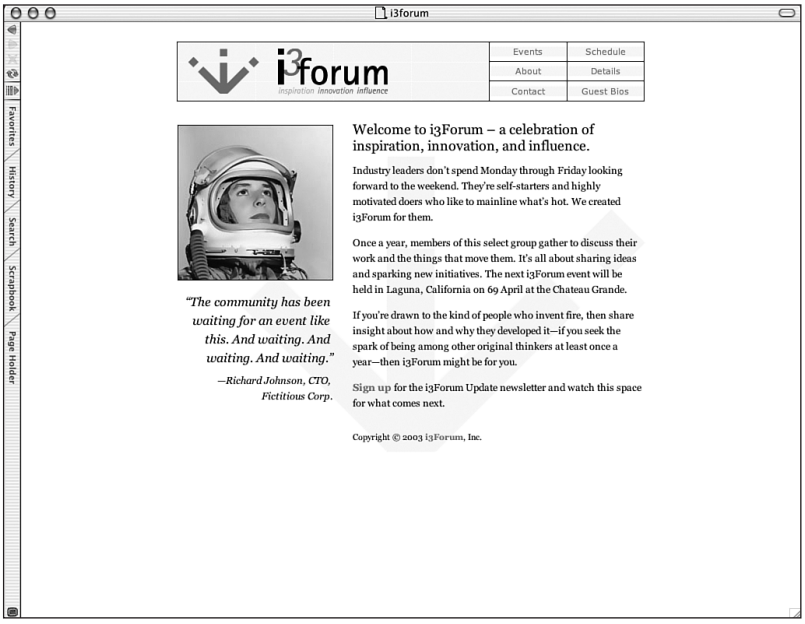
chapter ten

CSS in Action: A Hybrid Layout (Part II)

In Chapter 8, "XHTML by Example: A Hybrid Layout (Part I)," we created hybrid markup for the i3Forum site, combining structural elements like `<h1>`, `<h2>`, and `<p>` with nonstructural components (XHTML tables used to lay out the basic grid), and we used table summaries, `accesskey`, and Skip Navigation to make the site more accessible in nontraditional browsing environments.

In this chapter, we'll complete our production task by using CSS to achieve design effects that support the brand and make the site more attractive without relying on GIF text, JavaScript rollovers, spacer pixel GIF images, deeply nested table cell constructions, or other staples of old-school web design.

Figure **10.1** shows the home page template as it appears after our first pass at writing a style sheet. As with all design, using CSS is an iterative process. In this chapter, we'll complete our CSS in two passes. The first pass handles 90% of what's needed; the second fixes errors and adds finishing touches.



10.1
The template as it appears after our first pass at CSS. Elements, sizes, fonts, and colors are in place, but backgrounds don't quite fill the right-side menu "buttons." A bit more work is needed.

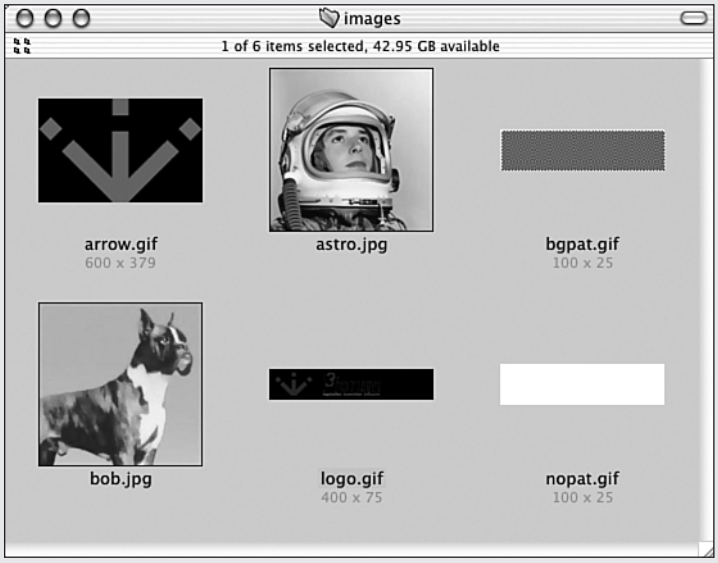
Preparing Images

Although the site was designed in Photoshop, it's not your typical slice and dice job. Figure **10.2** shows the six images used to create the entire site. Three are for foreground use: the astronaut photo, the "best of breed" dog photo, and the transparent logo GIF used at the top-left corner of the menu bar.

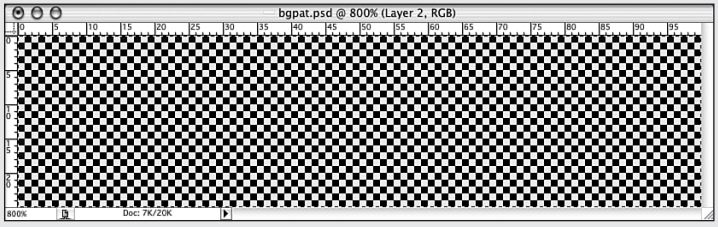
The remaining three images are backgrounds. Arrow.gif is a screened image derived from the logo that will be placed in the background as a watermark. Bgpat.gif [**10.3**], consisting of single-colored pixels alternating with single transparent pixels, will be used to create background color effects in the menu bar. Nopat.gif, a plain-white background, will replace bgpat.gif in CSS rollover effects and might also be used to indicate the visitor's position within the site's hierarchy via an embedded style added to each page at the end of the project (see the following sidebar, "The Needless Image").

The Needless Image

Strictly speaking, one of our six images is not needed in this execution. Nopat.gif, the plain-white background image [**10.2**], is superfluous. A CSS rule specifying `background: white` would have the same effect (and later in this chapter we'll use a rule like that instead of nopat.gif).



10.2
Just six images were used in this site, three of them as backgrounds. Photoshop's/ ImageReady's "slice and dice" features are not needed.



10.3
The alternating pixel background GIF image enlarged 800% and with a black background, inserted for this book's purposes, standing in for the transparent pixels.

For our nav bar, however, we've gone ahead and used this background anyway, so you can see how a CSS background image swap is achieved. In an execution involving swapped watermarked or textured backgrounds, you need two images; therefore, you would need to write the kind of CSS rules shown in this part of the chapter.

Establishing Basic Parameters

Our images are in place, and with our original comp to guide us, we can begin using CSS to establish basic design parameters. We know that the site will be white with a black background, that its text will be set in several sizes of Georgia (or an alternative serif), that a screened watermark based on the logo will hug the bottom of the content area, and that certain whitespace values must be enforced without using spacer GIF images or additional table junk. Let's make it happen.

Overall Styles, More About Shorthand and Margins

In our first rule, we establish basic page colors and upper and lower margins:

```
body  {
      color: #000;
      background: #fff;
      margin: 25px 0;
      padding: 0;
      }
```

Per this rule, all text will be black (`#000`) on a white (`#fff`) background. The colors are described in CSS shorthand, as explained in Chapter 9, "CSS Basics." (`#000` is shorthand for `#000000`; `#fff` is the same as `#ffffff`.) Shorthand can only be used to replace paired characters; `#fc0` is the same as `#ffcc00`. Shorthand cannot be used in the absence of paired characters. There is no shorthand for a nonwebsafe color like `#f93C7a`, for example.

Shorthand and Clock Faces

The first rule also establishes a top and bottom margin of 25px and left and right margins of `0`. It is a shorthand version of this:

```
margin: 25px 0 25px 0;
```

The preceding, in turn, is a shorthand version of this:

```
margin-top: 25px;
margin-right: 0;
margin-bottom: 25px;
margin-left: 0;
```

In CSS, values are assigned in the order of the main numbers on a clock: 12 o'clock (the top margin), 3 o'clock (the right margin), 6 o'clock (the bottom margin), and 9 o'clock (the left margin). If we wanted our page to have a top

margin of 25px, a right margin of 5px, a bottom margin of 10px, and a left margin of 30% of the entire width, our rule would read like this:

```
margin: 25px 5px 10px 30%;
```

When the vertical margins are the same at the top and bottom (as they are in this site—namely, 25px) and when the horizontal margins are the same at left and right (as they are in this site—namely 0), we can save a few bytes of user bandwidth by typing this:

```
margin: 25px 0;
```

As mentioned in Chapter 9, the 0 value does not require a unit of measurement. 0px is the same as 0cm, 0in, or 0bazillionmiles. (There is no "bazillion miles" unit of measurement in CSS, but if there were, we wouldn't need to use it when the value is zero.)

Finally, our rule sets padding to 0 to accommodate Opera, which uses padding rather than margins to enforce page gutters.

Hide and Block

In our next step, with two simple rules, we accomplish several useful things:

```
.hide  {
       display: none;
       }

img {
       display: block;
       border: 0;
       }
```

The first rule creates a class called hide that can be used to make elements or objects invisible in CSS-capable browsers. As explained in Chapter 8, we're using this CSS feature to hide our Skip Navigation link in modern browsers while making it readily available to users of text browsers, screen readers, and PDA- and phone-based browsers that do not support CSS. (As of this writing, some PDA-based browsers partially support CSS, but they do about as poor a job as 3.0 and 4.0 desktop browsers did. Hopefully, many of these implementations will have improved by the time you read this book.)

The Image Rule

The second rule is more useful and less problematic. First, `display: block;` states that every image on the page will be rendered as a block-level element instead of inline. If you're unfamiliar with these terms, here are two easy examples. Paragraphs are block-level elements; the deprecated `<i>` (italic) tag is an inline element. Block-level elements exist in their own "box" and are followed by an implied carriage return. Inline elements are part of the flow, with no carriage return and no clear-cut "box."

By telling the browser to treat images as block-level elements, we avoid having to write `<br>` or `<br clear="all">` or similar junk before and after our images, and we also avoid having to stick those images in their own table cells to preserve our layout's spacing requirements. (You'll learn more about that in Chapter 11, "Working with Browsers Part I: DOCTYPE Switching and Standards Mode." If you can't wait, you might also read "Images, Tables, and Mysterious Gaps" at `http://devedge.netscape.com/viewsource/2002/img-table/`.)

This seems so simple that many of you will skip over it without thinking about it, but explicitly assigning `block` or `inline` status to an element is an incredibly powerful tool. Ordinary links, by being made block-level elements via CSS, can turn into buttons, for example. In a later rule, using an additional selector, we will be able to add specific vertical whitespace to images that live in a particular part of the layout, thus achieving with a few lines of CSS what would otherwise require a markup mess of slicing, dicing, table cell nesting, and spacer GIF images.

Read it again. We're telling you how to achieve layouts that look like 50 table cells and a dozen sliced images but are just a few lines of markup and a few CSS rules. Okay. Point made.

Next, the `border: 0;` declaration turns off image borders, so we don't have to write `border="0"` in our markup. (If we care about the way the site looks in non-CSS browsers, we *will* have to write `border="0"` in our markup anyway. We did so, and explained why we did so, back in Chapter 8.)

Coloring the Links (Introducing Pseudo-Classes)

In presentational HTML, we controlled link colors via attributes to the `body` element such as `vlink="#CC3300"`. In modern web design, we can leave our `body` naked and unadorned and use CSS instead. To sweeten the deal, CSS adds

a "hover" state to the familiar link, visited, and active states we learned to love in the 1990s. CSS also allows us to do more than merely change link colors.

CSS calls these anchor (link) states *pseudo-class selectors*. In the CSS way of thinking, a "real" class is one that you specify explicitly with a `class=` attribute. A pseudo-class is one that depends on user activity or browser state (`:hover`, `:visited`). There are also pseudo-elements (`:before` and `:after`). In any case, with the four rules that follow, we control link colors and more [**10.4**, **10.5**].

```
a:link {
       font-weight : bold;
       text-decoration : none;
       color: #c30;
       background: transparent;
       }
a:visited {
       font-weight : bold;
       text-decoration : none;
       color: #c30;
       background: transparent;
       }
a:hover {
       font-weight : bold;
       text-decoration : underline;
       color: #f60;
       background: transparent;
       }
a:active {
       font-weight : bold;
       text-decoration : none;
       color: #f90;
       background: transparent;
       }
```

10.4
The link color is dark red, bold, and without an underline, per the CSS a:link rule.

10.5
When the visitor's mouse hovers over the link, its text color "lights up" in orange, and an underline appears, per the CSS a:hover rule.

More About Links and Pseudo-Class Selectors

In the previous four rules, the only thing that might be new to you is the text-decoration property. When its value is `none`, there is no underline. When its value is `underline`—you guessed it—the link is underlined, just like all links were back in the mid-1990s. When its value is `overline`, a line appears *over* the text instead of under it. You can combine overline and underline like so:

```
text-decoration: underline overline;
```

What would that look like? It would look like the linked text was in a box with a top and bottom but no sides. We know two web designers—one of them being us—who have used that effect at least once. Two additional points are worth noting before we leave the land of lovely links.

Use LVHA or Be SOL

Mark you this, oh brothers and sisters: Some browsers will ignore one or more anchor element pseudo-class rules unless they are listed in the order shown earlier, namely link, visited, hover, active (LVHA). Change that order at your peril. A popular mnemonic to remember the order is "LoVe—HA!" (There are some bitter people out there.) If you want to understand why the order matters, `http://www.meyerweb.com/eric/css/link-specificity.html` explains it in some detail.

Pseudo-Shenanigans in IE/Windows

Note that even in its latest, greatest incarnation (at least as of this writing) Internet Explorer for Windows has trouble with the hover and active pseudo-classes. Hover states tend to get stuck. Use your Back button in IE/Windows, and you will very likely find that the last link you moused over is still in its hover state. For that matter, you will very likely find that the link you clicked to move forward is still in its active state. You will probably not like this one bit.

Because the active color also gets stuck, if you apply background images to the `a:active` pseudo-class, IE/Windows will get them wrong. If your audience includes IE/Windows users (and whose audience does not?) you might decide to avoid `a:active` altogether. Or you might choose, as we have, not to do anything especially creative or challenging with it.

Whether this freezing of link states is a bug or a useful feature depends on whom you ask. Two hundred million IE/Windows users are probably accustomed to it by now, and many might believe the web is supposed to work this way.

Sketching in Other Common Elements

In the code block that follows, we see our good friend, the "Be Kind to Netscape 4" rule (Chapter 9) being used to tell the browser that the entire site should use Georgia or an alternative serif face [**10.6**].

```
p, td, li, ul, ol, h1, h2, h3, h4, h5, h6   {
             font-family: Georgia, "New Century Schoolbook",
Times, serif;
             }
```

10.6
Fonts are specified by a single rule applied to multiple selectors (`p`, `td`, `li`, `ul`, `ol`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`). Sizes and white-space are controlled by means of additional rules and selectors.

Georgia, a Microsoft screen font designed by Type Directors Club (TDC) Medal winner Matthew Carter to be legible even at small sizes, is found on nearly all Windows and Macintosh systems. New Century Schoolbook is found on most UNIX systems; Times has been found on Paleolithic computing systems; and when all else fails, there's our other good friend, the generic serif.

In the following rule, we let the browser know that headlines should be slightly larger than the user's default font size. When no base font size has been specified, the browser will consider the user's default font size to be 1em. (It doesn't matter if the user's default font size is 12px or 48px. It's still 1em.) To make the headline a bit larger than the user's default, we'll set it at 1.15em. We'll also insist that the headline be of normal (not bold) weight:

```
h1    {
      font-size: 1.15em;
      font-weight: normal;
      }
```

There is no need to tell the browser that h1 should be set in Georgia. We did that in the previous rule.

Now we use the `html` selector to add more detail to our `p` style. All elements on the page except `html` itself are children of `html`. We could as easily have written `p` instead of `html p`, but we wanted you to feel your money was well spent on this book. Here is the rule, followed by explanations:

```
html p    {
          margin-top: 0;
          margin-bottom: 1em;
          text-align: left;
          font-size:  0.85em;
          line-height: 1.5;
          }
```

In the preceding rule, we establish that paragraphs have no whitespace at the top (thus enabling them to snugly hug the bottoms of headlines and subheads), 1em of whitespace at their bottoms (thus preventing them from bumping into each other), and are somewhat smaller (0.85em) than the user's default font size (using the same reasoning applied to the `h1` earlier, but in the opposite direction).

More About Font Sizes

It is tricky to specify relative font sizes as we've done in the previous rule. Specifically, it is tricky to specify that fonts should be slightly smaller than the user's default because the user might have set a small default. If he *has* set a small default, your small text might be too small for his liking.

For example, if the user has specified 11px Verdana as his default font, your small text might be 9px or 10px tall—hardly a comfortable size for reading long passages of text. The user so afflicted can easily adjust the layout by using his browser's font size widget, but some users might find it annoying to do so, and a few might not know they can resize text.

In most systems as they come from the factory, default font sizes are as huge as the least attractive part of a horse, and a size like 0.85em should look darn good. If the user is visually impaired and has set her size much larger than the default, the font will still look good to her, and she will see that it is slightly smaller than normal. But if a Windows user chooses "small" as his default browsing size, or if a Mac user sets her browser to 12px/72ppi, our text might look too small, causing the user to weep piteously or (more likely) exit the site in frustration and haste.

Alternatively, we might have chosen a pixel value for our text size:

```
font-size: 13px;
```

We could have done this and created the leading as well using CSS shorthand:

```
font: 13px/1.5 Georgia,"New Century Schoolbook", Times, serif;
```

Unlike relative sizes based on em, pixel-based sizes are 99.9% dependable across all browsers and platforms. And if a pixel-based size is too small for a given user, he or she can adjust it via Text Zoom or Page Zoom in every modern browser on earth but one. Unfortunately that one browser is IE/Windows, currently the web's most used browser.

It's especially ironic because Text Zoom was invented in a Microsoft browser (IE5/Mac) in early 2000. But this incredibly useful feature *still* has not made its way to the Windows side.

This means that if you use pixels to safely control font sizes, you risk making your text inaccessible to visually impaired IE/Windows users. But if you try to avoid that problem by using ems, as we've done on the site we're building in this chapter, you will frustrate visitors who've shrunk their default font size preference to compensate for the fact that the factory-installed default is way too big for most humans.

In short, no matter what you do, you are going to frustrate somebody. We once designed a site setting no font sizes at all. We figured all users would finally be happy. Instead, that site provoked hundreds of angry letters complaining that the text was "too big." For more about the joys and sorrows of font size, see Chapter 13, "Working with Browsers Part III: Typography."

The Wonder of Line-Height

Look again at the line-height declaration in the rule currently being discussed:

```
line-height: 1.5;
```

Line-height is CSS-speak for leading. Line-height of 1.5 is the same as leading of 150%. The line-height could be marked 1.5em, but that is not necessary.

Before CSS, we could only simulate leading by making nonstructural use of the paragraph tag (see the discussion of `Suck.com` in Chapter 1, "99.9% of Websites Are Obsolete"); by using `<pre>`; or by sticking spacer pixel GIF images

between every line of text, forcing that text into table cells of absolute widths, and praying that the invisible spacer pixel images downloaded seamlessly. If they didn't, the visitor would see broken GIF placeholder images instead of lovely leading.

But why even think about it? CSS solves this problem forever.

The Heartache of Left-Align

Finally, if you can find your way back to the previous rule, you might wonder why we've specified text-align: left. The answer is simple. If we don't do that, IE6/Windows might center the text due to a bug. IE5/Windows did not suffer from this defect, nor do any other browsers. The behavior appears to be random. Many elements not overtly left-aligned in CSS will correctly show up left-aligned anyway in IE6/Windows. But some won't. And you never know which ones will do what. Use left-align and you avoid this bug.

Setting Up the Footer

By now, you're hip to the CSS lingo and will be able to understand the little rule that follows…

```
#footer p   {
      font-size: 11px;
      margin-top: 25px;
      }
```

…without our having to tell you that it uses the unique `id` of `footer` as a selector and that any paragraph inside `footer` will be set in a font size of 11px and graced with 25px of whitespace at its top.

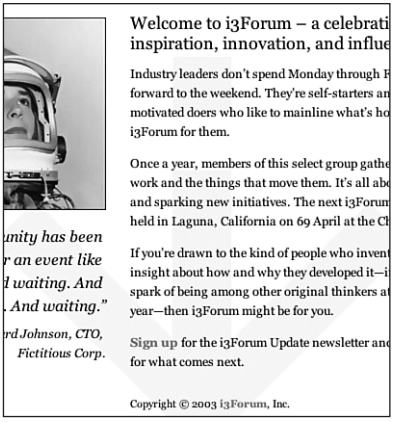
We also don't have to remind you that the browser knows which font to use because it was established by an earlier rule on the page.

Laying Out the Page Divisions

Our next set of rules establishes basic page divisions. We have put them close to each other in our CSS file to make editing and redesign easier, and we've preceded them with a comment to remind us—or to explain to a colleague who might subsequently have to modify our style sheet—what the set of rules is for. If you're familiar with commenting in HTML, it's the same deal here, but with a slightly different convention based on C programming.

Following the comment, we have a rule that establishes that the primary content area will have 25px of whitespace at its left and top [**10.7**], and another that places a nonrepeating graphic background image (arrow.gif) at the bottom and in the center of the table whose `id` is `content` [**10.8**].

10.7
Vertical spacing between the navigation area and body content and horizontal whitespace between the sidebar photo area and the body text are both handled by a single rule applied to the primary content selector. No spacer GIF images or table cell hacks are needed.



10.8
A screened-back arrow image derived from the logo anchors the content area and serves as its watermark thanks to a background declaration applied to the content selector. Because the selector encompasses the entire table, the arrow is able to span the two table cells (sidebar and primary content). Simple as the effect is, achieving it via old-school methods would be difficult if not impossible.

To achieve this effect by means of the deprecated background image attribute to the table cell tag would be difficult if not impossible. For one thing, the background image would have to span two table cells. Therefore, we would need to slice our background image in pieces, assign each piece to a different table cell, and hope all the pieces lined up.

Then, too, the deprecated background image tag in HTML tiles by default, and there is no way to prevent it from tiling. We would have to make two transparent images exactly as tall as the table cells that contain them and pray that the user would not resize the text, thus throwing the table cell heights out of alignment.

We would also have to insist that every page be the same height, which would limit how much text our client could add to or subtract from each page. Our client might not appreciate that.

With CSS, we never have to think about such stupid stuff again. The rules take far less time to read and understand than the paragraphs we just wrote to describe them:

```
/* Basic page divisions */
#primarycontent    {
      padding-left: 25px;
      padding-top: 25px;
      }
#content {
      background: transparent url(images/arrow.gif)center
      ➥bottom no-repeat;
      }
```

Next, we establish rules for the sidebar:

```
/* Sidebar display attributes */
#sidebar p    {
      font-style: italic;
      text-align: right;
      margin-top: 0.5em;
      }
#sidebar img    {
      margin: 30px 0 15px 0;
      }
#sidebar h2    {
      font-size: 1em;
      font-weight: normal;
      font-style: italic;
      margin: 0;
      line-height: 1.5;
      text-align: right;
      }
```

The first rule says that paragraphs within the element whose unique `id` is `sidebar` will be right-aligned and italic and have an upper margin (whitespace)

of half their font size height (0.5em). If it looks familiar, it's because we snuck it into Chapter 9's discussion of `id` selectors in CSS.

The second rule says that images within the element whose unique `id` is `sidebar` will have an upper margin of 30px, a lower margin of 15px, and no extra whitespace at left or right [**10.9**]. We alluded to this rule earlier in the chapter in the section "The Image Rule." Now it has arrived.

10.9
Carefully chosen vertical whitespace values above and below the sidebar photograph are achieved by applying upper and lower margin values to #sidebar img. Images within the `div` labeled `sidebar` will obey these whitespace values; images elsewhere on the site will not.

It's rules like this that make life worth living because they free us from the necessity of using multiple empty table cells and spacer GIF images to create whitespace. (Can you imagine any other visual medium forcing designers to jump through their own eardrums simply to create whitespace? That's how the web was, but we don't have to build it that way any more.)

The third rule in this section makes `h2` headlines look like magazine pull quotes (especially in smooth text environments like Windows Cleartype and Mac OS X Quartz) instead of HTML headlines [**10.10**]. It specifies that `h2` text within `sidebar` will be of modest size (1em), normal weight, italic, and right-aligned, and that it will have the same line-height value (1.5) as other text on the page.

10.10
Sidebar pull quotes, marked up as second-level headlines (`h2`), nevertheless look like magazine pull quotes, not like HTML headlines.

Navigation Elements: First Pass

Up to now, we've been doing all right. Every rule we've written displays as expected in the standards-friendly, best-case-scenario browser we use to test our work. Getting the navigation bar just right will be trickier. In our first pass, shown next, we nail certain desired features and just miss on others:

```
/* Navigation bar components */
table#nav   {
    border-bottom: 1px solid #000;
    border-left: 1px solid #000;
    }
```

This rule tells the table whose `id` is `nav` to create a 1px solid black border effect at its bottom and left (but not at the top or right).

```
table#nav td   {
     font: 11px verdana, arial, sans-serif;
     text-align: center;
     vertical-align: middle;
     border-right: 1px solid #000;
     border-top: 1px solid #000;
     }
```

You don't need us to explain that this rule specifies 11px Verdana as the preferred menu text font and fills out the border elements that the previous rule neglected (namely, at the right and top). If we had told the table to create a border effect around all four sides, then the table cells would have added an extra pixel of border at the top and right, bringing shame to our family and sadness to all viewers.

The preceding rule also tells text to be horizontally centered in each table cell and vertically aligned in the middle of the table cell, much like the old-school `td valign="middle"` presentational hack we all know and love. (This seemed like the right thing to do, but in our second pass, we had to remove it.)

```
table#nav td a   {
     font-weight: normal;
     text-decoration: none;
     display: block;
     margin: 0;
     padding: 0;
     }
```

In the preceding rule, you recognize that we're telling links how to behave, and you also recognize that we're doing so with a sophisticated chain of contextual selectors. The multipart selector means "apply the following rule only to links within table cells, and only if they are found in the table whose unique identifier is `nav`."

As we hinted in the "The Image Rule" discussion, we're also using the CSS `display: block` declaration to turn the humble XHTML links into block-level elements that completely fill their table cells. (At least, we *hope* they will completely fill their table cells.)

```
#nav td a:link, #nav td a:visited {
      background: transparent url(images/bgpat.gif)repeat;
      display: block;
      margin: 0;
      }
#nav td a:hover {
      color: #000;
      background: white url(images/nopat.gif)repeat;
      }
```

The final two rules use contextual and `id` selectors to control the link, visited, and hover pseudo-classes, filling the first two classes with our alternating-pixel background color image [refer to **10.3**] and using a plain-white background image for the hover/rollover state. (See the following sidebar, "Needless Images II: This Time It's Personal.")

Needless Images II: This Time It's Personal

Remember our earlier sidebar about "the needless image," where we said the plain-white background really isn't needed for this execution? Well, the plain-white background really isn't needed for this execution.

Instead of this…

```
#nav td a:hover {
      color: #000;
      background: white url(images/nopat.gif)repeat;
      }
```

continues

continued

…we might have written the following rule:

```
#nav td a:hover {
      color: #000;
      background-image: none;
      }
```

Removing the image from hovered link states (`background-image: none;`) would create the same rollover effect of a plain-white background with one less image to worry about and a bit less bandwidth consumed. A bit later in this chapter, when we create the "you are here" effects for individual pages, we'll do so without relying on the plain-white background GIF.

Nevertheless, because swapped CSS background image swaps are cool and because you might want to harness their power on your own projects, we've used them to create our navigation bar rollover effects in this chapter.

A glance back at Figure **10.1** shows you everything we've gotten right and wrong in our first pass at styling the site. Everything we wanted to achieve we have, except for the navigation bar. The logo is okay; the background color is completely filled in [**10.11**], and on-hover rollover effects [**10.12**] work as expected.

10.11
CSS rollover effects in action, accomplished by means of link, visited, and hover pseudo-classes applied to table cells within the table whose `id` is `nav`. Here we see the default state of a menu graphic.

10.12
CSS rollover effects, part two: When the visitor's cursor hovers over a menu graphic, the background turns white. Look, Ma, no JavaScript! (Not that there's anything wrong with it.)

But the default (link, visited) background pattern [**10.1**] fills only *part* of each right-side menu item, and we intended to fill the entire space. We feel inadequate, vulnerable, and slightly ashamed. In a first attempt at a "final" CSS solution, we will solve this problem but create new ones.

Navigation Bar CSS: First Try at Second Pass

In a first try at a second pass, we specify sizes on our link effects:

```
#nav td a:link, #nav td a:visited {
      background: transparent url(images/bgpat.gif)repeat;
      display: block;
      margin: 0;
      width: 100px;
      height: 25px;
      }
```

As expected, this causes the right-side buttons to be filled in completely, but it louses up our logo, whose background is also now a mere 100 × 25 pixels [**10.13**]. 100 × 25 is the right value for the little buttons, but it's wrong for the logo (which is 400 × 75).

10.13
One step forward, two steps back: Specifying sizes on `nav` pseudo-classes fills in their backgrounds completely but louses up the logo's background. It also obliterates the vertical alignment we established earlier. Text now hugs the top of each cell.

Somehow, the changes we've made also kill the vertical alignment we established earlier. Elements *are* vertically aligned within their cells, but their content is not. As Fig. **10.13** makes plain, "button" text now hugs the top of each table cell instead of being vertically aligned in the middle. This top-hugging presentation is the same in all CSS-compliant browsers tested. It is not a bug, but unexpected behavior that falls out of the CSS layout model.

Fortunately, when creating the markup way back in Chapter 8, we gave each cell of the table a unique identifier. `Home` is the `id` for the cell that contains our

logo. Can we use `home` to create an additional set of rules that override the rules used to fill in the 100 × 25 buttons? You bet we can:

```
td#home a:link, td#home a:visited {
      background: transparent url(images/bgpat.gif) repeat;
      width: 400px;
      height: 75px;
      }
td#home a:hover {
      background: white url(images/nopat.gif) repeat;
      width: 400px;
      height: 75px;
      }
```

These new rules fill in the logo just right, and the site is nearly perfect. But the loss of the behavior we expected with `vertical-align: middle` is still unacceptable. We'll fix it in the final pass.

Navigation Bar CSS: Final Pass

In the final pass, we get everything we wanted:

```
/* Navigation bar components */
table#nav   {
      border-bottom: 1px solid #000;
      border-left: 1px solid #000;
      }
table#nav td   {
      font: 11px verdana, arial, sans-serif;
      text-align: center;
      border-right: 1px solid #000;
      border-top: 1px solid #000;
      }
table#nav td a    {
      font-weight: normal;
      text-decoration: none;
      display: block;
      margin: 0;
      padding: 0;
      }
#nav td a:link, #nav td a:visited {
      background: transparent url(/images/bgpat.gif) repeat;
      display: block;
      margin: 0;
      width: 100px;
```

```
      line-height: 25px;
      }
#nav td a:hover {
      color: #f60;
      background: white url(/images/nopat.gif) repeat;
      }
td#home a:link img, td#home a:visited img {
      color: #c30;
      background: transparent url(/images/bgpat.gif) repeat;
      width: 400px;
      height: 75px;
      }
td#home a:hover img {
      color: #f60;
      background: white url(/images/nopat.gif) repeat;
      width: 400px;
      height: 75px;
      }
```

What changed? We removed the `vertical-align: middle` instruction altogether. Then we deleted the line that said buttons were 25px tall and replaced it with this:

```
      line-height: 25px;
```

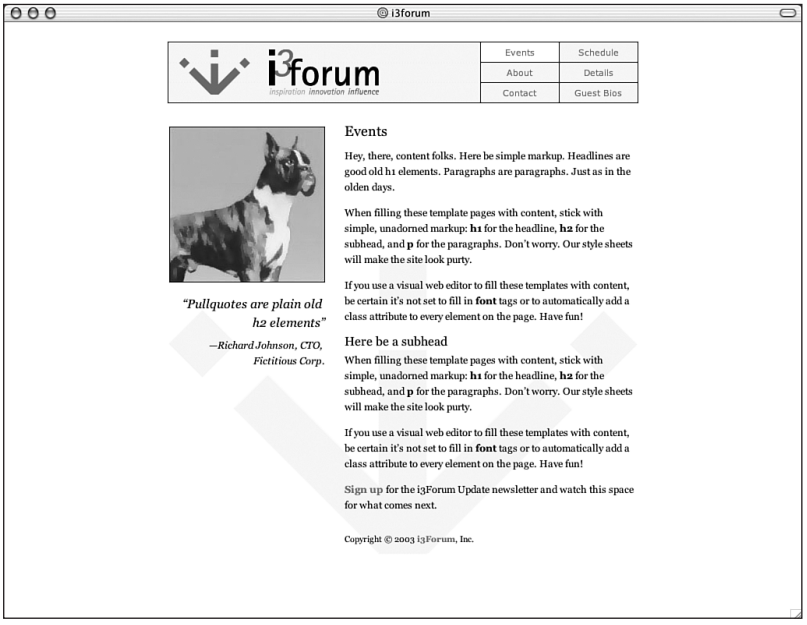
Line-height filled in the 25px just as height had done, but it also correctly positioned the text in the vertical middle of each button. It would take a CSS genius to explain why this method worked better than the other. The main thing is, it worked.

Final Steps: External Styles and the "You Are Here" Effect

To wrap the site and ship it to the client, two more steps are needed. First, we must move our embedded styles to an external CSS file and delete the embedded style sheet, as explained in Chapter 9. Then we must create a "you are here" effect [**10.14**, **10.15**] to help the visitor maintain awareness of which page she's on. Remember: We're not changing the markup. We want to create this effect using CSS, *without* applying additional classes to our navigation bar.

10.14

The "you are here" effect on the Events page template.



10.15

The "you are here" effect on the About page template.

The "you are here" effect is quite easy to do. Now that we've removed embedded styles, every template page gets its CSS data by linking to an embedded style sheet like so:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
 <title>i3forum</title>
<link rel="StyleSheet" href="/css/i3.css" type="text/css"
media="all" />
```

All we need to do is use the style element to add an embedded style sheet to each page. That embedded style sheet will contain just one rule: a rule that inverts the ordinary presentation of the "menu button" for that page. It's easier to show you than to explain it.

On the Events page, the embedded rule reads as follows, with the important selector highlighted in bold:

```
<style type="text/css" media="screen">
td#events a:link, td#events a:visited {
     color: #c30;
      background: #fff;
      }
</style>
```

Notice that we're using `background: #fff;` instead of nopat.gif to create the plain-white background highlighting, as promised in the earlier sidebar, "Needless Images II."

On the About page, the embedded rule reads like so:

```
<style type="text/css" media="screen">
td#about a:link, td#about a:visited {
     color: #c30;
     background: #fff;
     }
</style>
```

Each page of the site contains a rule like this; therefore, every page tells the visitor "you are here" without changing one line of markup. You might ask, "Why not change the markup from page to page? Why not create a `thispage` class for the 'you are here' indicator and use it to override the menu style for that link?" That could certainly be done, and often it is.

But leaving the navigation markup untouched from page to page makes it possible to insert the same data over and over via server-side includes—a handy approach for small- to mid-sized sites like the one we're producing.

At `www.zeldman.com`, we use this technique to modify a pure CSS nav bar's "you are here" indicator on a page-by-page basis while using SSI to insert the same XHTML data on every page. But that's another site and another story, and we've come to the end of another chapter. In the next chapter, we'll plumb new CSS depths, learn about browser bugs and workarounds, discover some things browsers do right (but we might not want), and discuss stubborn elements that seem to resist every effort to use web standards.