Beginning CSS Web Development

From Novice to Professional

■ ■ ■

Simon Collison

Apress®

Beginning CSS Web Development: From Novice to Professional

Copyright © 2006 by Simon Collison

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-689-0

ISBN-10 (pbk): 1-59059-689-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Chris Mills
Technical Reviewers: Richard Rutter and Dan Rubin
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editor: Ami Knox
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositor: Susan Glinert
Proofreader: Nancy Riddiough
Indexer: John Collin
Artist: Susan Glinert
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

CHAPTER 15

Tips, Tricks, and Troubles

Were there world enough and time, this chapter—like the others that preceded it—would remain focused around a particular topic. Unfortunately, there are 3.73 million topics related to CSS tips and tricks, and covering them all would demand a rather large and expensive book.

Therefore, this chapter is a bit of a smorgasbord, aiming to draw together the most common tips, tricks, and troubles. Over the next few pages, you'll find ideas for maximizing page space by manipulating overflowing content and learn to create incredibly simple CSS rollover images. Later, you'll enter the crazy world of hacks and filters, finding out how to serve separate style sheets to different browsers, before a final delve into common problems and quirks that you need to be aware of.

This chapter will cover

- Rollover images
- Overflow trickery
- Hacks and filters
- Troubleshooting

Rollover Images

A rollover image adds a simple bit of interaction for the user. Place the cursor over a rollover image, and another replaces it. For years, web designers preloaded on-state (i.e., on mouseover—when the mouse is rolled over the image) images to assist the browser in its rollover presentation. Preloading increases the weight of the initial download, but adds to usability by decreasing the wait time for an on-state image to appear. Basically, the browser would only start downloading the on-state image upon rollover, which was why preloading was so important.

In the Old Days

Before widespread adoption of CSS to achieve rollover images, this technique was done the hard way. Following is a typical example of the markup that was required for each and every rollover image. Please note that this chunk is *not* available as a code download!

```
<A HREF="target.html" onMouseOver="rollover('home')" onMouseOut="rollout('home')">
<IMG SRC="/images/home.gif" NAME="home" ALT="Home Page" BORDER="0" ➥
HEIGHT=130 WIDTH=115 ></A>
<SCRIPT TYPE="text/javascript">
<!--
setrollover("/images/home_over.gif");
//-->
</SCRIPT>
```

And this would be linked to a mammoth JavaScript file needed to perform the actual rollover function. Laborious and painful, there is thankfully a much better and easier way using CSS.

The (X)HTML

How is this for reduced markup? All that is required in the (X)HTML is the addition of a class to the `<a>` element.

```
<a href="target.html" class="rollover">Big Lee Hickman</a>
```

This unobtrusive additional markup allows you to take control of this particular link in any way you see fit using your style sheet.

The Image

Yes, image, not images. Just one image is required for this technique. To get around the preloading problem, both image states are combined as one image. This means that when the user rolls over the image, the on-state image is already available. The trick is to place the two versions side by side as one image, making very sure that each half is of an identical size (see Figure 15-1).

Figure 15-1. *The rollover image is a combination of the two image states. Only one half will be visible at any time.*

With this dual-purpose image prepared, we can now think about the CSS that will be used to slide the image backward and forward depending on the link state.

The CSS

First, the `rollover` class is declared. All that is required is to assign the `width` and `height` of the visible area of the dual image and set the `display` to `block` (to ensure the dimensions are respected). Note that the dimensions match the measurements in Figure 15-1. Also, the image itself is added using the `background` property, set to `no-repeat`.

```
/* Rollover class */
  .rollover {
    display:block;
    width:210px;
    height:157px;
    background:url(/images/rollover.jpg) no-repeat;
    text-indent:-9999px;
      }
```

Note also that the text is indented using a huge negative value (`text-indent:-9999px`). This ensures that the link text does not show, unless the style sheet is turned off or unavailable. In that situation, the user would not see the image, but would still have a clickable text link.

One gotcha to be aware of is that Firefox will currently outline links when they are clicked, placing a thin line around them. While not usually a problem for standard links, if a link has a huge negative value, this is going to look rather ugly, especially with links appearing to the right of your layout. Thankfully, there is a very useful bit of CSS that can be employed to turn such outlines off for all link states, by utilizing the `outline` property as follows:

```
/* Turn off all browser link outlines */
  a {
    outline:none;
    }
```

The next step is to make the magic happen. Remember that in Chapter 7 you learned about the various pseudo link states? Well, these will now be used to create the rollover effect. The three states `:link`, `:visited`, and `:active` are grouped, as they share the same values. For each of these states, the first half of the dual image will be visible.

```
  a.rollover:link, a.rollover:visited, a.rollover:active {
    background:url(/images/rollover.jpg) no-repeat;
    }
```

The most important step is to use the `:hover` state to reposition the dual image so that the second half (the rollover state) is in view. To do this, the image is repositioned 210 pixels to the left using a negative position value.

```
  a.rollover:hover {
    background-position:-210px 0;
    }
```

This simple method slides the first half of the dual image out of view, bringing the second half into the available space. This happens instantly upon rollover, creating a seamless, simple rollover effect (see Figure 15-2).

Figure 15-2. *On rollover, the second half of the dual image appears automatically.*

It should go without saying that this technique is invaluable for all kinds of interactive images, especially buttons and navigation items. Just make sure that the link still makes sense should the dual image be unavailable.

The Overflow Property

The `overflow` property defines the way that a child element is displayed when it exceeds its containing element. In other words, if there is too much content, the overflow value will dictate how or whether it should be displayed.

Overflow Values

There are four possible values for the `overflow` property, as detailed in Table 15-1.

Table 15-1. *Overflow Values*

Value	Description
`visible`	The content is not clipped. It renders outside the element.
`hidden`	The content is clipped, but the browser does not display a scrollbar to see the rest of the content.
`scroll`	The browser displays a scrollbar even if there is enough room to display the entire content.
`auto`	If there is too much content, the browser will display a scrollbar to see the remainder.

Let's look at two values in particular: `auto` and `hidden`.

overflow:auto

I love `overflow:auto` trickery (whereby specifying the height of a `<div>` and applying `overflow:auto` creates a mock `<iframe>` without all the accessibility headaches of `<iframe>`s). Take a look at the Style Company web site in Figure 15-3. For this job, all pages needed to be the same height all the way across, regardless of how much content was used.

Figure 15-3. *The Style Company web site (www.stylecompany.co.uk) uses overflow management to ensure all pages are the same height, regardless of content.*

This was achieved by using a container with `overflow:auto` applied and a set `height`, ensuring that no matter how much information appears in the container, the total height of the page does not increase. Let's look at creating our own example—a simple news stream using CSS.

The (X)HTML

There is nothing special about the markup. All that is needed is a simple container. Here, it is given a class of `stream` to denote that it carries all the streamed news items.

```
<div class="stream">
  <p>All the stuff you wanna scroll goes in here.</p>
</div>
```

Note that for this example, you will need to paste plenty of content into this container to see the effect of the overflow, such as masses of Lorem Ipsum text. With that taken care of, we can move on to the nifty styling.

The CSS

Now define the `stream` selector. First, the size of the box is declared (here it is a square 300×300 pixels), and some `padding` is declared to ensure the content does not touch the border. Most importantly, the `overflow` property is given the `auto` value.

```
.stream {
  width:300px;
  height:300px;
  padding:10px;
  border:1px solid #999;
  background-color:#FFF;
  overflow:auto;
  }
```

By giving the box a set `height`, you ensure that it does not expand or contract based on the content it holds. If there is more content than the box can hold, the scrollbar will automatically appear, allowing you to scroll through the whole text, as you see in Figure 15-4. Brilliant.

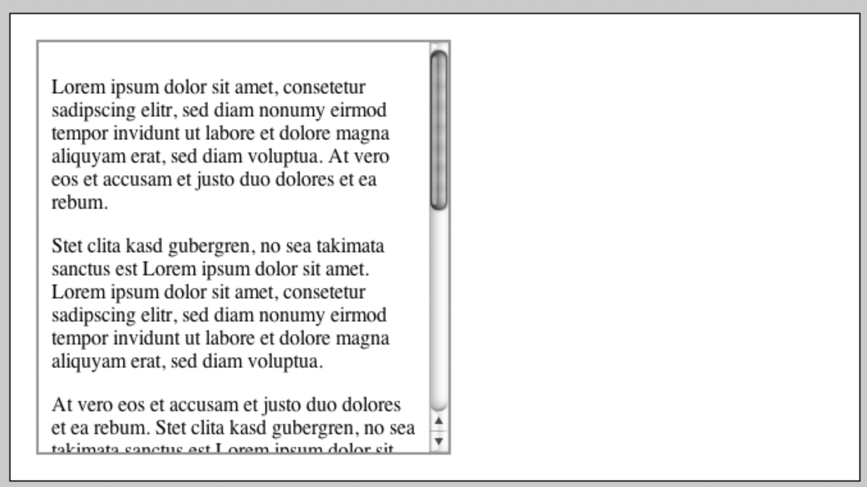

Figure 15-4. *Using overflow:auto to have a scrollbar appear when there is more content than the box can show*

The possibilities here are endless, but be wary of using multiple `overflow` boxes on your pages. A web page that features umpteen scrollbars can confuse users and look horrible! Still, this is a superb technique that shows great foresight from the creators of CSS.

overflow:hidden

There are occasions when you don't want the overflowing content to show, or you only wish it to appear in certain circumstances. This is where `overflow:hidden` comes in handy.

For example, you might have a beautiful photograph of some mountains that you wish to use as a banner image. This image is 796 pixels wide, and you know that if you add it to the page, it will always force the site to be at least that wide, and would force horizontal scrollbars if the window was decreased below that width. This is where `overflow:hidden` can come to your rescue. Let's do it!

The (X)HTML

For this example, I have an image that is 796 pixels in width by 320 pixels in height. The image needs to be placed into the (X)HTML page as follows:

```
<div id="masthead"><a href="#"><img src="images/mountains.jpg" ➥
alt="Mountain banner" width="796" height="320" /></a></div>
```

Note that the `width` and `height` of the image is declared to prevent the browser from waiting until after the image loads to render the rest of the layout (it will know how much space to use for the image and get on with implementing the CSS). This is of course totally optional.

The CSS

Here, the `container` div is also shown, so that you can see the context into which the banner will be placed. Following that, the CSS for the `masthead` ID is shown. Note that the `width` is a percentage value (`100%`) ensuring the `masthead` always fills the available space. The `height` is equal to that of the image it will hold. Finally, `overflow:hidden` is specified.

```
/* Container for all page content */
  #container {
    border:1px solid #000;
    padding:20px;
    background-color:#FFF;
    }
/* Masthead */
  #masthead {
    width:100%;
    height:320px;
    border:2px solid #999;
    background:#CCC;
    overflow:hidden;
    }
```

That is all you need to do. Figure 15-5 shows how the browser deals with the image at three different widths. Under normal circumstances, the thinnest window would force a horizontal scrollbar due to the larger width of the image. Thanks to hiding the overflow, this is skillfully avoided.

Figure 15-5. *Thanks to overflow:hidden, the containing element only reveals as much of the image as the window dictates.*

Combining Classes

It is possible to combine classes. This functionality provides real power when it comes to reusing elements. For example, you can use a containing element as often as you want, but you may not always want the elements it contains to be displayed in the same way. So, you could make several versions of the containing element and set the unique properties for each, but why would you want to repeat the `margin`, `padding`, and `background` styles for each and end up with more class names to worry about? This is where combined classes are really useful, as used on the Poptones web site (see Figure 15-6).

Figure 15-6. *The Poptones web site (www.poptones.co.uk) makes heavy use of combined classes to apply heading background colors that are in line with those of their containing boxes.*

Now, if you wanted an `<h3>` inside a particular `container` to have a red background, you might create the following styles:

```
h3 {
  font-size:110%;
  margin:10px 5px 10px 5px;
  padding:5px;
  }
.custom_background {
  background-color:#F00;
  }
```

You might then call that as follows:

```
<div class="container">
  <h2>Latest News</h2>
  <h3 class="custom_background">Man found in vase</h3>
    content
</div>
```

No! This is extra markup—forced to use a `class` attribute with each heading to define its color. Instead, create unique selectors for each section, keeping the existing `<h3>` declaration, and then creating a set of background styles with semantically meaningful names:

```
.news h3{
  background:#F00;
  }
.entertainment h3{
  background:#666;
  }
.sport h3{
  font-size:110%;
  background:#CCC;
  }
.music h3{
  background:#999;
  }
```

Then, all you need to decide is the purpose for each `container`. Let's say you add a `container` to house articles about sport. Add the `sport` attribute to the existing `container` class, separating the two class names with a whitespace character:

```
<div class="container sport">
  <h2>Sports News</h2>
  <h3>England win 26-0</h3>
    content
</div>
```

Suddenly, all instances of a `<h3>` heading inside this `container` take on the light gray background color. Exchange the word "sport" for "music" and now all headings in that `container` have a darker gray background. Remove the heading rule entirely, and headings have no background color at all.

Hacks and Filters

Hacks and filters are always lumped together it seems. Few talk of one without mentioning the other. However, the two are distinctly different.

Hacks use invalid CSS or parsing bugs—that is, methods of having the CSS work even though the syntax used is not correct (not opening and closing with a curly brace, for example). **Filters** on the other hand use valid CSS that the targeted browser just doesn't support.

CSS hacks and filters can help you selectively apply CSS (or not apply it as the case may be) to various elements. Always remember, though, that instead of using them any time you hit a snag, try finding a more standard cross-browser way to achieve the effect you're after first before resorting to them.

Let's look at hacks in more detail. You have already worked through a hack—The Box Model hack (Chapter 11). You now know that hacks use either naughty CSS or incorrect syntax to make their mark. Hacks exploit the broken bits of browsers—the bits that prevent a browser from being considered "modern" or "standards-compliant," for example. It is very important to consider that these browser manufacturers may or will eventually fix these errors, which might not only render your hacks useless, but also mean your style sheet malfunctions totally.

Safe Hacks

There are some hacks that can be considered as *safe*. Netscape 4 and IE5/Mac are not being developed any more. What we have now are, for better or (usually) worse, the definitive, final versions of these browsers. There will never be a next version that could make a mess of your pages, and therefore it could be considered OK to let these hacks live in your CSS for all time. Let's look at a couple of example hacks.

@import

Early browsers are notorious for malfunctioning when presented with CSS rules they don't understand (Netscape 4 will crash at the sight of certain rules). The @import hack allows you to hide entire style sheets from Netscape 4 and older browsers because they don't understand it—as you learned way back in Chapter 1, the `@import` rule links to an external style sheet from within another style sheet (be it external or in a `<style>` element within the `<head>` of the (X)HTML document). Interestingly, early browsers do not understand this approach and simply ignore it and therefore also the style sheet it refers to.

Using @import

Here is a simple guide to using the @import hack. First, you will begin with two style sheets:

- `basic.css` (only simple rules for early browsers)
- `modern.css` (advanced CSS2, rules to override rules in `basic.css`)

Next, create a third style sheet called `import.css` that contains only the following:

```
@import "modern.css";
```

Then, link the `basic.css` and `import.css` in the `<head>` of the (X)HTML document.

```
<link rel="stylesheet" type="text/css" href="basic.css" />
<link rel="stylesheet" type="text/css" href="import.css" />
```

Note that the `basic.css` style sheet must be linked first.

The Effect

Here's what happens. All older browsers (compliant with the original CSS1 specification) will load `basic.css` and `import.css`. However, only modern browsers will understand the `@import` rule and therefore only they will see `modern.css`. As `modern.css` is linked after `basic.css`, its rules will override those in the latter style sheet.

Caio Hack

The Caio hack is named after a bloke called Caio Chassot who invented it. This hack is used to hide certain rules from Netscape 4 (rather than the whole style sheet as with the @import hack).

How Do We Say "Caio"?

This hack makes use of a comment like this to begin hiding from Netscape 4:

```
/*/*/
```

This is almost identical to a correct comment, the same as those discussed in Chapter 1; however, there is a forward slash inserted where spaces and words would normally be placed. The poor old browser thinks the comment didn't close, and will therefore ignore all following CSS. To stop the ignorance, just add another standard comment, and Netscape 4 will jump back to life and start reading your CSS again.

Example Caio

Take a look at the following CSS. Note the use of traditional comments within the list of rules, and the single use of the Caio hack (comments highlighted in bold).

```
body {
  color:#000;
  background:#FFF;
 }

/* Netscape 4 is looking the other way */
/*/*/
  body {
    background-image:url(images/we_hate_netscape_banner.jpg);
    }

/* Bring Netscape 4 back into the room */
  ...remaining CSS follows
```

With this example, all browsers (including Netscape 4) will see the first `body` selector and render black text on a white background. After the hack, however, a background image is specified that shouts "We Hate Netscape!" As this comes after the Caio hack and before the comment that brings Netscape's focus back, this offensive image will not show on that browser. As Netscape 4 is not being developed further, you can use this hack safe in the knowledge that no sudden updates will expose your hatred.

Hide Style Sheets from All but IE/Mac

This super-useful hack is a method of serving style sheets to IE5/Mac only. It validates, and also appears to work for, IE 4.5/Mac, although that version is now extremely rare.

The IE/Mac sheet is linked as follows.

```
<style type="text/css">
  @import("ie-mac.css");
</style>
```

Be very careful not to leave a space between `@import` and `("`, otherwise the hack will not work as intended. IE/Mac was once the most standards-compliant browser in existence, but soon its shortcomings with CSS became obvious as better browsers came into being. By serving IE5/Mac-specific styles, dealing with this browser can be a little less problematic. Also, the fact that Microsoft will never, ever upgrade it means that this hack is perfectly safe. Ideally, all Mac users would be browsing with Safari, Camino, or Firefox, but we can't take this for granted (yet).

IE7 Is Coming

Looming over the horizon, like a potentially troubling but nonetheless welcome new friend, is Internet Explorer 7, expected to ship—annoyingly—around about the same time as this book. As such, this author hasn't explored it fully, but so far so good, in that many of IE6's failings have been corrected, and a number of those trusty old hacks will no longer be necessary. As the public beta shows, most of the really annoying CSS bugs from IE6 are now fixed. What's more, there is finally support for a range of CSS2 (and even CSS3) selectors. Let's have a street party! Well, after we've updated all our web sites to make sure they still work in IE7, given that the IE hacks we've used now break our sites in IE7!

The IE7 team have dropped support for the Star HTML hack, which targets problems in IE5 and IE6 that are pretty much removed from IE7. I heartily recommend you read Dave Shea's article "Stop Hacking, Or Be Stopped" (`www.thinkvitamin.com/features/css/stop-css-hacking`) if you demand more enquiry into IE7 and what it means for CSS hacks.

There are one or two hacks and filters that may be worth readying for IE7, however, especially during the initial evaluation period. Let's look at a couple.

Conditional Commenting

Your old hacks won't work—mostly. The IE team has publicly moaned about hacks—kind of like kids who made a cool exhibit at the science fair, only to have some bullies come along and tell them that while it's great that they invented a money machine, it's too bad that it's held together with duct tape. The IE boys are in favor of an IE-proprietary method known as **conditional comments**. Let's look at what this means.

Out with the Star HTML Hack

Since IE5/Win, web designers have used code like this to filter a rule to IE/Win:

```
* html #selector {
  width:100px;
  }
```

This is known as the Star HTML hack, again exploiting a parsing bug to send different values only to IE5 and upwards. As it is a hack, the best thing to do is avoid it.

The New Way

We should instead be using valid conditional comment syntax to serve up a link to an external CSS file that will contain the IE-specific CSS. Here's an example of the conditional comment that would be placed in the (X)HTML document's `<head>`, above or below the link to the main style sheet used for other browsers.

```
<!--[if IE]>
  <link rel="stylesheet" href="ie-specific.css" />
<![endif]-->
```

And then in the IE-specific CSS file, the rule that was once the essence of the hack can now be placed neatly in the style sheet.

```
#selector {
  width:100px;
  }
```

This is much better. Conditional comments are authorized, and the IE team won't ever send the FBI to your house for using them. They validate, and even though they're proprietary, when used with discretion, they allow us to accomplish exactly the same thing as CSS hacks.

A Hack Specifically for IE7? If You Must...

At the time of writing, IE7-specific hacks are materializing, such as the Triple-X hack. A targeted CSS filter for IE7, this hack applies or excludes CSS specifically from the newcomer. I'm not one to condone hacks, but then again I never use a ratchet, yet I have one in the tool shed just in case. If you need to know more, go forth to Brothercake's site (`www.brothercake.com/site/resources/reference/xxx/`) for further instruction.

Troubleshooting

Here is a section that could easily have spanned 426 pages. Every day, vocal CSS users publish more and more findings about quirks, problems, and bugs. Likewise, each designer has his or her own recommendations for more productive CSS.

This is a list of perhaps the most common problems those new to CSS might encounter. If these tips don't help you, don't be downhearted. The CSS community is big, sharing, and search-engine optimized, so get with the Google for a quick answer to your problem if it isn't covered here.

Common Problems

I recommend reading through these problems even if you are currently content. It pays to be aware of the kind of issues that may well crop up, some of which apply to more general CSS use too. Let's begin with a fairly detailed overview of one of the most annoying issues—margin collapsing.

Margin Collapsing

If we're on the subject of mysterious quirks, it makes sense to start with margin collapsing, which will hurt you at some point. The problem can manifest itself in one of two ways when two elements meet: either extra whitespace appears that just can't be removed or you suddenly are unable to add whitespace using margin values. Believe it or not, this is actually supposed to happen!

Example One

In the example shown in Figure 15-7, both paragraph elements have been given a 20-pixel margin. However, because the bottom margin of the first paragraph touches the top margin of the second paragraph, the margins collapse, making the space between both paragraphs 20 pixels instead of 40.

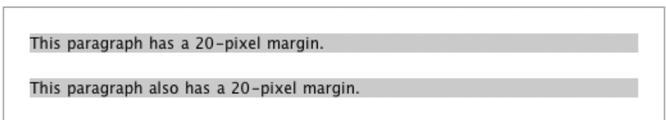

Figure 15-7. *Margin collapse sees only one of the 20-pixel margins having effect.*

Example Two

This doesn't just happen when one block-level element follows another; it also happens when the second element (the light gray paragraph) is the child of the first (the darker gray container) as in Figure 15-8. The top and bottom margins of the paragraph collapse into the margins of the div, leaving just a 20-pixel vertical margin around both elements. Crumbs.

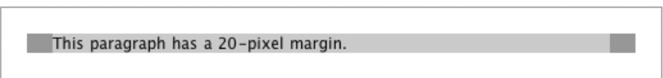

Figure 15-8. *Even with child elements, one of the specified margins has no effect.*

The Solution

There are a number of ways to get around margin collapsing. The easiest is to add a border or `1px` of padding around the elements so that the margins no longer touch and therefore no longer collapse (see Figure 15-9).

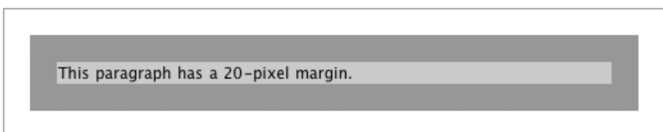

Figure 15-9. *Adding a border or padding value separates the margins and prevents the collapse.*

If relevant, you can also stop margins collapsing by changing the `position` property of the element. Margins of absolutely and relatively positioned boxes don't collapse. Also, if you `float` an element, its margins should no longer collapse. This may or may not be possible, of course.

Perfect Percentage Values Still Break Your Layout

As if life weren't difficult enough as it is, sometimes 50% + 50% does not add up to 100%—not with browsers anyway. The actual value could be 100.1%, which will force some layouts to break in some browsers. To find out if this is happening to you, try changing 50% to 49%, and then retest your layout. Always worth a try.

Flash of Unstyled Content

If for some reason you rely on `@import` alone to import your style sheet(s), you may notice that Internet Explorer will show unformatted (X)HTML for a second or two before applying the CSS. This can be avoided by adding just one `<link>` element or `<script>` element inside the document's `<head>`. For further investigation, take a look at "Flash of Unstyled Content (FOUC)"over at the Blue Robot site (`www.bluerobot.com/web/css/fouc.asp`).

The Selector Has a Matching Element, but Nothing Happens

This is all down to case sensitivity. To cut a short story shorter, if you have a CSS selector called `leftColumn` (note the uppercase `C`), the value in the (X)HTML must also use identical case. This problem typically occurs with Mozilla-based browsers.

The Background Image Will Not Display

The image is on the server, but it will not show as a background image. You load it via its direct URL in the browser, and there it is, so why not via CSS? Chances are that you have specified the image with quotation marks (`background:url("image.gif");`), which is wrong, but oh so easy to do. Remove the quotations, and the image should appear. The main culprit here is IE5/Mac, as most other browsers will spot your stupidity and work around it.

Recommendations

Here follows several of this author's own favorite tips for fitter, happier, more productive CSS management. None of these are rules or regulations, but they have certainly helped me.

Use a Modern, Standards-Compliant Browser

This will ensure your (X)HTML and CSS will be more on the button from the start, and you won't have to hack as much to support other browsers. Ideally, you should use a Mozilla browser (such as Firefox), Safari, or Opera—and not IE. Working within an outdated browser can have you relying on its inaccurate rendering. Always start with what you know is right and then amend if necessary for elderly browsers. This will also ensure that the whole experience is infinitely more pleasurable.

Avoid Reliance on the Box Model Hack

Earlier in this chapter you learned of the care needed when using hacks of any kind. Remember that as current browsers are updated and bugs are fixed, your hacks might come back to haunt you. You can avoid reliance on the Box Model hack (see Chapter 11 for a recap) by applying padding to the parent element instead of the child that has a fixed width specified.

Track Down Errors by Commenting Out Areas of the Style Sheet

This is a top, top tip, especially useful when your familiarity with a style sheet has waned, or you are working with one created by another person. Often you will find an error in the rendered design, or need to track down a particular problem. The best initial approach is to comment out half of the style sheet and reload the page. If the problem persists, remove the commenting and apply it to the other half. Once you know which half the problem lives in, you can use similar mass commenting to delve in further until you find the culprit. This method is particularly helpful when trying to understand a particular inheritance woe.

Avoid Aggressive Style Sheet Caching During Development

If (like this author's ISP) your Internet service provider caches your style sheets on site, you are going to have a hard time seeing all those little tweaks and changes—until probably Thursday, when they flush it out.

To get around this problem, forget external style sheets while developing. Embed your CSS in the `<head>` of your test page, forcing it to reload each time you view the page. When it is time to launch, or create further pages, then you can move the CSS from the document into its own external style sheet. This approach isn't always possible, but that is why recommendations are not laws.

Stopped Using a Style? Remove It

I do it. My grandmother does it, and so will you. During development, you will try out styles, use them for a while, and then move on, doing things another way. That is fine, but you end up with lots of unused styles littering your style sheet.

This isn't a major problem, although any unwanted code adds to overall page weight. The main issue is how these unwanted styles might affect those in use. Knowing what you learned about inheritance and the cascade in Chapter 2, you'll be aware that these harmless extraneous styles might be influencing others, causing the oddities on your rendered page. Play safe and either comment them out or remove them altogether.

Validate Your Code

Your CSS looks right. You know it's right, so why isn't it doing what you think it should? Obviously there could be several negative factors, but often it is nothing more than badly formed (X)HTML.

Validated code isn't necessarily accessible code, but it's at least well formed. Run your (X)HTML through the validator to find any unclosed elements or other oversights. Think of your CSS as being like a really expensive Prada coat. It looks much better on Kate Moss than on a bicycle.

To Conclude...

This chapter has hopefully . . . hey, hang on! This is the penultimate chapter. Quickly, put the kettle on and break out the Christmas biscuits—you are a CSS professional!

Over the last 15 chapters, you've blossomed from a mere seed planted in slightly damp cotton wool into a glorious tree full of early springtime flowers. Sure, you have plenty more growing to do, but already people are impressed by your development. Is that a squirrel on your shoulder?

The final chapter of this book is the glorious *Dead Goods* case study—just over the page. This case study will tie together much of what we have been through, drawing examples from each chapter to produce a living, breathing, CSS-based web site. The proof of the pudding is in its standards compliance, as they say.