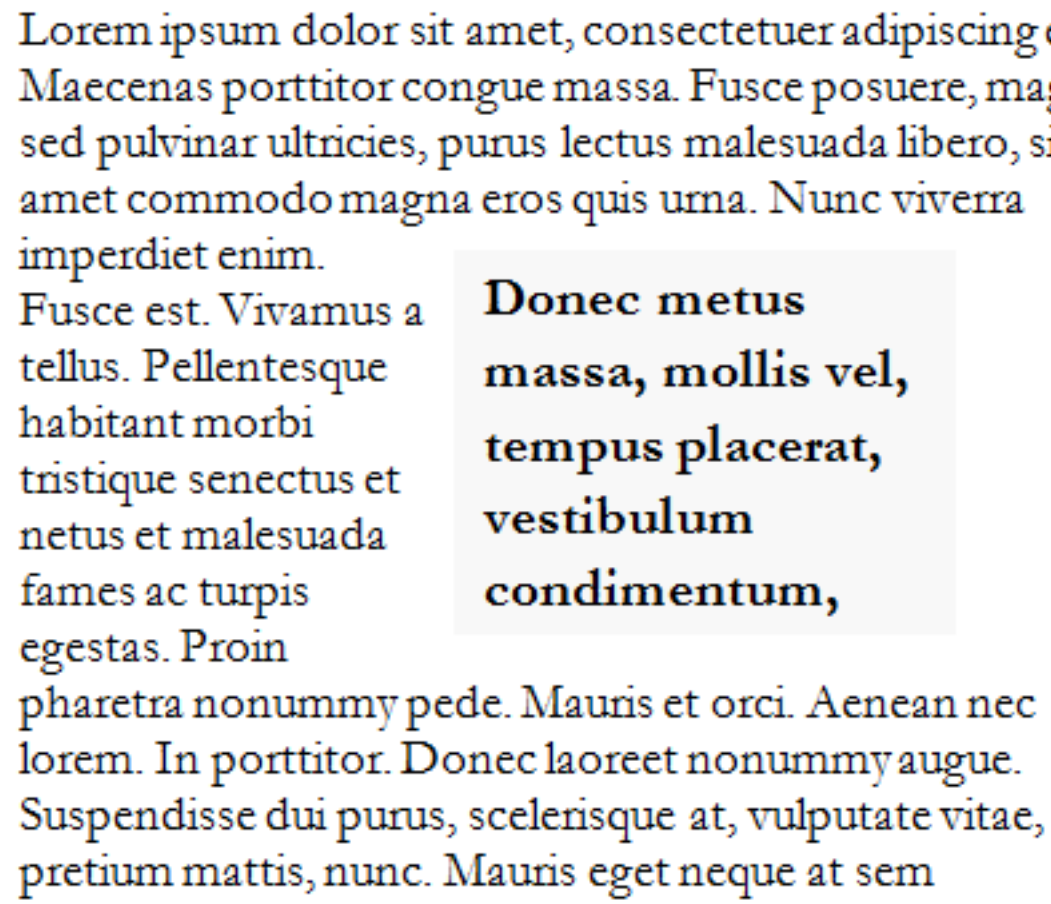
wrap-flow: both

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas porttitor congue massa. Fusce posuere, magna sed pulvinar ultricies, purus lectus malesuada libero, sit amet commodo magna eros quis urna. Nunc viverra

Donec metus massa, mollis vel, tempus placerat, vestibulum condimentum,

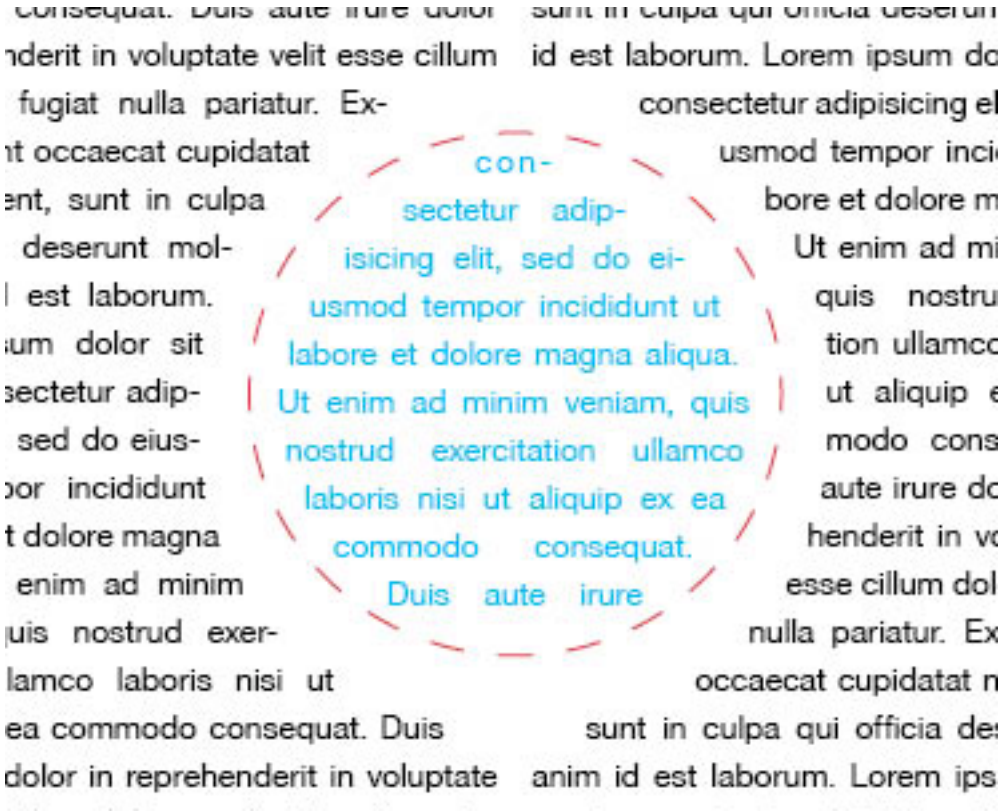
imperdiet enim. Fusce est. Vivamus a tellus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Proin pharetra nonummy pede. Mauris et orci. Aenean nec lorem. In

wrap-flow: clear



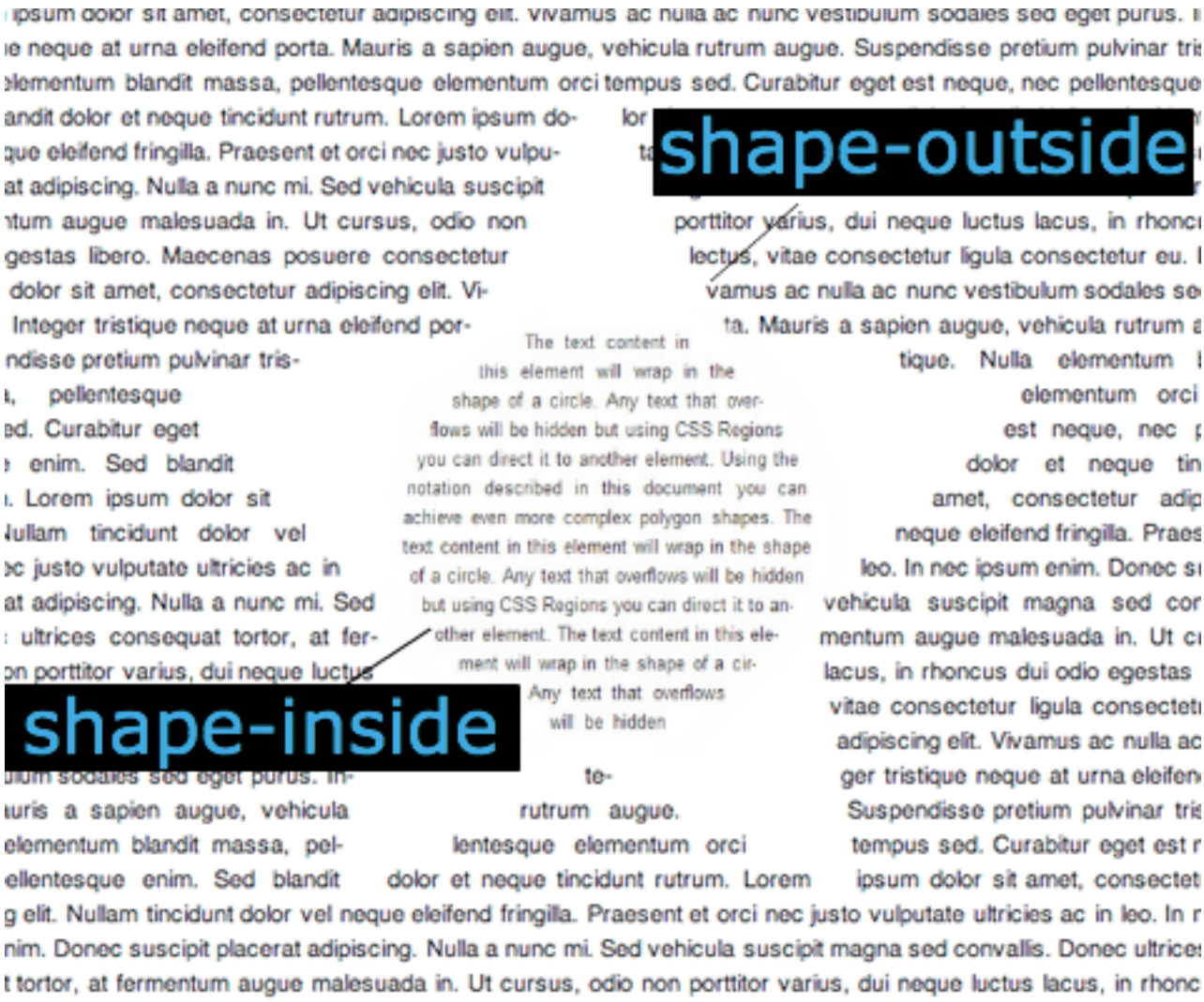
wrap-flow: maximum

The **`wrap-margin`** property can be used to offset the space between the boundary of the exclusion element and the inline text *outside* of it. The **`wrap-padding`** property is used to offset the space between the boundary of the exclusion element and the inline text *inside* it.



In the above image, the space between the content outside of the red dashed circular border and the black text outside of it is determined by the **`wrap-margin`**, while the space between the red dashed circular border and the blue text within it is determined by the **`wrap-padding`**.

Now comes the fun part: specifying custom shapes for the wrapping area. You can use two properties: **`shape-outside`** lets you set the wrapping area for inline text outside of the exclusion element, while **`shape-inside`** lets you set the wrapping area for inline text inside the exclusion element.



Both of these properties can take SVG-like syntax (`circle(50%, 50%, 100px);`) or image URLs to set the wrapping area.

Exclusions make magazine-like layouts on the Web a trivial matter and could spark the kind of creative use of content that we are used to seeing in print!

Grid

- W3C Editor's Draft
- Demo

- Browser support: IE 10+

Grid properties allow you to throw block-level elements into a grid cell, irrespective of the flow of content within the grid parent element. An element becomes a grid when **`display`** is set to **`grid`.** You can then set the number of columns and rows with the **`grid-columns`** and **`grid-rows`** properties, respectively. You can then declare each child selector itself as part of a grid cell, like so:

```
#title {
grid-column: 1; grid-row: 1;
}

#score {
grid-column: 2; grid-row: 1;
}
You can also use a template to plan the grid:
body {
  grid-template: "ta"
                 "sa"
                 "bb"
                 "cc";
}
```

In this syntax, each string refers to a row, and each character refers to a grid cell. In this case, the content of grid cell represented by the character `a` spans two rows but just one column, and the content represented by **`b`** spans two columns but just one row.

Now you can set any of the child element's **`grid-cell`** position:

```
#title {
grid-cell: 't';
}
```

This will make the element with the ID **`title`** within the body element to be positioned in the grid cell represented by the character **`t`** in the **`grid-template`** property.

If you are not using **`grid-template`**, you can also declare how many columns or rows a particular element should occupy with the **`grid-row-span`** and **`grid-column-span`** properties.

Flexbox

- W3C Editor's Draft
- Demo
- Browser support: WebKit Nightlies

Flexbox allows you to distribute child elements anywhere in the box (giving us the much-needed vertical centering), along with flexible units that let you control the fluidity of the child elements' dimensions.

Note that this specification has changed substantially since it was first proposed. Previously, you would invoke Flexbox for an element with **`display: box`**, but now you would use **`display: Flexbox`** to do so. Child elements can be vertically aligned to the center with **`flex-pack: center`** and horizontally aligned to the center with **`flex-align: center`**. Also note that all elements that obey the Flexbox need to be block-level elements.

How Do Various Properties Interact With Each Other?

You might wonder how to use these properties in combination. The following table shows which of these features can be combined.

	Paged Media	Multiple Columns	Regions	Exclusions	Grid	Flexbox
Paged Media	✓					
Multiple Columns	✓	✓	✓	✓		
Regions		✓	✓	✓		
Exclusions		✓	✓	✓		
Grid					✓	
Flexbox						✓

As you can see, the multiple-column properties can be used in conjunction with generated content for paged media, regions and exclusions. ~~But grid, Flexbox and regions are mutually exclusive (i.e. if an element is a grid, it cannot be a Flexbox or region).~~ Do note that, as Alan Stearns says in the comments, while a grid container cannot be a Flexbox or a region, a grid cell could become a region, or a Flexbox child item could be a region.

A NOTE BEFORE YOU RUSH OUT TO USE THEM IN CLIENT PROJECTS

The specifications are always changing, so be careful with them. Except for multiple columns, I would recommend using these strictly in personal projects and demos. The syntaxes and properties used in some of the demos are different from what you would find in the actual specifications, because they have changed since the builds that support a previous version

of the specification came out. Also, because they are still unstable, all of these properties are vendor-prefixed, which means you have to add support for each prefix as support is added.

If you do use these features, just make sure that the content is readable in browsers that do not support them. The easiest way to do this would be to use feature detection and then use CSS to make the content readable when the feature is unsupported.

Help The Working Group!

Do these layout proposals sound exciting to you? Jump on the www-style mailing list to provide feedback on them! Just note that the mailing list will flood your inbox, and you should carefully filter the emails so that you pay attention only to the drafts you are interested in.

Write demos and test how these work, and if you find bugs in the builds that have these properties, provide feedback to the browser vendors and submit bug reports. If you have suggestions for changing or adding properties to these proposals, do write in in the mailing list (or you can bug me on Twitter)!

These are exciting times, and within a few years the way we lay out Web pages will have changed dramatically! Perhaps this will finally sound the death knell of print. (Just kidding.)

About The Authors

Divya Manian

Divya Manian is a web opener for Opera Software in Seattle. She made the jump from developing device drivers for Motorola phones to designing websites and has not looked back since. She takes her duties as an Open Web vigilante seriously which has resulted in collaborative projects such as HTML5 Boilerplate.

Harry Roberts

Harry Roberts is a Senior UI Developer for BSkyB and type nerd from the UK. Enthusiastic, passionate and often outspoken, he is a writer, designer and member of Smashing Magazine's Experts Panel. He tweets at @csswizardry.

Inayaili de Leon

Inayaili de León is a London-based Portuguese web designer, specialist in cross-browser, semantic HTML and CSS, and clean, functional design. She writes frequently for well-known online and print publications and also on her own web design blog, Web Designer Notebook. In May 2011, she published a book, Pro CSS for High Traffic Websites, and she speaks frequently at local and international web conferences and meetups. She is currently working as a web designer on Canonical's Design team.

Jonathan Snook

Jonathan Snook writes about tips, tricks, and bookmarks on his blog at Snook.ca. He has also written for A List Apart, 24ways, and .net magazine, and has co-authored two books, The Art and Science of CSS and Accelerated DOM Scripting. Snook currently works on the design team at Shopify.

Louis Lazaris

Louis Lazaris is a freelance web developer based in Toronto, Canada. He blogs about front-end code on Impressive Webs and is a coauthor of HTML5 and CSS3 for the Real World, published by SitePoint. You can follow Louis on Twitter or contact him through his website.

Michael Martin

Michael Martin writes about Web design, WordPress and coding at Pro Blog Design. You can subscribe there for advice on making the most of your blog's design, or follow him on Twitter.

Niels Matthijs

Niels Matthijs spends his spare time combining his luxury life with the agonizing pressure of blogging under his Onderhond moniker. As a front-end developer he is raised at Internet Architects, investing plenty of time in making the web a more accessible and pleasant place.

Thierry Koblentz

Thierry is passionate about Web Design and CSS. He loves the challenge of solving problems and dreaming up original ideas that get him out of bed in the morning and keep him working late at night. He is a front-end engineer at Yahoo! and owns TJK Design and ez-css.org.

Zoe Mickley Gillenwater

Zoe Mickley Gillenwater is a freelance graphic and web designer, developer and consultant. She is the author of the book *Flexible Web Design: Creating Liquid and Elastic Layouts with CSS* and the video training title *Web Accessibility Principles* for lynda.com, and is working on the upcoming book *Stunning CSS3: A Project-based Guide to the Latest in CSS*. Zoe is currently a member of the Web Standards Project (WaSP) Adobe Task Force and was previously a moderator of the popular css-discuss mailing list. Find out more about Zoe on her blog and portfolio site or follow her on Twitter.