FIGURE 33-1

The background color is the color that an object rests on when rendered.

You can use the CSS `background-color` property to define a particular color that should be used for an element's background. The `background-color` property's syntax is similar to other element color properties:

```
background-color:  <color_value>
```

For example, you could use this property to define a navy-blue background for the entire document (or at least its body section):

```
body { background-color:  navy;
       color: white; }
```

This definition also sets a foreground color of white so the default text will be visible against the dark background.

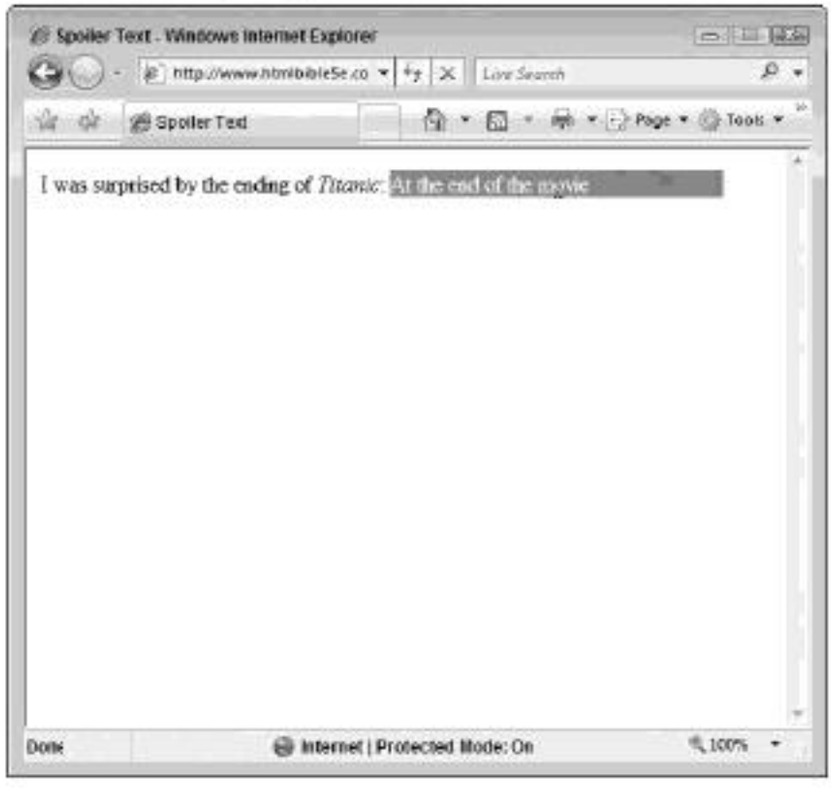
Sometimes it can be advantageous to use similar foreground and background colors together. For example, on a forum that pertains to movie reviews, users may wish to publish spoilers — pieces of the plot that others may not wish to know prior to seeing the movie. On this type of site, a

style can be defined such that the text cannot be viewed until it is selected in the user agent, as shown in Figure 33-2. You could define the style as follows:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Spoiler Text</title>
  <style type="text/css">
    .spoiler | background-color: gray; color: gray; |
  </style>
</head>
<body>
<p>I was surprised by the ending of <i>Titanic</i>:
   <span class="spoiler">At the end of the movie, the boat sinks.
   </span></p>
</body>
</html>
```

FIGURE 33-2

A non-contrasting background has its uses.



Note that an element's background extends to the end of its padding. If you want to enlarge the background of an element, expand its padding accordingly. For example, both paragraphs in Figure 33-3 have a lightly colored background. However, the second paragraph has had its padding expanded, as shown in the following code:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Expanding Backgrounds</title>
  <style type="text/css">
    p { background-color: #CCCCCC; }
    p.larger-background { padding: 20px; }
  </style>
</head>
<body>
<p>The Oasis boasts three saunas, two whirlpools, and a full-size
swimming pool for the use of our clients. Each of these facilities
has a small usage fee, but many services include the use of one or
more of these facilities--be sure to ask your service consultant
about our many combination packages.</p>
<p class="larger-background">The Oasis boasts three saunas, two
whirlpools, and a full-size swimming pool for the use of our clients.
Each of these facilities has a small usage fee, but many services
include the use of one or more of these facilities--be sure to ask
your service consultant about our many combination packages.</p>
</body>
</html>
```

FIGURE 33-3

An object's background spans the size of its padding.

Background Images

In addition to solid colors, you can specify an image for an element's background. To do so, you use the `background-image` property. This property has the following syntax:

```
background-image: url("<url_to_image>");
```

For example, the following code produces the document rendered in Figure 33-4, where the paragraph is rendered over a light gradient image (`gradient.gif`):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Background Images</title>
  <style type="text/css">
    p.gradient { background-image: url("gradient.gif");
                 padding: 20px; }
  </style>
</head>
<body>
 <p class="gradient">The Oasis boasts three saunas, two whirlpools,
and a full-size swimming pool for the use of our clients. Each of
these facilities has a small usage fee, but many services include the
use of one or more of these facilities--be sure to ask your service
consultant about our many combination packages.</p>
 <p>Background image:<br />
 <img src="gradient.gif" alt="gradient" width="400"
  height="300" /></p>
</body>
</html>
```

Background images can be used for interesting effects, such as that shown in Figure 33-5, rendered from the following code:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Text Frame</title>
  <style type="text/css">
    div.sightbox { height: 300px; width: 400px;
       background-image: url("sightframe.jpg");
       background-repeat: no-repeat; }
    div.sightbox p { font-size: small;
                     padding: 22px 120px 22px 25px; }
  </style>
</head>
<body>
<div class="sightbox">
```

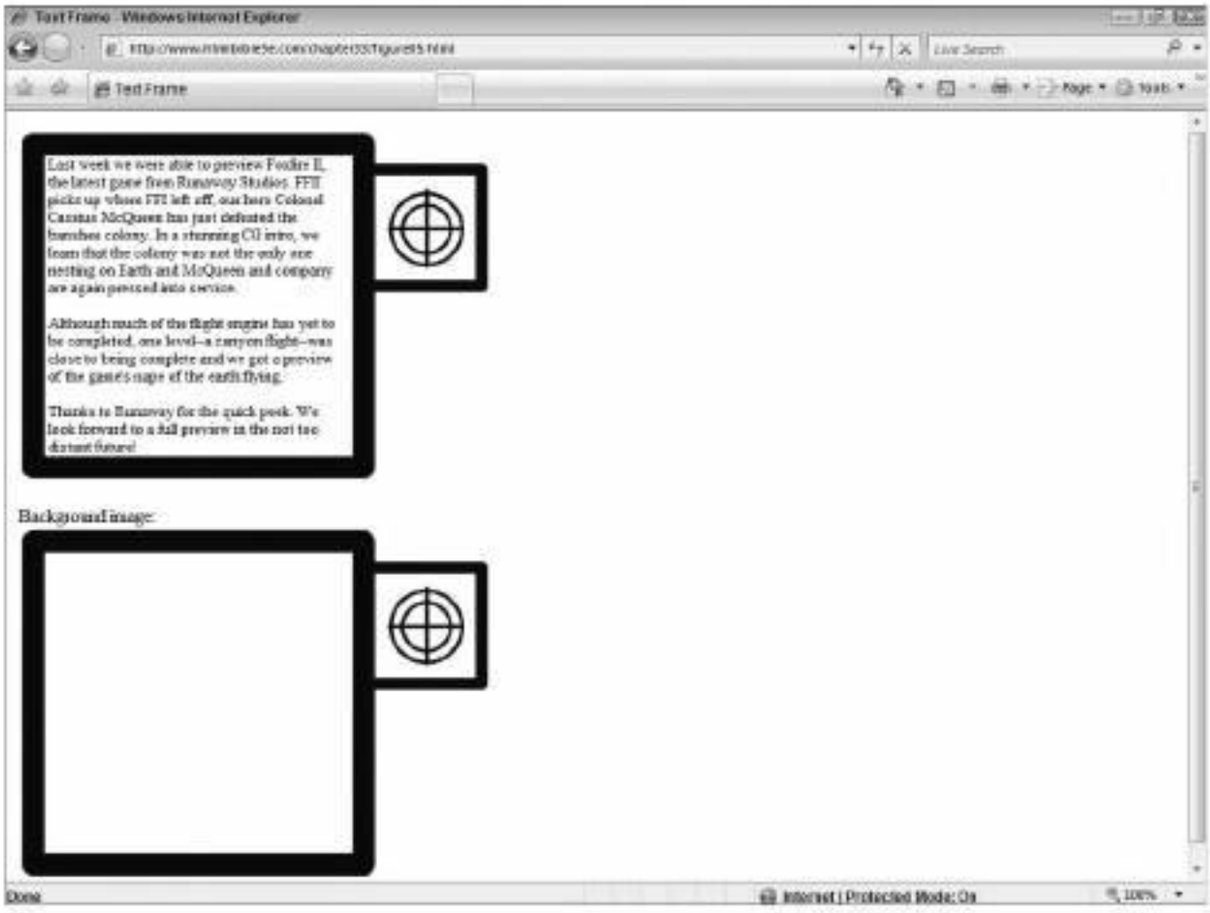
```
 <p>Last week we were able to preview Foxfire II, the latest game
from Runaway Studios. FFII picks up where FFI left off, our hero
Colonel Cassius McQueen has just defeated the banshee colony. In a
stunning CG intro, we learn that the colony was not the only one
nesting on Earth and McQueen and company are again pressed into
service.<br/><br/>
 Although much of the flight engine has yet to be completed, one
level--a canyon flight--was close to being complete and we got a
preview of the game's nape of the earth flying.<br/><br/>
 Thanks to Runaway for the quick peek. We look forward to a full
preview in the not too distant future!</p>
 </div>
 <p>Background image:<br />
 <img src="sightframe.jpg" alt="sight frame" width="400"
   height="300" /></p>
 </body>
 </html>
```

FIGURE 33-4

Images such as this light gradient can also be used as backgrounds for objects.

FIGURE 33-5

Background images can be used for interesting effects, such as frames around text.



Note how the various sides of the paragraph were padded to ensure that the text appears in the correct position relative to the background frame without overfilling it.

Cross-Ref

Additional CSS formatting and display tips and tricks are covered in Chapter 41. ■

Repeating and scrolling images

Element background images tile themselves to fill the available space, as you saw in Figure 33-4 where the gradient tiles horizontally to span the width of the paragraph. You can control the scrolling and placement properties of a background image using the `background-repeat` and `background-attachment` properties.

The `background-repeat` property has the following syntax:

```
background-repeat: repeat | repeat-x | repeat-y | no-repeat;
```

The `background-attachment` property has the following format:

```
background-attachment: <i>scroll | fixed</i>;
```

Using the `background-repeat` property is straightforward — its values specify how the image repeats. For example, to repeat the crosshairs across the top of the paragraph, specify `repeat-x`, as shown in the following definition code and Figure 33-6:

```
div.sightfill | background-image: url("sight.gif");
                background-repeat: repeat-x;
                /* Border to clarity paragraph */
                border: thin solid black;
                padding: 10px; |
```

FIGURE 33-6

Images used as backgrounds can also be tiled vertically, horizontally, or both.

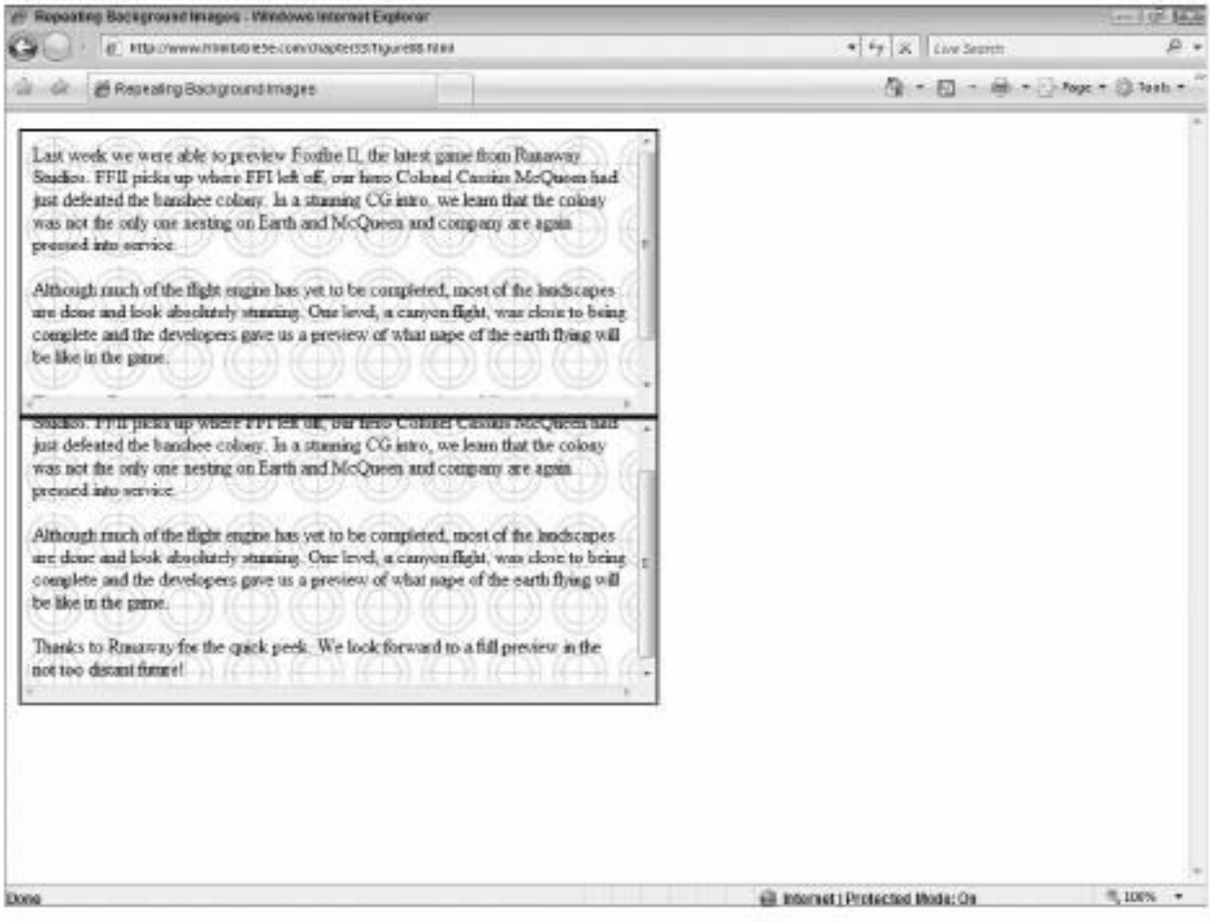
Specifying `repeat-y` would repeat the image vertically instead of horizontally. If you specify just `repeat`, the image tiles both horizontally and vertically. Specifying `no-repeat` will cause the image to be placed once only, not repeating in either dimension.

The `background-attachment` property specifies how the background image is attached to the element. Specifying `scroll` allows the image to scroll with the contents of the element, as shown by the second paragraph in Figure 33-7. Both paragraphs were rendered with the following paragraph definition — the second paragraph has been scrolled a bit, vertically shifting both text and image:

```
div.sightscroll { height: 220px; width: 520px;
                  /* Scroll the element's content */
                  overflow: scroll;
                 /* Define a background image and set
                    it to scroll */
                 background-image: url("sight.gif");
                 background-attachment: scroll;
                 /* Border for clarity only */
                 border: thin solid black;
                 padding: 10px; }
```

FIGURE 33-7

Background images can be fixed in place or set to scroll with an object (notice the images scrolling with the text in the second box).



Specifying a value of `fixed` for the `background-attachment` property will fix the background image in place, causing it not to scroll if/when the element's content is scrolled. This value is particularly useful for images used as the background for entire documents for a watermark effect.

Note

Using the `overflow` property in the code for Figure 33-7 controls what happens when an element's content is larger than its containing box. The `scroll` value enables scroll bars on the element so users can scroll to see the entire content. The `overflow` property also supports the values `visible` (which causes the element to be displayed in its entirety, despite its containing box size) and `hidden` (which causes the portion of the element that overflows to be clipped and remain inaccessible to the user). ■

Positioning background images

You can use the `background-position` property to control where an element's background image is placed in relation to the element's containing box. The `background-position` property's syntax isn't as straightforward as some of the other properties. This property has three different forms for its values:

- Two percentages are used to specify where the upper-left corner of the image should be placed in relation to the element's padding area.
- Two lengths (in inches, centimeters, pixels, ems, and so on) specify where the upper-left corner of the image should be placed in relation to the element's padding area.
- Keywords specify absolute measures of the element's padding area. The supported keywords include `top`, `left`, `right`, `bottom`, and `center`.

No matter what format you use for the background-position values, the syntax for the definition is as follows:

```
background-position: <horizontal_value> <vertical_value>;
```

If only one value is given, it is used for the horizontal placement and the image is centered vertically. The first two formats can be mixed together (for example, `10px 25%`), but keywords cannot be mixed with other values (for example, `center 30%` is invalid).

To center a background image behind an element, you can use either of the following definitions:

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

If you want to specify an absolute position behind the element, you can do so as well:

```
background-position:  10px 10px;
```

Tip

You can combine the background image properties to achieve diverse effects. For example, you can use `background-position` to set an image to appear in the center of the element's padding, and specify `background-attachment: fixed` to keep it there. Furthermore, you could use `background-repeat` to repeat the same image horizontally or vertically, creating striping behind the element. ■

The background shortcut property

The `background` property is one of the more powerful shortcut properties, combining the `background-color`, `background-image`, `background-repeat`, `background-attachment`, and `background-position` properties in one property declaration. It has the following syntax:

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

You can use this shortcut property, for example, to define a background image and its particulars all together:

```
background: url('/images/joystickicon.jpg') repeat-x left center;
```

As with all shortcut properties, you should balance your use of it between convenience and readability. Generally speaking, it's easier to gather all information about an element from within one aggregated property, but longer property declarations require decoding of their own.

Summary

This chapter covered how CSS can affect the background and foreground of an element, employing colors and images. You learned how to change the text color of an element using the `color` property and how to use the `background` properties to change an element's background. The last chapters in this part of the book cover CSS formatting, generated and dynamic content, and how to define pages for printing.

CHAPTER 34

CSS Layouts

In the various chapters within this part, you have seen how dynamic documents can be when formatted with CSS. This chapter describes how you can position elements to create various page layouts using CSS properties.

IN THIS CHAPTER

Understanding CSS Positioning

There are several ways to position elements using CSS. Which method you use depends on what you want the element's position to be in reference to and how you want the element to affect other elements around it. The following sections cover the three main positioning models.

Note

It is important to include a valid DTD within documents using positioning. Without a valid DTD the browser might be prone to slipping into *quirks mode* and refuse to position your elements properly. For more information on quirks mode, see `www.quirksmode.org`. For more information on DTDs, see Chapter 1 of this book. ■

Static positioning

Static positioning is the standard positioning model — elements are rendered inline or within their respective blocks as normal. Figure 34-1 shows three paragraphs; the middle paragraph has the following styles applied to it:

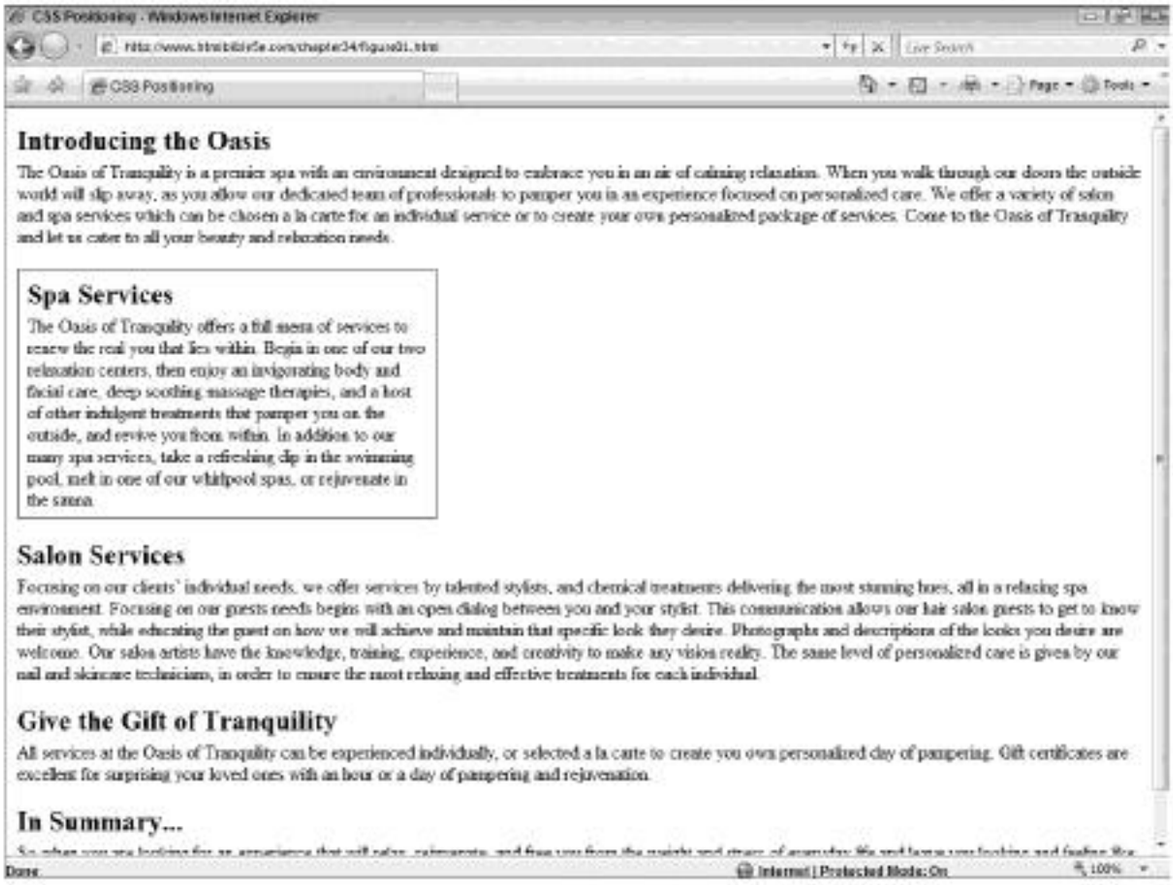
```
width: 350px;
height: 200px;
border: 1pt solid black;
background-color: white;
padding: .5em;
position: static;
```

Note

Several styles have been inserted for consistency throughout the examples in this section. A border and background have been added to the element to enhance the visibility of the element's scope and position. The element also has two positioning properties (top and left), although they do not affect the static positioning model. ■

FIGURE 34-1

Static positioning is the normal positioning model, rendering elements where they should naturally be.



Relative positioning

Relative positioning is used to move an element from its normal position — where it would normally be rendered — to a new position. The new position is *relative* to the normal position of the element.

Figure 34-2 shows the second paragraph positioned using the relative positional model. The paragraph is positioned using the following styles (pay particular attention to the last three: `position`, `top`, and `left`):

```
width: 350px;
height: 200px;
```

```
border: 1pt solid black;
background-color: white;
padding: .5em;
position: relative;
top: 100px;
left: 100px;
```

FIGURE 34-2

Relatively positioned elements are positioned *relative* to the position they would otherwise occupy.

With relative positioning, you can use the side positioning properties (`top`, `left`, and so on) to position the element. Note the one major side-effect of using relative positioning: The space where the element would normally be positioned is left open, as though the element were positioned there.

Note

The size of the element is determined by the sizing properties (`width` or `height`), the positioning of the element's corners (via `top`, `left`, and so on), or by a combination of properties. ■

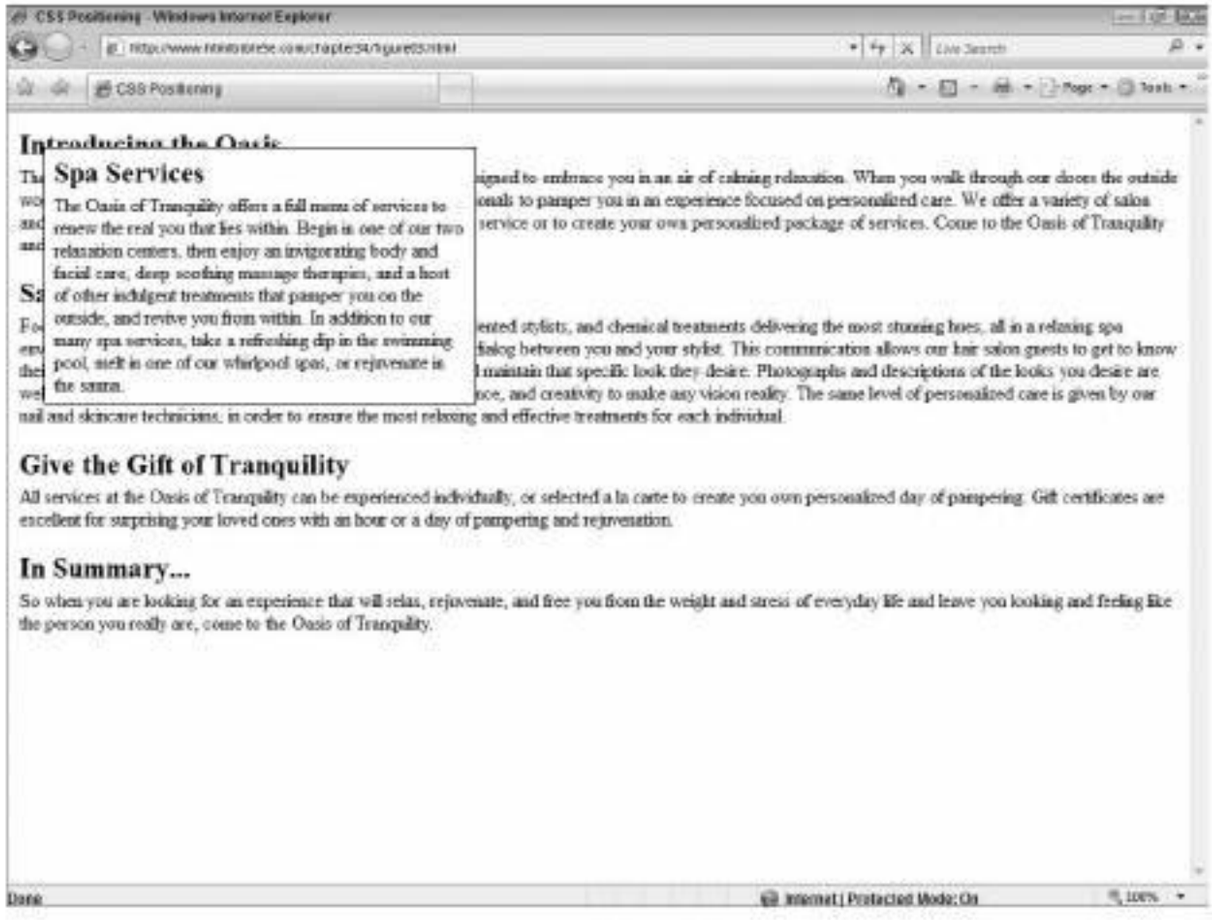
Absolute positioning

Elements using *absolute positioning* are placed in an absolute position, relative to the viewport instead of their normal position in the document. For example, the following styles are used to position the second paragraph in Figure 34-3:

```
width: 350px;
height: 200px;
border: 1pt solid black;
background-color: white;
padding: .5em;
position: absolute;
top: 30px;
left: 30px;
```

FIGURE 34-3

The absolute positioning model uses the user agent's viewport for positioning reference.



Note that the positioning properties are referenced against the viewport when using the absolute positioning model. The element in this example is positioned 30px from the top and 30px from the left of the viewport's edges.

Unlike the relative positioning model, absolute positioning does not leave space where the element would have been positioned. Neighboring elements position themselves as though the element were not present in the rendering stream.

Fixed positioning

Fixed positioning is similar to absolute positioning in that the element is positioned relative to the viewport. However, fixed positioning causes the element to be fixed in the viewport — it will not scroll when the document is scrolled; it maintains its position. The following code is used to position the second paragraph shown in Figures 34-4 and 34-5:

```
width: 350px;
height: 200px;
border: 1pt solid black;
background-color: white;
padding: .5em;
position: fixed;
top: 100px;
left: 100px;
```

FIGURE 34-4

Elements using the fixed positioning model are positioned relative to the viewport, much like absolute positioning.

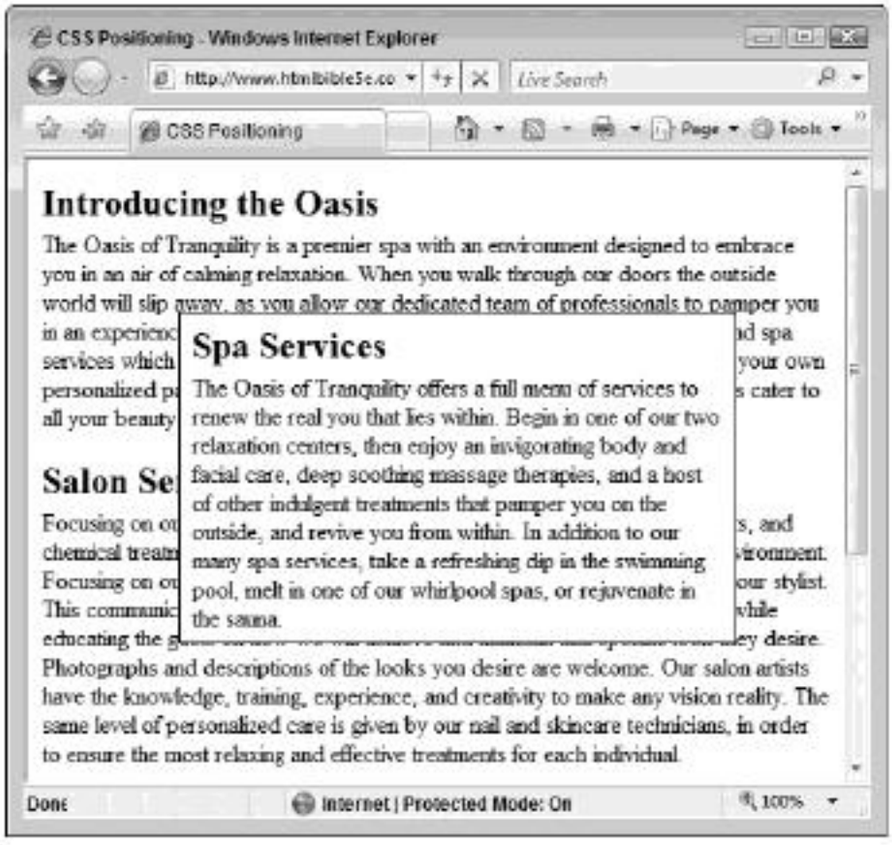


FIGURE 34-5

Elements using the fixed positioning model do not scroll within the viewport, as shown when this document scrolls (note the scroll bar's position compared to that in Figure 34-4).

Note that when the document scrolls (refer to Figure 34-5) the fixed element stays put.

Note

Not all user agents support all the positioning models. In addition, some user agents change their support between versions — the positioning support in Internet Explorer 7 is different from that in Internet Explorer 6, for example. Before relying upon a particular model in your documents, you should test the documents in your target user agents. ■

Specifying the Element Position

Element positioning can be controlled by four positioning properties: `top`, `right`, `bottom`, and `left`. The effect of these properties on the element's position is largely driven by the type of positioning being used on the element.

The positioning properties have the following format:

```
<side>: <length> | <percentage> ;
```

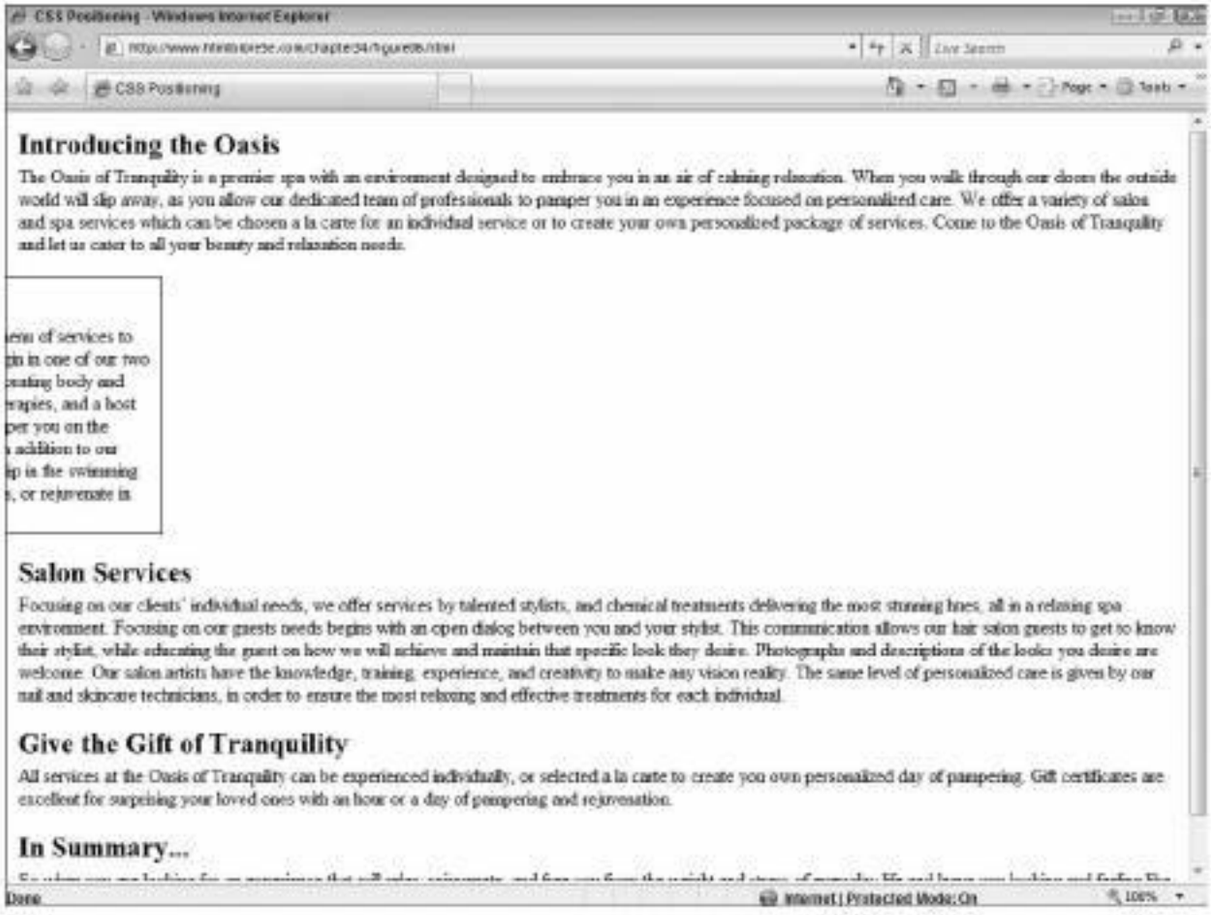
The specified side of the element is positioned according to the value provided. If the value is a length, the value is applied to the reference point for the positioning model being used — the element's otherwise normal position if the relative model is used, the viewport if the absolute or fixed model is used. For example, consider the following code:

```
position: relative;
right: 25%;
```

These settings result in the element being shifted to the left by 25 percent of its width, as shown in Figure 34-6. This is because the user agent is told to position the right side of the element 25 percent of the element's width from where it should be.

FIGURE 34-6

A relative, 50% right value results in an element being shifted to the left by 50% of its width.



Note

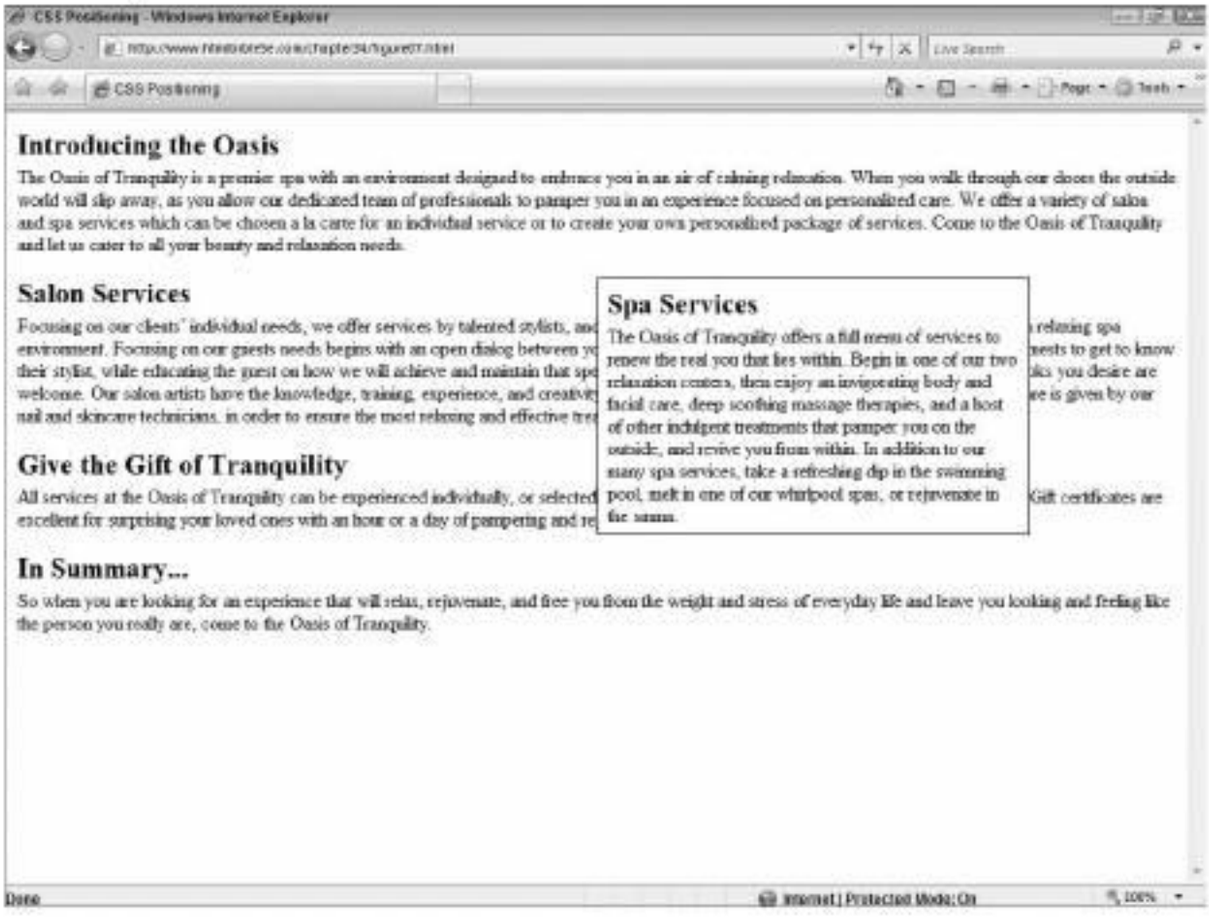
This type of positioning can be confusing. For example, positioning an element *right* 50 percent means that it will be positioned 50 percent away from where it would have been on the right, which is the same as 50 percent toward the *left*. ■

However, if the following settings are used, the element is positioned with its left side in the horizontal center of the viewport, as shown in Figure 34-7:

```
position: absolute;
left: 50%;
```

FIGURE 34-7

An absolute, 50% left value results in an element being shifted such that its left side is in the middle of the viewport.



Here, the user agent references the positioning against the viewport (`absolute`), so the element's left side is positioned at the horizontal 50-percent mark of the viewport.

Note

Positioning alone can drive the element's size. For example, the following code will result in the element being scaled horizontally to 25 percent of the viewport, the left side positioned at the 25-percent horizontal mark, and the right at the 50-percent horizontal mark:

```
position: absolute;
left: 25%;
right: 50%;
```

Whichever property appears last in the definition has the most influence over the final size of the element. For example, the following definition will result in an element that has its left side positioned at the viewport's horizontal 25-percent mark, but is 300 pixels wide (despite the size of the viewport):

```
position: absolute;
left: 25%;
right: 50%;
width: 300px;
```

The `width` property overrides the `right` property because of the cascade effect of CSS. ■

Floating Elements to the Left or Right

Another method to position elements is to *float* them outside of the normal flow of elements. When elements are floated, they remove themselves from their normal position and float to the specified margin.

The `float` property is used to float elements. This property has the following format:

```
float: right | left | none;
```

If the property is set to `right`, the element is floated to the right margin. If the property is set to `left`, the element is floated to the left margin. If the property is set to `none`, the element maintains its normal position according to the rest of its formatting properties. If the element is floated to a margin, the other elements will wrap around the opposite side of the element. For instance, if an element is floated to the right margin, the other elements wrap on the left side of the element's bounding box.

Compare the image in Figure 34-8, which is not floated and appears in the normal flow of elements, with the same image floated to the right margin (via the style `float: right`) in Figure 34-9.

FIGURE 34-8

A nonfloated image is rendered where its tag appears.

Cross-Ref

If you don't want elements to wrap around a floated element, you can use the `clear` property to keep the element away from floaters. See Chapter 30 for more information on the `clear` property. ■

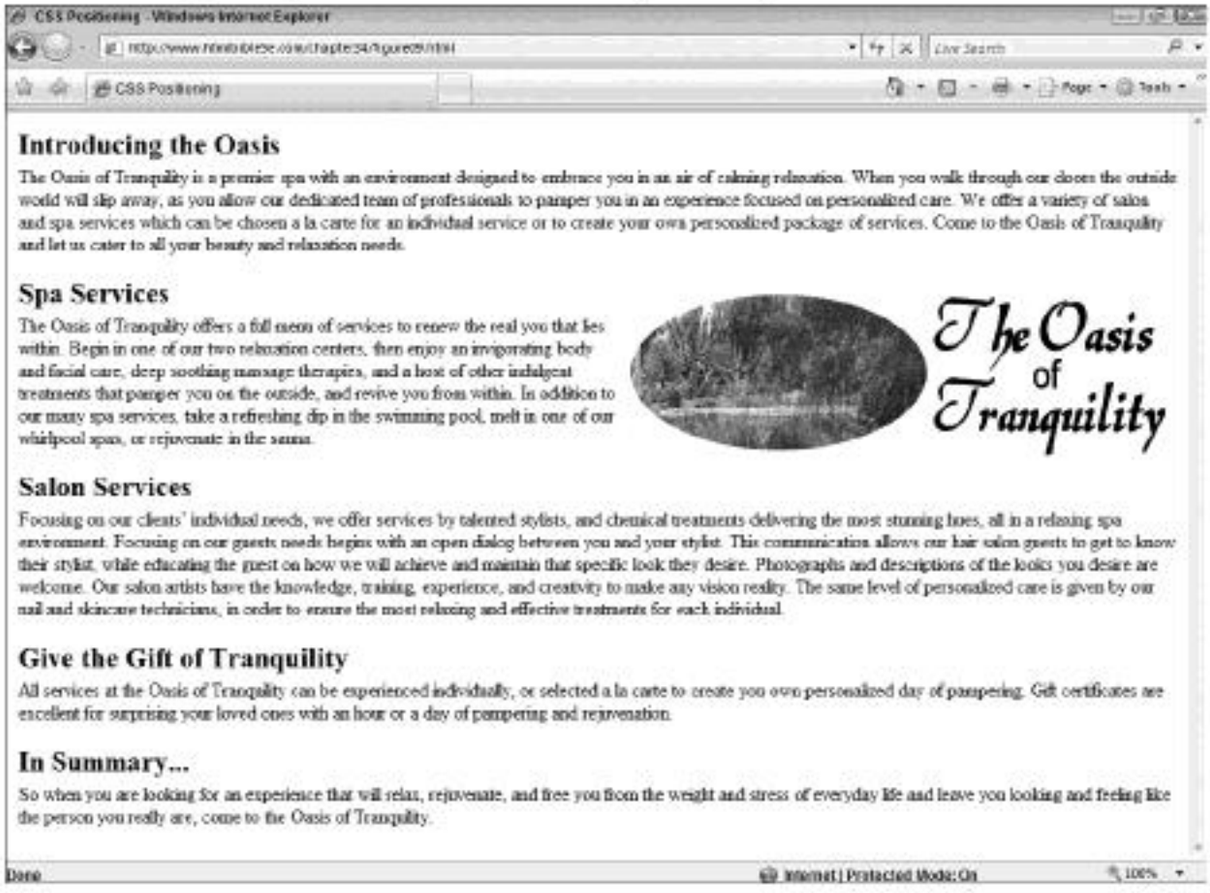
The float property can also be used to effectively create parallel columns from elements. For example, consider the following code, whose output is shown in Figure 34-10:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
  "http://www.w3.org/TR/html4/frameset.dtd">
<html>
<head>
  <title>Parallel Floated Divs</title>
  <style type="text/css">
  div {
    border: 1pt solid black;
    padding: 10px;
```

```
    width: 200px;
    height: 400px;
    float: left;
  }
  </style>
</head>
<body>
<div id="1"><p>This is text for div 1</p></div>
<div id="2"><p>This is text for div 2</p></div>
<div id="3"><p>This is text for div 3</p></div>
</body>
</html>
```

FIGURE 34-9

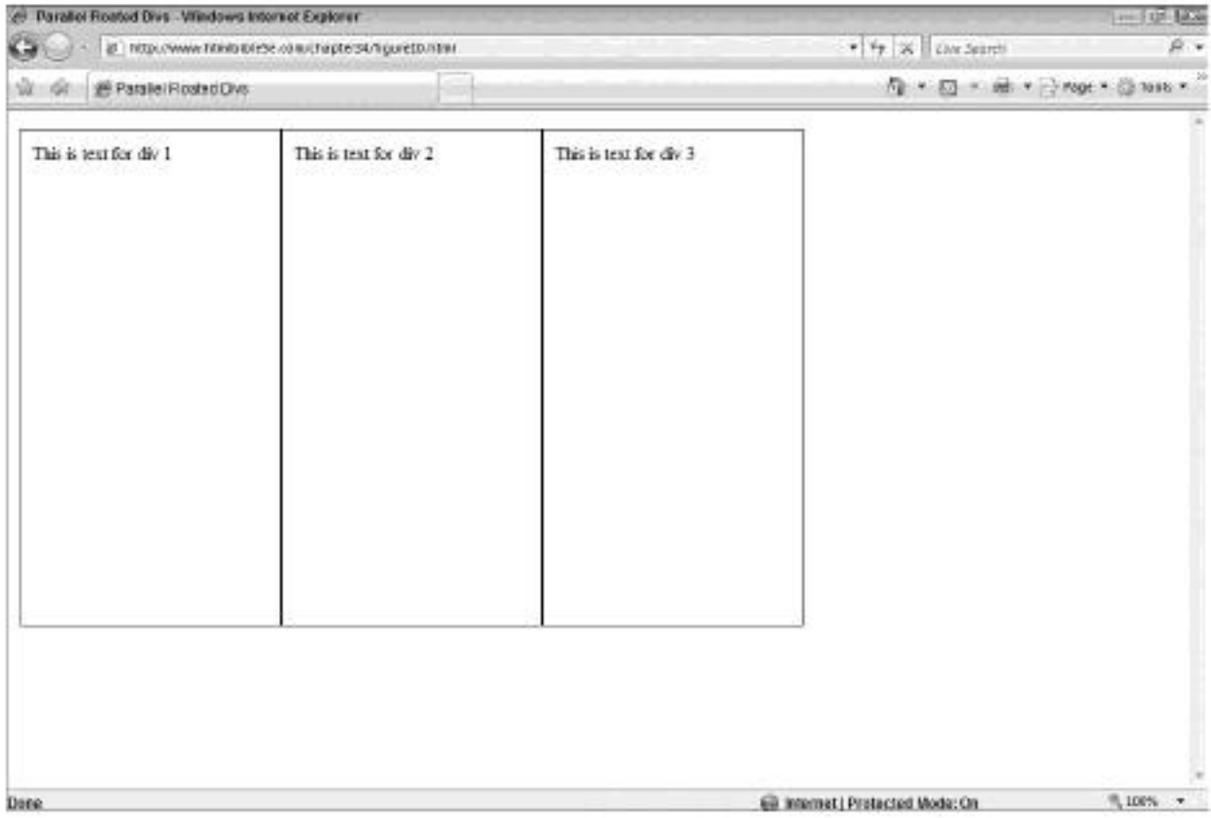
An image that is floated is removed from the normal flow and is moved to the specified margin (in this case, the right margin). The other elements wrap on the exposed side of the element.



To create margins for the text within the columns, use the corresponding element's `padding` properties. Likewise, to increase the margins outside the columns — between the column and neighboring elements — use the corresponding element's `margin` properties.

FIGURE 34-10

Floated elements stacked against one another can be used for multiple, parallel columns.



Defining an Element's Width and Height

There are multiple ways to affect an element's size. You have seen how other formatting can change an element's size; in the absence of explicit sizing instructions the user agent does its best to make everything fit. However, if you want to intervene and explicitly size an element, you can. The following sections show you how.

Specifying exact sizes

You can use the `width` and `height` properties to set the size of the element. For example, if you want a particular section of the document to be exactly 200 pixels wide, you can enclose the section in the following `<div>` tag:

```
<div style="width: 200px;"> ... </div>
```

Likewise, if you want a particular element to be a certain height, you can specify the height using the `height` property.

Note

Keep in mind that you can set size constraints — minimum and maximum sizes — as well as explicit sizes. See the next section for details on minimum and maximum sizes. ■

Specifying maximum and minimum sizes

There are properties to set maximum and minimum sizes for elements as well as explicit sizes. At times, you will want the user agent to be free to size elements by using the formatting surrounding the element, while still maintaining some constraint to ensure that an element will be displayed in its entirety, rather than be clipped or displayed in a sea of white space.

Note

Most user agents do not support min and max CSS settings. ■

You can use the following properties to constrain an element's size:

- `min-width`
- `max-width`
- `min-height`
- `max-height`

Each property takes a length or percentage value to limit the element's size. For example, to limit the element from shrinking to less than 200 pixels in height, you could use the following:

```
min-height: 200px;
```

Tip

All of the length and width properties accept values in any acceptable length format — pixels, ems, points, percentages, and so on. ■

Controlling element overflow

Whenever an element is sized independently of its content, there is a risk of the element becoming too small for its content. For example, consider the paragraphs in Figure 34-11. They are the same except that the second paragraph has had its containing box specified too small, causing the contents to fall outside of the border.

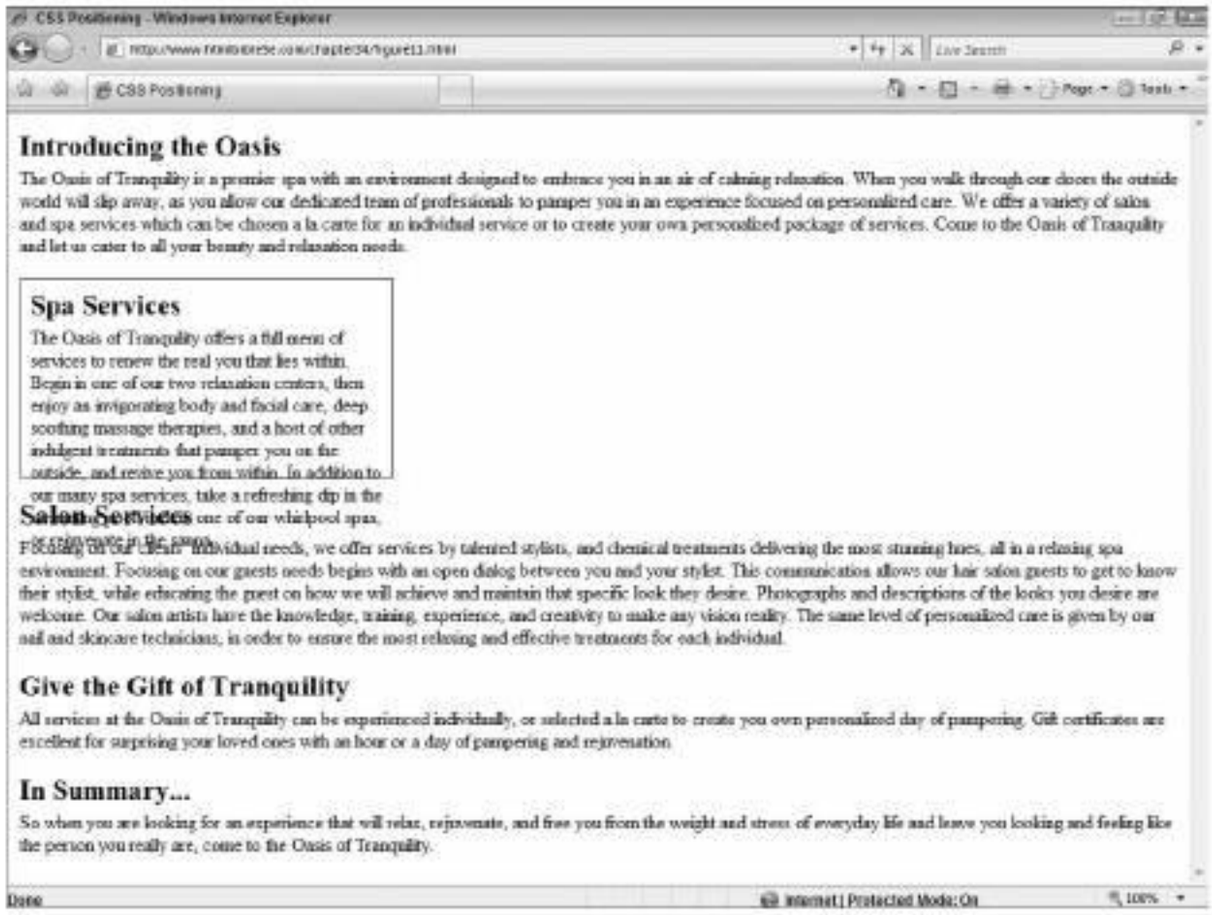
In this example, the user agent chose to display the rest of the element outside its bounding box. Other user agents may crop the element or refuse to display it at all.

If you want to control how the user agent handles mismatched elements and content sizes, use the `overflow` property. This property has the following format:

```
overflow: visible | hidden | scroll | auto;
```

FIGURE 34-11

An element that is mis-sized doesn't always handle its content properly.



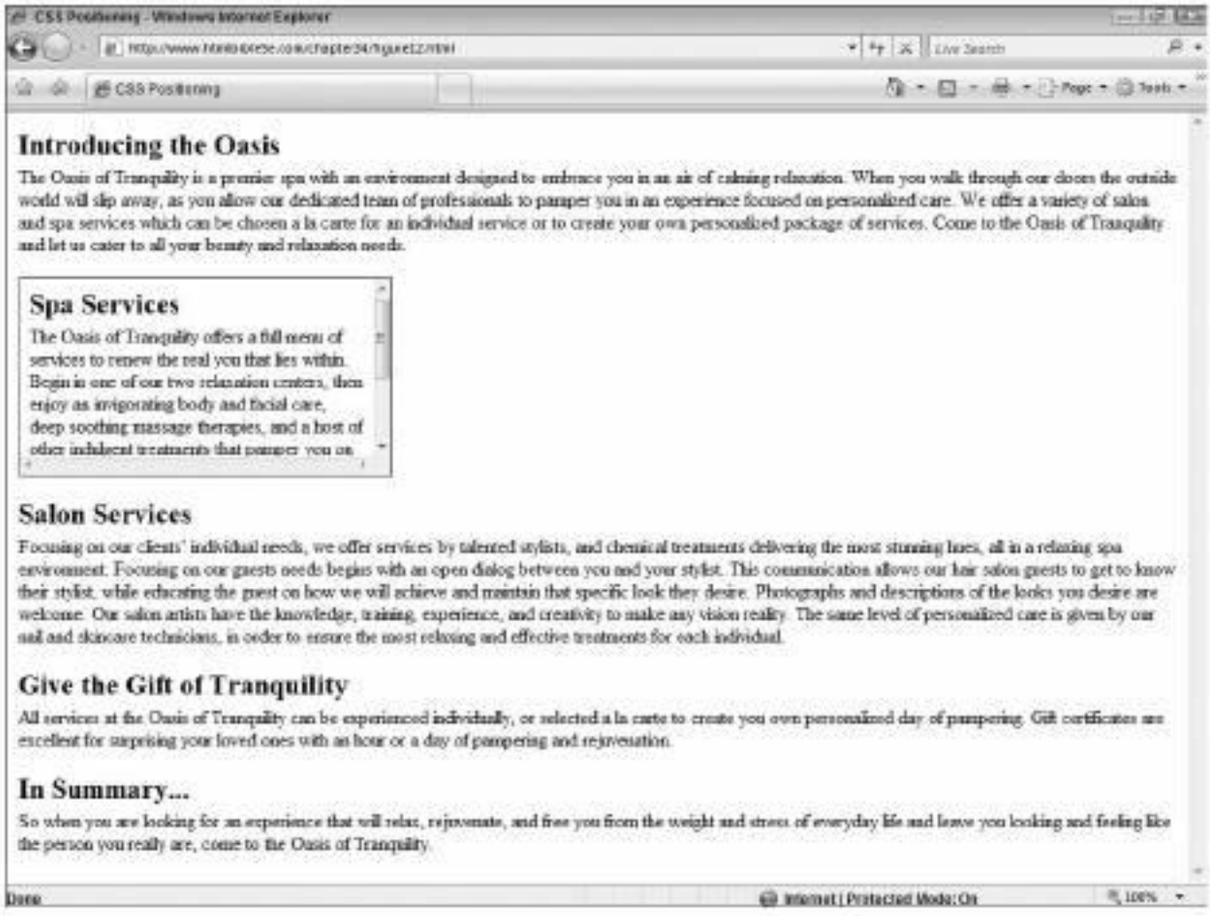
The values have the following effect:

- `visible` — The content is not clipped and is displayed outside of its bounding box, if necessary (refer to Figure 34-11).
- `hidden` — If the content is larger than its container, the content will be clipped. The clipped portion will not be visible, and the user will have no way to access it.
- `scroll` — If the content is larger than its container, the user agent should contain the content within the container but supply a mechanism for the user to access the rest of the content (usually through scroll bars).
- `auto` — The handling of element contents is left up to the user agent. Overflows, if they happen, are handled by the user agent's default overflow method.

Figure 34-12 shows the same paragraph shown in Figure 34-11, but with its `overflow` property set to `scroll`. The user agent obliges by providing scroll bars to access the rest of the element's content.

FIGURE 34-12

When the overflow property is set to scroll, the user agent supplies a mechanism (usually scroll bars) to view the entire content.



Stacking Elements in Layers

Using CSS positioning often causes elements to be stacked on top of one another. Usually, you can anticipate how the elements will stack and leave the user agent to its own devices regarding the final display of stacked elements. At times, however, you will want to explicitly specify how overlapping elements stack. To control the stacking of elements, you use the `z-index` property.

The `z-index` property has this format:

```
z-index:  value;
```

This property controls where elements should be positioned in the third dimension of the otherwise flat HTML media. Because the third dimension is typically referred to along a Z axis, this property is named accordingly (with a Z). You can think of the z-stack as papers stacked on a desktop, overlapping each other — some of the papers are covered by others.

The value of the `z-index` property controls where on the stack the element should be placed. The beginning reference (the document) is typically at index 0 (zero). Higher numbers place the element higher in the stack, as shown in the diagram in Figure 34-13.

FIGURE 34-13

The effect of the z-index property

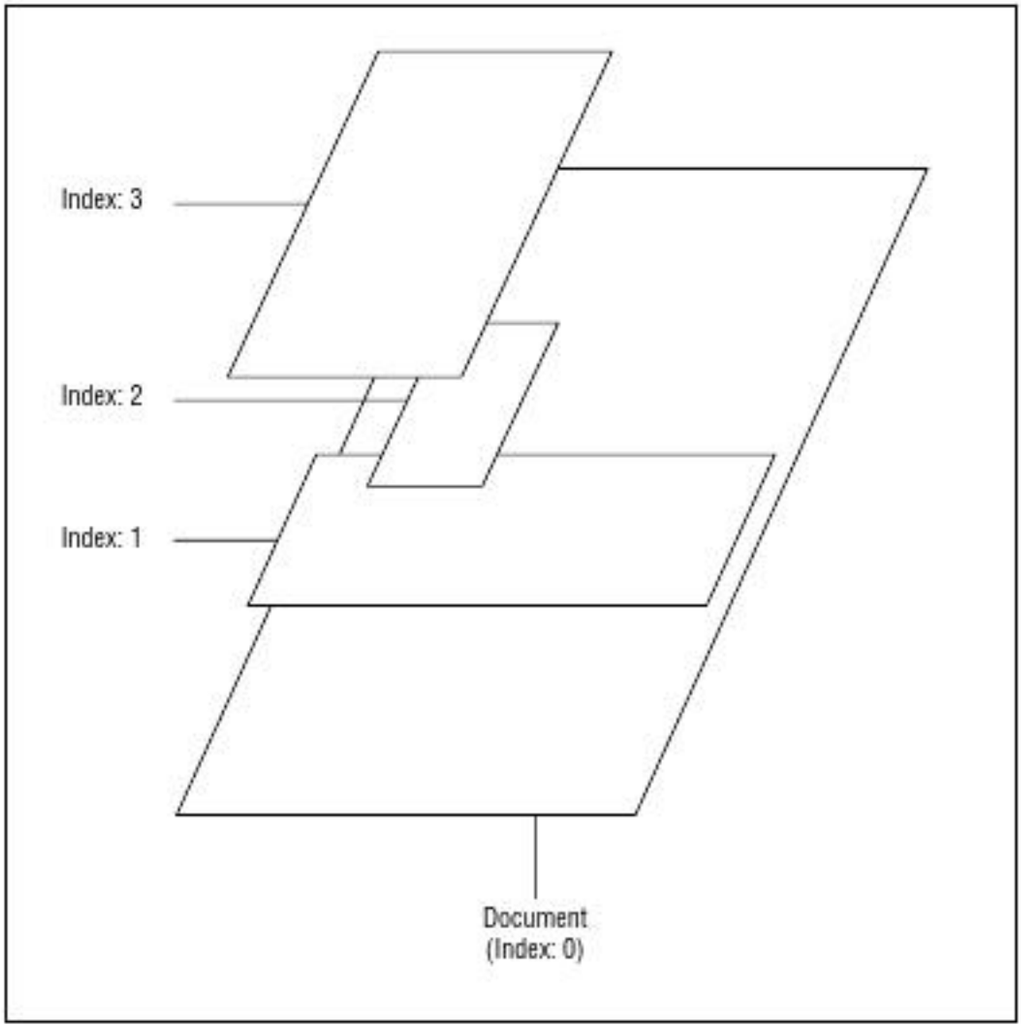


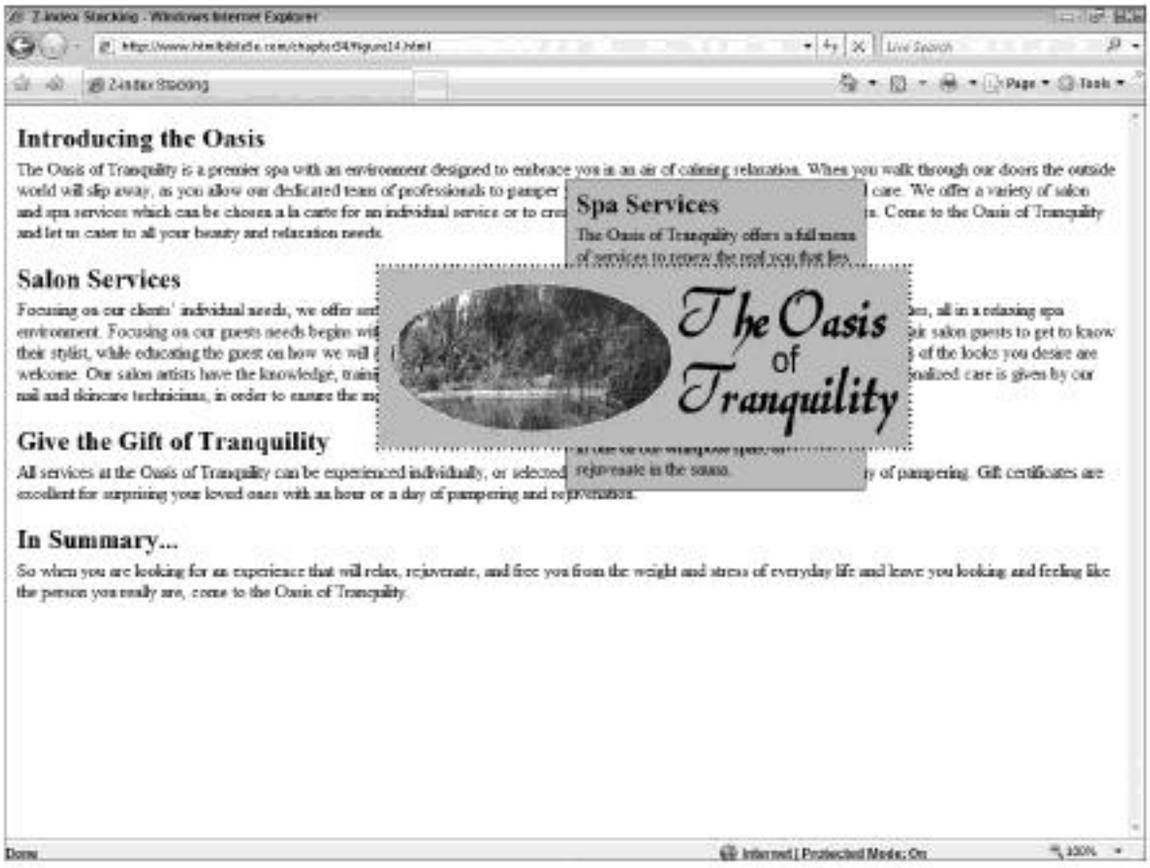
Figure 34-14 offers a practical example of `z-index` stacking. Each element is assigned a z-index, as shown in the following code:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Z-index Stacking</title>
  <style type="text/css">
```

```
        .box1 { position: absolute;
                top: 25%;
                left: 25%;
                width: 200px;
                height: 200px;
                background-color: red;
                color: white;
                z-index: 200; }
        .box2 { width: 400px;
                height: 400px;
                background-color: yellow;
                z-index: 100; }
        .box3 { width: 400px;
                height: 100px;
                background-color: green;
                position: absolute;
                top: 20%;
                left: 10%;
                color: white;
                z-index: 150; }
    </style>
  </head>
  <body>
  <div class="box2">
  <p><b>Box 2:</b> Lorem ipsum dolor sit amet, consectetuer adipiscing
elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna
aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud
exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea
commodo consequat. Duis autem vel eum iriure dolor in hendrerit in
vulputate velit esse molestie consequat, vel illum dolore eu feugiat
nulla facilisis at vero eros et accumsan et iusto odio dignissim qui
blandit praesent luptatum zzril delenit augue duis dolore te feugait
nulla facilisi.</p>
  <p class="box1"><b>Box 1:</b> This is text</p>
  <p>Ut wisi enim ad minim veniam, quis nostrud exerci tation
ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo
consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate
velit esse molestie consequat, vel illum dolore eu feugiat nulla
facilisis at vero eros et accumsan et iusto odio dignissim qui
blandit praesent luptatum zzril delenit augue duis dolore te feugait
nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing
elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore
magna aliquam erat volutpat.</p>
  </div>
  <div class="box3">
    <p><b>Box 3:</b> This is text.</p>
  </div>
  </body>
  </html>
```

FIGURE 34-14

A sample of z-index stacking



The code uses a mix of `div` and `p` elements for diversity. Because the index for `box1` is the highest (`200`), it is rendered on the top of the stack. The index for `box3` is the next highest (`150`), so it is rendered second to the top. The index for `box2` is the lowest (`100`), so it is rendered near the bottom. The document itself is recognized as being at `0`, so its content and any other unspecified elements are rendered at the bottom of the stack.

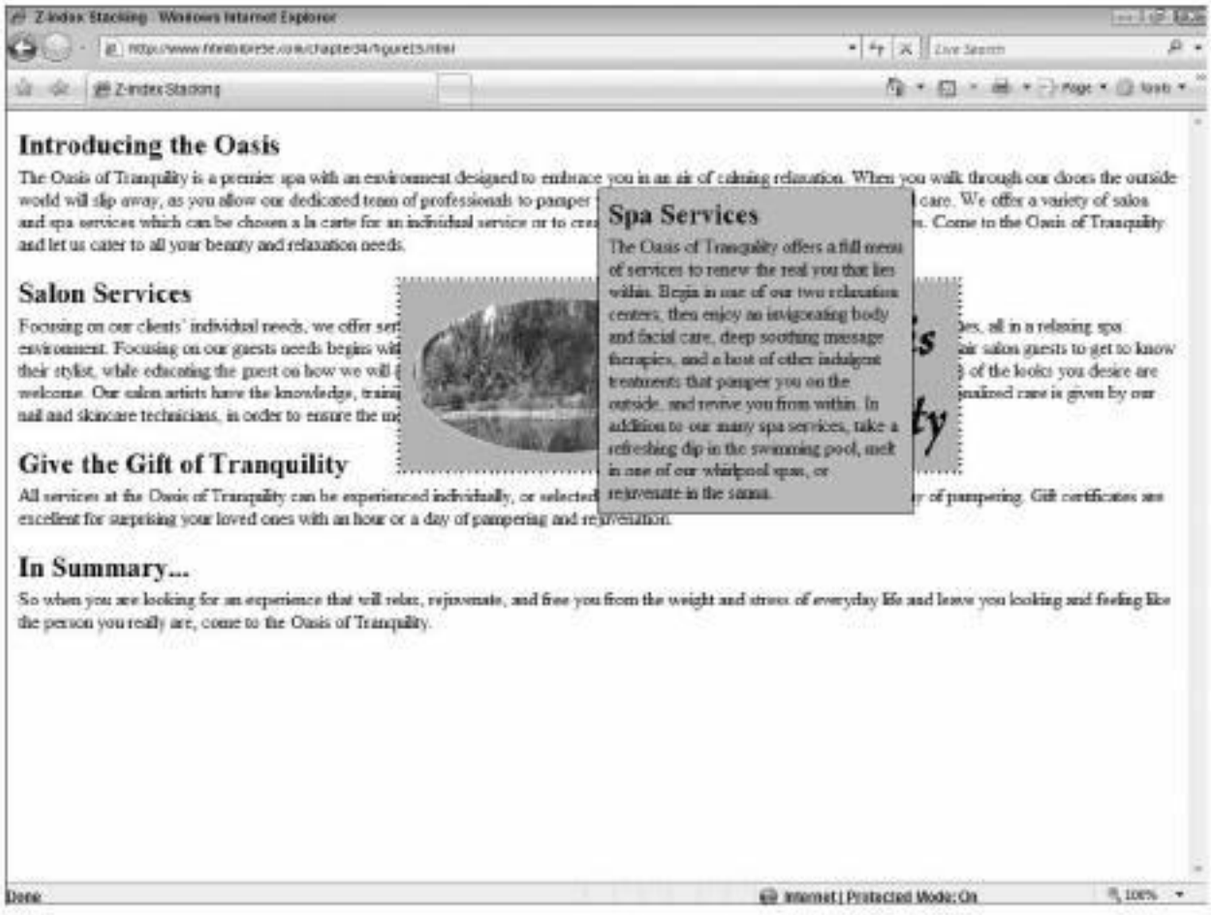
If you change the `z-index` of `spa` to `300`, it will then render over `logo`, as shown in Figure 34-15.

Tip

You can use many of the properties in this chapter for animation purposes. Using JavaScript, you can dynamically change an element's size, position, and/or index to animate it. For more information, see Chapters 16, 17, and 36. ■

FIGURE 34-15

Changing an element's z-index changes its position in the stack.



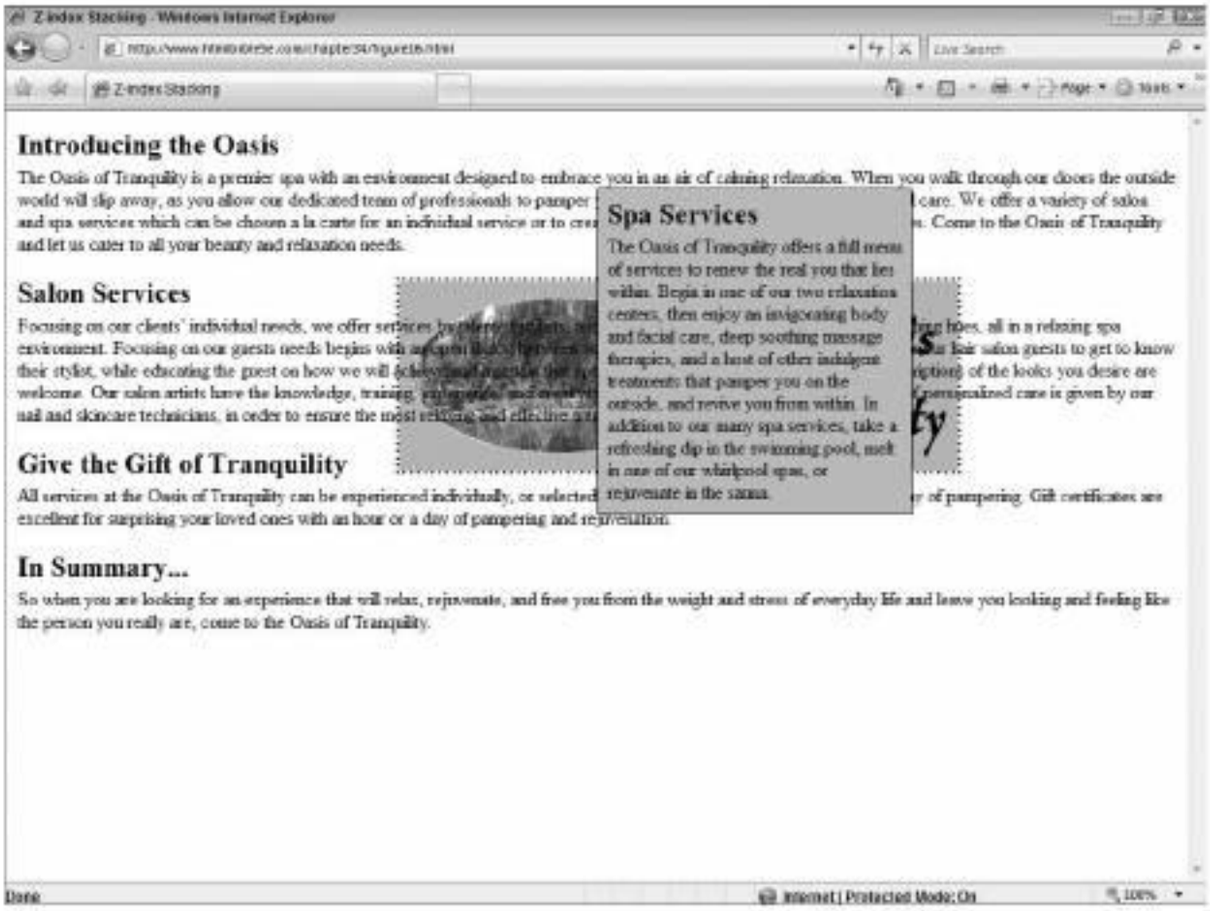
If you then change the `z-index` of `logo` to –200, it will render under the main body of the document (whose z-index is 0), as shown in Figure 34-16.

The effect shown in Figure 34-16 has several caveats:

- The user agent must support such layering.
- The user agent must support transparent backgrounds for the block to show through the document layer.
- The main document layer must not have a background that would otherwise obscure the visibility of the lower layers.

FIGURE 34-16

Items can have negative z-index values as well.



Controlling Element Visibility

You can use the `visibility` property to control whether an element is visible or not. The `visibility` property has the following format:

```
visibility: visible | hidden | collapse;
```

The `visible` and `hidden` values are fairly self-explanatory — set to `visible` (default), an element is rendered as normal; set to `hidden`, the element is still rendered but not displayed.

Note

Even though an element is hidden with `visibility`, set to `hidden` it will still affect the layout — that is, space for the element is still reserved in the layout. ■

The `collapse` value causes an element with rows or columns to collapse its borders. If the element does not have rows or columns, then this value is treated the same as `hidden`.

Cross-Ref

For more information on collapsing borders, see Chapter 30. ■

To truly hide an element, set its `display` property to `none`. An element styled this way will not be part of the render stream; it will be invisible and completely unknown within the user agent's rendering of the document's. However, the element will still be visible within the document's source.

Summary

This chapter covered the use of CSS to position elements, another one of the key concepts that really unlocks the power of CSS. If you master modifying the box model — padding, borders, and margins — and positioning, you have the power to create some truly amazing documents. The next few chapters wrap up the coverage of CSS with information on some of the more niche CSS topics.

CHAPTER

35

Pseudo-Elements and Generated Content

CSS works extremely well when you have concrete, single-state HTML elements to which to assign properties; but what happens when you want to assign certain properties to pieces of a document that aren't delimited by standard elements? In addition, there are times when it is convenient or necessary to automatically include generated content around elements. These out-of-bound cases are where CSS pseudo-elements come in handy.

This chapter introduces you to CSS pseudo-elements and generated content using CSS methods.

IN THIS CHAPTER

The Content Property

The CSS `content` property plays a key role in pseudo-elements, as it provides the actual content used by two pseudo-elements, `:before` and `:after`. The property itself is very simple and has the format

```
content: "<text>"
```

where "*<text>*" is the text that comprises the content. Note that the text must be plain text — no markup or other content needing parsing — and it must be enclosed in quotes. The text itself will inherit the attributes of its parent element.

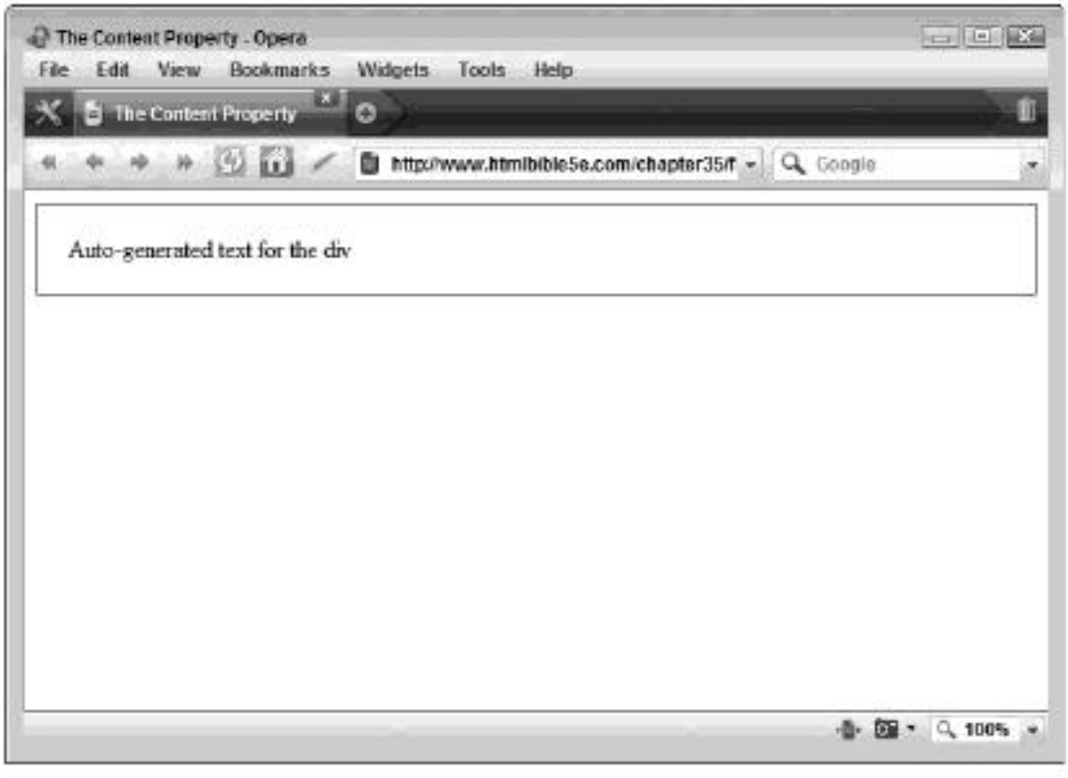
The next section examines the particulars of using the `content` property with the `:before` and `:after` pseudo-elements, but use of the `content`

property is not limited to those two elements. For example, the `content` property can be used to auto-generate content within any element. Consider the following code and the result shown in Figure 35-1 (the additional styles for the `div` are to enhance its visibility for the figure):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>The Content Property</title>
<style type="text/css">
div.text { padding: 20px;
border: thin solid black;
content: "Auto-generated text for the div";
}
</style>
</head>
<body>
<div class="text">
<p>This text will be replaced by the content property.</p>
</div>
</body>
</html>
```

FIGURE 35-1

The content property can supply auto-generated content for any textual element.



The `content` property can be used to insert content other than plain text, such as media delivered via the `url()` function. For example, the following property definition uses the `content` property to insert a `wav` sound effect:

```
content: url('media/sound.wav');
```

Note

Some applications of the `content` property are impractical and could be annoying to users. Still, it is worth noting the `content` property's capabilities for special applications. ■

It is important to note that the value of the `content` property replaces the contents of the `div`. This is true for any element that has a `content` property — whatever content is initially placed within the element will not be rendered in the user agent.

The `content` property can also be used to automate quotation marks by using a special `quotes` property to define the quote marks, and unique keywords to insert the marks. Quotations marks are covered later in this chapter.

Note

As of this writing, few browsers support using the `content` property or pseudo-elements. ■

Pseudo-Elements

CSS pseudo-elements enable you to assign properties to areas of a document that are not delimited by standard elements. Table 35-1 lists the available pseudo-elements and their scope — that is, the amount of a document they affect.

TABLE 35-1

Pseudo-Elements

Element	Scope
`:first-line`	The first line of an element's text — that is, text from the beginning of the element to the first line wrap, end of the paragraph, or line break.
`:first-letter`	The first letter of an element's text
`:before`	The space before an element. Commonly used to automatically place content before an element.
`:after`	The space after an element. Commonly used to automatically place content after an element.

Note that each element begins with a colon (:) because the identifiers are meant to be appended to the end of a selector, turning the elements that match that selector *into* pseudo-elements.

For example, consider the following two selectors, the first without and the second with an `:after` identifier:

```
/* selector (p) with no pseudo-element */
p  { color: white;
background-color: black;
padding: 10px;}
/* selector (p) with pseudo-element */
p:after  { color: white;
background-color: black;
padding: 10px;}
```

The pseudo-element identifier can be used with the most complex of selectors; simply append it to the last element of the selector, as in the following example:

```
div.sidebar p:after { ...}
```

Any paragraph that is a descendant of a `div`, with a class of `sidebar`, will have the `:after` pseudo-element applied.

Note

In CSS versions 1 and 2, both pseudo-elements and pseudo-classes began with a single colon (:). This will change in CSS3 — pseudo-elements will begin with two colons (::) to distinguish them from pseudo-classes. As of this writing, CSS3 is still under development, and adoption of its conventions is scarce. As such, we will stick with the CSS2 conventions. ■

The following sections detail the effects of the various pseudo-elements.

:first-line

The `:first-line` pseudo-element enables you to dynamically style the first line of an element. Of course, there are many things that can change the length of an element's first line, including a change in the size of the user agent's window, movement of elements within the document, or a subsequent change of the element's content.

If the element's text wrapping changes for any reason, the scope of the pseudo-element does as well. This effect can be seen in Figure 35-2, where two different-sized windows display the same paragraph using this style:

```
div:first-line  { text-decoration:  underline;}
```

:first-letter

The `:first-letter` pseudo-element enables you to style the first letter of an element. This ability is typically used for typographic effects such as drop caps, where the first letter of a section of text is printed in a larger or stylistic manner to offset it from the rest of the text. An example of this technique is provided with the following, whose output is shown in Figure 35-3:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

```
<html>
<head>
<title>Drop Cap Effects</title>
<style type="text/css">
div.dropcap p:first-letter {
padding: 5px;
border: 1pt solid black;
font-size: 300%;
line-height: .8em;
margin-right: 10px;
float: left;
}
</style>
</head>
<body>
<div class="dropcap">
<p>Welcome to The Oasis of Tranquility, your source of day
spa services at hair salon prices. Come visit us for that
deep tissue or relaxing massage, facial, manicure, or
hair coloring you have been putting off.<br />
<br />
Our concept is simple{\emdash}to provide luxurious service
affordable to most consumers. So stop in and let our experts
please and pamper you today.</p>
</div>
</body>
</html>
```

FIGURE 35-2

The :first-line pseudo-element can be used to style the first line of an element — the styling is dynamic and changes if and when the first line changes.

FIGURE 35-3

The :first-letter pseudo-element can create effective drop caps.

In the preceding example, a handful of additional properties help craft the drop-cap effect:

- The `padding` and `border` properties produce a thin border without crowding the letter.
- The `font-size` property sets the size of the letter to an appropriate size for a drop cap.
- The `line-height` property shortens the height of the box, giving it a more square profile.
- The `margin-right` property widens the space between the drop cap and the text to its right.
- The `float` property enables the drop cap to occupy the vertical space of several lines, instead of only the line on which it appears (the first line).

Tip

Combine the `:first-letter` and `:first-line` pseudo-elements to create a drop cap and first-line stylistic combo. ■

:before and :after

The `:before` and `:after` pseudo-elements enable you to specify text that will be automatically prepended or appended to an element. Both pseudo-elements have the same syntax; one simply controls the space before the element to which it is attached, and the other controls the space after the element to which it is attached.

For example, you could use a definition similar to the following to insert a colon after every `h1` element:

```
h1:after  {  content: ":";}
```

Likewise, you can use the `:before` pseudo-element to add content before an element. For example, you could add the word "Section" before all `h1` elements using a definition similar to the following:

```
h1:before  {  content: "Section";}
```

You can also use the `:before` and `:after` pseudo-elements to help automate quotation marks, as outlined in the next section.

Note

When using string values with the `content` property, be sure to enclose the string in quotes. If you need to include newlines in the text, use the `\A` placeholder. ■

Quotation Marks

You can use the auto-generation features of CSS to define and then automatically display quotation marks. First, you need to define the quotes, and then you can add them to appropriate elements.

The `quotes` property takes a list of arguments in string format to use for the open and close quotes at multiple levels. The various levels are designed to accommodate nested quotation marks. This property has the following form:

```
quotes: <open_first_level> <close_first_level>
<open_second_level> <close_second_level> $ $$\ldots$;
```

The standard definition for most English uses is as follows:

```
quotes: "" '' "" '';
```

This specifies a double-quote for the first level (open and closing) and a single-quote for the second level (open and closing). Note the use of the opposite quote type (single enclosing double and vice versa) within the definition.

Note

Most user agents do not support auto-generated content. ■

Once you define the quotes, you can use them along with the `:before` and `:after` pseudo-elements, as in the following example:

```
blockquote:before { content: open-quote;}
blockquote:after  { content: close-quote;}
```

The open-quote and close-quote words are shortcuts for the values stored in the quotes property. Technically, you can place just about anything in the content property, as it accepts string values. The next section shows how you can use the content property to create automatic counters.

Numbering Elements Automatically

One of the nicest features of using the content property with counters is the ability to automatically number elements. The advantage to using counters over standard lists is that counters are more flexible, enabling you to start at an arbitrary number, combine numbers (for example, 1.1), and so on.

Note
Most user agents do not support counters. ■

The counter object

A special object, counter, can be used to track a value, and can be incremented and reset by other style operations. The counter object has the following form when used with the content property:

```
content: counter(<char:Variable>counter_name)</char:Variable>);
```

This places the current value of the counter in the content object. For example, the following style definition will display "Chapter", the current value of the "chapter" counter, and a space at the beginning of each h1 element:

```
h1:before { content: "Chapter " counter(chapter) " ";}
```

Of course, it's of no use to always assign the same number to the :before pseudo-element. That's where the counter-increment and counter-reset objects come in, described in the next section.

Changing the counter value

The counter-increment property takes a counter object as an argument and increments its value by one. You can also increment the counter by other values by specifying the value to add to the counter. For example, to increment the chapter counter by 2, you would use this definition:

```
counter-increment: chapter 2;
```

Tip

You can increment several counters with the same property statement by specifying the additional counters after the first, separated by spaces. For example, the following definition will increment the chapter and section counters each by 2:

```
counter-increment: chapter 2 section 2; ■
```

You can also specify negative numbers to decrement the counter(s). For example, to decrement the `chapter` counter by 1, you can use the following:

```
counter-increment: chapter -1;
```

The other method for changing a counter's value is to use the `counter-reset` property. This property resets the counter to zero or, optionally, an arbitrary number specified with the property. The `counter-reset` property has the following format:

```
counter-reset: <char:Variable>{\it counter_name}</char:Variable>
[value];
```

For example, to reset the `chapter` counter to 1, you can use this definition:

```
counter-reset: chapter 1;
```

Tip

You can reset multiple counters with the same property by specifying all the counters on the same line, separated by spaces. ■

If a counter is used and incremented or reset in the same context (in the same definition), then the counter is first incremented or reset before being assigned to a property or otherwise used. This is important to keep in mind to ensure that the first use of the counter is the correct value.

A counter example: chapter and section numbers

Using counters, you can easily implement an auto-numbering scheme for chapters and sections. To implement this auto-numbering, use `h1` elements for chapter titles, and `h2` elements for sections. We will use two counters, chapter and section, respectively.

First, set up your chapter heading definition as follows:

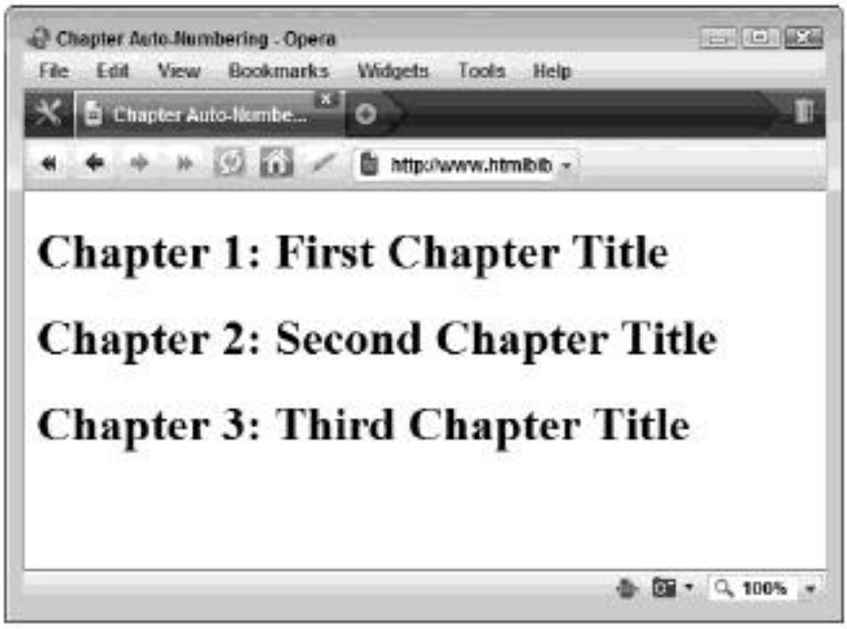
```
h1:before {content: "Chapter " counter(chapter) ": ";
           counter-increment: chapter;
           counter-reset: section;}
```

This definition will display "Chapter *chapter_num*:" before the contents of each `h1` element. The `chapter` counter is incremented and the `section` counter is reset — both of these actions take place prior to the counter and text being assigned to the `content` property. Therefore, the following text would result in the output shown in Figure 35-4:

```
<h1>First Chapter Title</h1>
<h1>Second Chapter Title</h1>
<h1>Third Chapter Title</h1>
```

FIGURE 35-4

Auto-numbering h1 elements



The next step is to set up the section numbering, which is similar to the chapter numbering but is applied to the `h2` elements (subheadings of `h1`):

```
h2:before {content: "Section " counter(chapter) "."
counter(section) ": ";
counter-increment: section;
h2 { text-indent: 20px;}
```

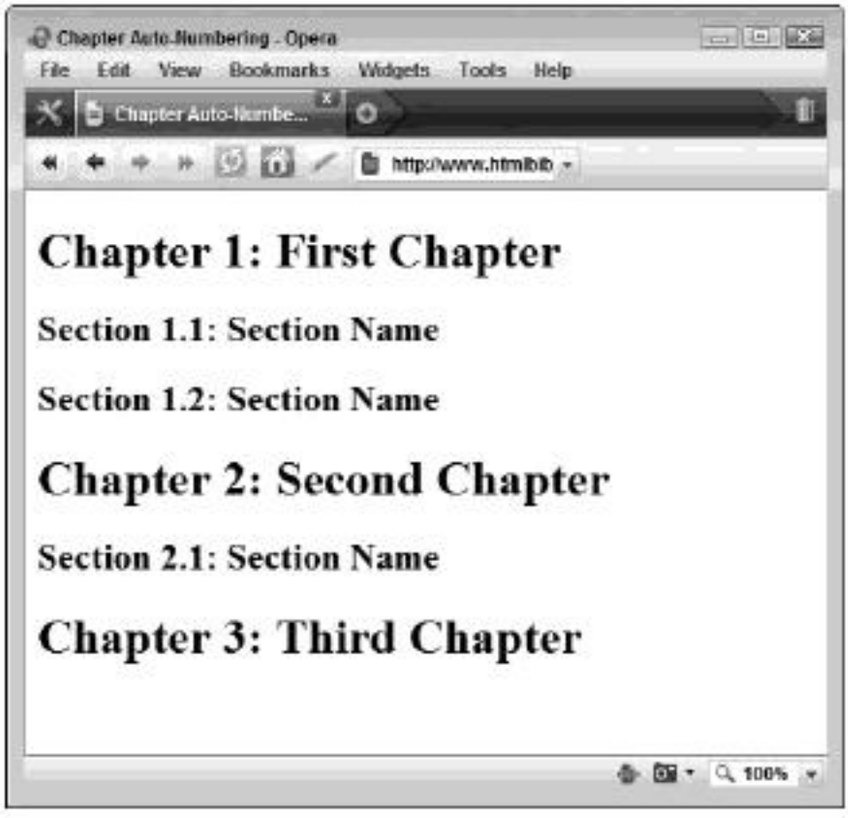
We also add `text-indent` to indent the subheads. Now the styles are complete. The final, following code results in the display shown in Figure 35-5:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
```

```
  <title>Chapter Auto-Number</title>
  <style type="text/css">
    h1:before {content: "Chapter " counter(chapter) ": ";
               counter-increment: chapter;
               counter-reset: section;}
    h2:before {content: "Section " counter(chapter) "."
               counter(section) ": ";
               counter-increment: section;}
    h2 { text-indent: 20px;}
  </style>
</head>
<body>
  <h1>First Chapter Title</h1>
    <h2>Section Name</h2>
    <h2>Section Name</h2>
  <h1>Second Chapter Title</h1>
    <h2>Section Name</h2>
  <h1>Third Chapter Title</h1>
</body>
</html>
```

FIGURE 35-5

The completed auto-numbering system numbers both chapters and sections.



Tip

The counters should automatically start with a value of 0. In this example, that is ideal. However, if you need to start the counters at another value, you can attach resets to a higher tag (such as `<body id="c35-body-0001">`), as in the following example:

```
body:before {counter-reset: chapter 12 section 10;} ■
```

Custom list numbers

You can use a similar construct for custom list numbering. For example, consider the following code, which starts numbering the list at 20:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>List Custom Numbering</title>
  <style type="text/css">
    li:before {content: counter(list) ": ";
               counter-increment: list;}
  </style>
</head>
<body>
  <ol style="counter-reset: list 19;
       list-style-type:none;">
    <li>First item</li>
    <li>Second item</li>
    <li>Third item</li>
  </ol>
</body>
</html>
```

Note that the `<ol>` tag resets the counter to 19 because of the way the `counter-increment` works (it causes the counter to increment before it is used). So you must set the counter one lower than the first occurrence.

Tip

You can have multiple instances of the same counter in your documents, and they can all operate independently. The key is to limit each counter's scope: A counter's effective scope is within the element that initialized the counter with the first reset. In the example of lists, it is the `<ol>` tag. If you nested another `<ol>` tag within the first, it could have its own instance of the `list` counter, and they could operate independently of each other. ■

Summary

This chapter covered using pseudo-elements and generating content using CSS. Pseudo-elements enable you to select pieces of the document that aren't bound by traditional element boundaries,

such as the first letter or first line of an element. Also introduced was the `content` property, which enables arbitrary pieces of text to be defined and then inserted via other CSS elements. Such a mechanism can automatically insert quotation marks or other desirable text. The last part of the chapter covered counters, which enable you to define and use an automatic numbering system, to auto-number elements (such as headers) or to custom number numbered lists. The next chapter expands on these concepts by introducing dynamic content.

CHAPTER

36

Dynamic HTML with CSS

CSS can be a powerful tool for creating well-formatted documents. This chapter describes how you can change a CSS property in various user agents to lend a dynamic nature to documents. Here, you'll learn how to access CSS properties and script them to perform tasks, such as change text colors. You'll see that every CSS property can be changed programmatically.

You'll also find that some browsers, most notably Internet Explorer, feature CSS-like syntax for creating dynamic filtered effects such as drop shadows and blurs.

IN THIS CHAPTER

Accessing CSS Properties with JavaScript

Mozilla and Internet Explorer (IE) browsers make CSS1 element properties accessible from JavaScript through their Document Object Model (DOM). However, the Mozilla DOM and Internet Explorer DOM are different. They both implement parts of the W3C CSS2 standards, but they consistently cover different areas, so CSS2 JavaScript code on one browser may not work on other browsers. Note that the Gecko layout engine covers all of the properties in W3C CSS2 standards.

Generally, CSS properties are all accessed the same way, via reading values as properties and setting values via methods. To access the CSS properties in script, you use the property name, unless there's a hyphen in the name. In the case of hyphenated property names, delete the hyphen and uppercase the next letter. The rest of the property name remains in lowercase. For example,

```
font-size
```

becomes

```
fontSize
```

The property name is then appended to the object name/id with a style collection. For example, to access the `font-size` property of an object named `bigText`, you can use the following statement:

```
bigText.style.fontSize
```

In turn, that statement can be used to assign a value to the object's property. For example, the following JavaScript statement sets the `font-size` property of the element with an id of `bigText` to `xx-large`:

```
bigText.style.fontSize = "xx-large";
```

Consider the code in the following document. When the paragraph in the document is clicked, the `onClick` handler runs the `SuperSizeMe( )` JavaScript function, which sets the paragraph's `font-size` property to `xx-large` (effectively supersizing the paragraph). Figure 36-1 shows the paragraph before it is clicked, and Figure 36-2 shows the paragraph after it is clicked.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Super Size Me</title>
  <style type="text/css">
    #bigText { font-size: medium;}
  </style>
  <script type="text/JavaScript">

    function SuperSizeMe(obj) {
      obj.style.fontSize = "xx-large";
}
  </script>
</head>
<body>
  <p id="bigText" onClick="SuperSizeMe(this);">Lorem ipsum dolor sit
  amet, consectetur adipisicing elit, sed do eiusmod tempor
  incididunt ut labore et dolore magna aliqua. Ut enim ad minim
  veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip
  ex ea commodo consequat. Duis aute irure dolor in reprehenderit
  in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
  Excepteur sint occaecat cupidatat non proident, sunt in culpa qui
  officia deserunt mollit anim id est laborum.</p>
</body>
</html>
```

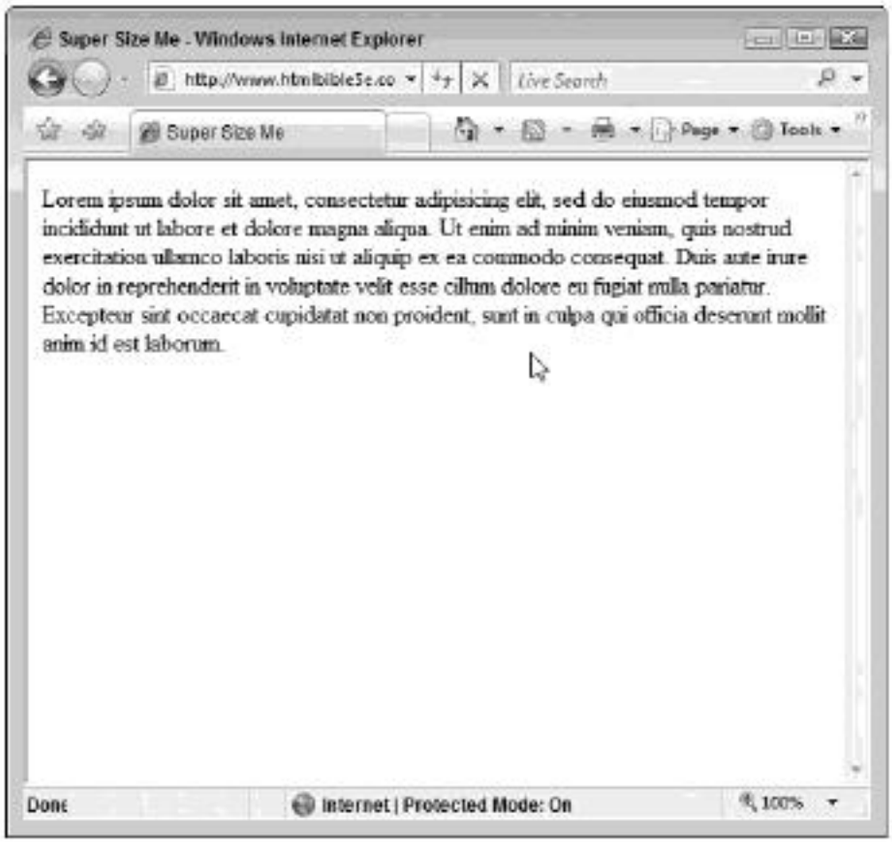
It is important to take a moment and examine what the code actually is doing. You might think that the script accessing the style collection can therefore access the styles assigned to the element no matter the source of said styles. However, that is not the case — the style collection can

only reference a local style embedded in the object's tag via the `script` attribute. As such, the following JavaScript statement would display a null value if run immediately after the preceding document was loaded into the user agent:

```
alert(document.getElementById('bigText').style.fontSize);
```

FIGURE 36-1

Before the text is clicked, it has a medium-size font.



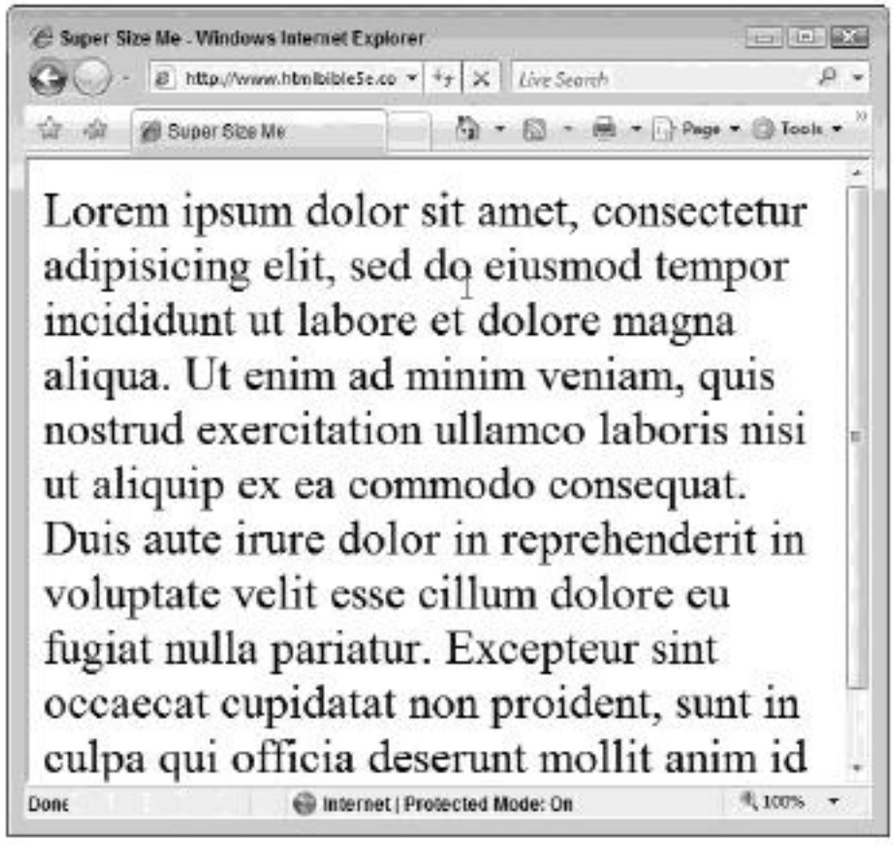
The `style.fontSize` property would be null because the `bigText` element does not contain a `style` attribute. How, then, does the preceding example work if it doesn't change the `medium` value of `font-size` set in the `<style>` section? The answer is that it doesn't need to change the values in the `<style>` section; it simply sets values in the element's `style` *attribute*, which has precedence over the styles in the `<style>` section.

Of course, if there were a value for an element's `style` attribute, the style collection could be used to determine it.

If you want to read the properties set in the `<style>` section, you must use one of two methods — one for IE browsers and another for Mozilla browsers. IE has an object property named `currentStyle`, while Mozilla browsers have a `window` objects property — specifically, `window.getComputedStyle`.

FIGURE 36-2

After the text is clicked, it now has an xx-large font.



Use of the IE property is straightforward: Find the object via its `id` attribute and then use the property to return the style property's value, as in the following code.

```
obj = document.getElementById(id);
value = obj.currentStyle['fontSize'];
```

Note that the style property is given in the "omit hyphen" style — that is, `fontSize` instead of `font-size`.

The Mozilla method has an extra step because its property returns a collection that must be parsed for the desired property. The parsing is done via the `getPropertyValue( )`method, as shown here:

```
obj = document.getElementById(id);
objstyles = window.getComputedStyle(obj,null);
value = objstyles.getPropertyValue('font-size');
```

Notice that the Mozilla method uses the standard CSS name for properties, not the "omit hyphen" name. In either case, at the end of the code, the variable `value` would hold the value of the `font-size` property.

You could combine these methods into a single function by adding a bit of browser detection. The following listing shows a sample function that, given an object id and the "omit hyphen" and normal CSS property name, will return the value of the property:

```
// Return the value of CSS propName for element
//   with given id
function getStyleVal (id, propName) {
  // Can we do this at all? (getElementById available)
  if (obj = document.getElementById(id)) {
    // Is currentStyle available (IE)
    if (obj.currentStyle) {
      // Convert property name to IE format
      if (propName.indexOf("-") != -1) {
        hyp = propName.indexOf("-");
        propName = propName.substr(0,hyp) +
                   propName.charAt(hyp+1).toUpperCase() +
                   propName.substr(hyp+2);
}
      return obj.currentStyle[propName];
}
    // Is getComputedStyle available (Mozilla)
    if (window.getComputedStyle) {
        compStyle = window.getComputedStyle(obj,null);
        return compStyle.getPropertyValue(propName);
}
}   // End If obj=document.getElementById
  // Else return a blank string
  return "";
}
```

Note that the function checks whether the browser supports `document.getElementById` (it will if it is a modern browser) before doing anything. It then determines whether the IE or Mozilla method is available and acts accordingly to return the property value. Along the way, it also converts the property name, if necessary, to deal with IE's preferred format. Figures 36-3 and 36-4 show the function running in IE and Firefox, respectively, on a document that contains the following code:

```
  <style type="text/css">
    #bigText { font-size: medium;}
  </style>
...
<p id="bigText"
onClick="alert(getStyleVal('bigText','font-
size'));">Lorem ipsum dolor sit amet, consectetur
adipisicing elit, ...
```

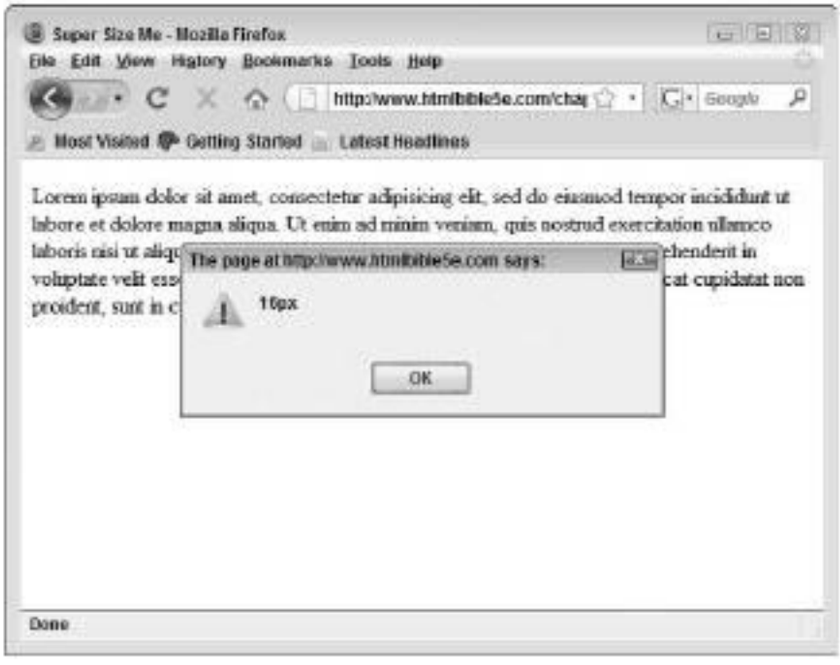
When the paragraph text is clicked, a JavaScript alert box pops up and shows the value of the paragraph's `font-size` property.

FIGURE 36-3

The getStyleVal function is used to show the value of a paragraph's font-size property in Internet Explorer.

FIGURE 36-4

The getStyleVal function is used to show the value of a paragraph's font-size property in Firefox.



Note

Notice that the `getStyleVal` function returns an absolute font size, in points, when run in Firefox. Because of the nature of the functions to return the values, they may vary in format from the values used to set properties in the actual styles. For example, you might set a color to orange via a value like `#FFA500`, while the JavaScript function returns `orange`. Or, as in the example in the previous screen shots, the absolute font size is returned instead of the relative setting of `medium`. ■

Why, then, can't you use these two methods to also manipulate styles? Because these methods are read-only, you can retrieve the value of style properties using these methods, but you can't set properties with them. You can set the value of style properties directly, as shown in the beginning of this section.

Useful CSS Manipulation

Earlier examples showed how you can manipulate the font properties of elements in a document. Although such manipulations can be helpful, more involved manipulations of CSS using JavaScript can be even more useful. This section shows a few examples to get you started.

Hiding and showing text

With CSS and JavaScript, it is fairly trivial to alternately show and hide text. This can be used for a variety of purposes, such as drop-down menus or hiding text until a user decides to reveal it. For example, consider a list of questions and answers. A user of the site might not want to see the entire list of answers, but may want to selectively reveal a few. Using the CSS `display` property, you can write a script to enable this behavior.

Consider the following code, which hides the answer until any portion of the question is clicked. If the question is clicked a second time, the answer is again hidden. Figure 36-5 shows the answer in its hidden state, and Figure 36-6 shows the answer revealed after the "Q" is clicked.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Hide and Seek Text</title>
  <style type="text/css">
    /* Initially hide all the questions */
    .hidenseek { display: none;}
    /* Question and answer display styles */
    .Q { font-size: xx-large;
         padding-bottom: 0;
         margin-bottom: 0;
         cursor: pointer;}
    .Qtext { margin-left: 20px;
             margin-top: 0;
             padding-top: 0;|
```

```
    .A { font-size: xx-large;
         padding-bottom: 0;
         margin-bottom: 0;
         clear: left;}
    .Atext { margin-left: 20px;
             margin-top: 0;
             padding-top: 0;}
  </style>
  <script type="text/JavaScript">
    // Alternately reveal or hide the element
    function hidenseek(id) {
      obj = document.getElementById(id);
      // If the style is blank, it hasn't been set yet
      //   and we can assume the element is hidden
      if ((obj.style.display == "") ||
          (obj.style.display == "none")) {
        obj.style.display = "block";
}       else {
        obj.style.display = "none";
}
}
  </script>
</head>
<body>
<div onClick="hidenseek('A1')"><p class="Q">Q:</p>
<p class="Qtext">What kind of equipment and peripherals do I need to
bring to the LAN event?</p>
</div>
<div id="A1" class="hidenseek">
<p class="A">A:</p>
<p class="Atext">You will need to bring the following (minimum)
items: your computer (minumum P4 with 1GB of RAM and a high-end
video card), a monitor (no larger than 19"), keyboard, mouse, a
surge protector, and a CAT-5 network cable at least 5' long. Do not
bring anything that is not listed herein, or attempt to bring and
use more than one computer.</p>
</div>
</body>
</html>
```

This example operates by using a JavaScript function that reads the value of an object's `display` property and sets the property to the opposite of its current value — revealing a hidden element or hiding a visible one. The function is called via an `onClick` handler attached to the `div` element that contains the question. The function call includes the `id` of the matching question so the function knows what element to act upon (in this case "A1").

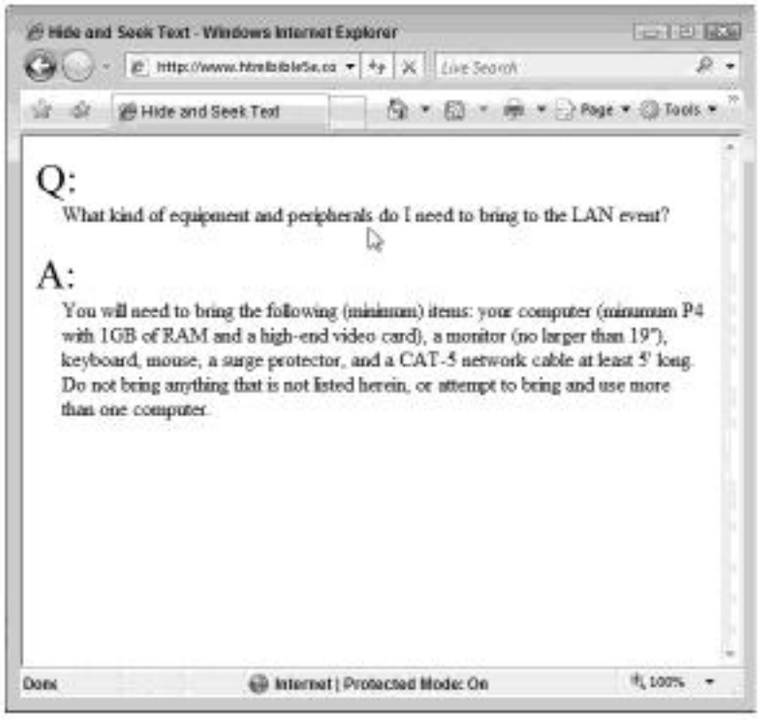
As previously mentioned, you can use this technique for a variety of purposes. You simply hide the elements you wish to start hidden (using a value of `none` for the elements' display property) and use a function call — tied to buttons or other events — to show and optionally toggle the visibility of those elements.

FIGURE 36-5

The answer to the question starts out in a hidden state.

FIGURE 36-6

If the Q (or any part of the question text) is clicked, the answer is revealed. Clicking again hides the answer.



Picture zooming

Another common use for such CSS effects is to zoom images from thumbnails to their full size. This technique is used often in picture galleries and on other pages where showing a full-size image is desirable but prohibitive.

The technique is similar to the techniques already shown in this chapter — a JavaScript event triggers a script to change CSS properties of certain elements. In this case, the script changes the `display` property of an image thumbnail and a full-size image. A sample document with such a script is shown here:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Picture Zoom</title>
  <style type="text/css">
    .zoom | display: none;
            float: left;|
    .thumb | display: block;
             float: left;|
    .text  | float: left;
             padding-left: 20px;|
  </style>
  <script type="text/JavaScript">
    function PicZoom(id) |
      pic = document.getElementById(id);
      thum = document.getElementById("T"+id);
      if ((pic.style.display == "") ||
        (pic.style.display == "none")) {
        pic.style.display = "block";
        thum.style.display = "none";
}       else |
        pic.style.display = "none";
        thum.style.display = "block";
}
}
  </script>
</head>
<body>
  <div id="1" class="zoom"><p><img src="ashley.jpg"
   width="428" height="568" alt="Ashley"
   onMouseOut="PicZoom(1);" /></p></div>
  <div id="T1" class="thumb"><p><img src="ashley-thumb.jpg"
   width="171" height="227" alt="Ashley"
   onMouseOver="PicZoom(1);" /></p></div>
  <div class="text" style="float:left;"><p>For the most wild, yet
   most homely narrative which I am about to pen, I neither expect
   nor solicit belief. Mad indeed would I be to expect it, in a case
   where my very senses reject their own evidence. Yet, mad am I
   not --and very surely do I not dream. But to-morrow I die, and
```

```
        to-day I would unburden my soul. My immediate purpose is to place
        before the world, plainly, succinctly, and without comment, a
        series of mere household events. In their consequences, these
        events have terrified --have tortured --have destroyed me. Yet I
        will not attempt to expound them. To me, they have presented
        little but Horror --to many they will seem less terrible than
        baroques. Hereafter, perhaps, some intellect may be found which
        will reduce my phantasm to the common-place --some intellect
        more calm, more logical, and far less excitable than my own,
        which will perceive, in the circumstances I detail with awe,
        nothing more than an ordinary succession of very natural causes
        and effects.
    </p></div>
    </body>
    </html>
```

The images, thumbnail and full-size, are embedded in `div` elements for formatting and flexibility purposes. The thumbnail `div` initially has its `display` property set to `block` so it is visible. Conversely, the full-size image `div` initially has its `display` property set to `none` so it is not visible. The thumbnail `<img>` tag includes an `onMouseOver` event to call the `PicZoom( )`script when the mouse is placed over the image. The script reverses the `display` property of the thumbnail and full-size image `div` elements, making the full-size image visible and hiding the thumbnail. The mouse remains in position, now over the full-size image. When the mouse is moved off the image, an `onMouseOut` event in the full-size image tag calls the script again, setting the `display` properties back to their original settings to hide the full-size image and reveal the thumbnail.

This action can be seen in Figure 36-7, which shows the document in its beginning state, and in Figure 36-8, which shows the full-size image revealed, and the thumbnail hidden.

Note

As with most examples in this chapter, many methods can be used to create the image zoom effect. Some methods place both images on one large image and move the image around a "frame" to display one or the other, as necessary. Another method is to change the `src` attribute of the `<img>` tag so it displays the desired image. There are also complex, layered solutions, changing the `z-index` value of the concerned elements. Don't be afraid to play with your own solutions as well. ■

Menu buttons with rollovers

All the previous examples in this chapter use JavaScript to manipulate CSS to achieve various effects. So far, we have neglected the CSS anchor pseudo-classes, which can achieve similar effects.

The CSS anchor pseudo-classes are listed in Table 36-1.

The pseudo-classes are typically used to style elements as dynamic anchor elements. Using `:hover`, for example, you can dynamically change an element as the mouse passes over it — just like normal HTML anchors do.

For instance, the following document uses `:hover` to change the styling of anchors in table cells as the mouse passes over them. The effect, shown in Figure 36-9, is similar to dynamically styled menus driven by JavaScript.

FIGURE 36-7

The document begins with the thumbnail visible and the full-size image hidden.


```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Pseudo Class Menus</title>

  <style type="text/css">
    .nav tr td { border: 1pt solid black;}
    .menu { color: black;
            background-color: white;
            text-transform: none;
            text-decoration: none;}
    .menu:hover { color: white;
                  background-color: black;
                  text-transform: uppercase;
                  text-decoration: none;}
    .menucase { width: 100px;}
  </style>
</head>
<body>
<div class="menucase">
<table border="0" width=100%" class="nav">
```

```
    <tr><td><a class="menu" href="home.html">Home</a></td></tr>
    <tr><td><a class="menu" href="products.html">Products</a></td></tr>
    <tr><td><a class="menu" href="services.html">Services</a></td></tr>
    <tr><td><a class="menu" href="support.html">Support</a></td></tr>
    <tr><td><a class="menu" href="about.html">About</a></td></tr>
</table>
</div>
</body>
</html>
```

FIGURE 36-8

When the mouse is placed over the thumbnail image, the full-size image is revealed.

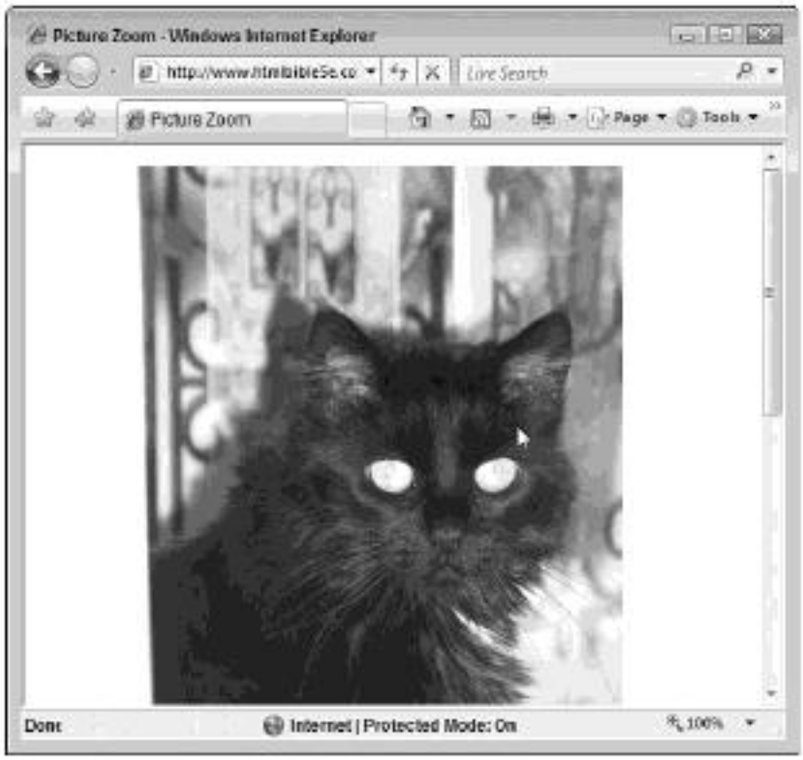


TABLE 36-1

CSS Anchor Pseudo-Classes

Pseudo-Class	Use/Effect
:link	Formats the element(s) matched in the attached selector as unvisited links
:visited	Formats the element(s) matched in the attached selector as visited links
:hover	Formats the element(s) matched in the attached selector when the mouse hovers over them
:active	Formats the element(s) matched in the attached selector as active links

FIGURE 36-9

The pseudo-classes can be used to create dynamic menus as you move the mouse over the menu items they highlight.

Note that although the technique of using pseudo-classes for such effects is very popular, it does disregard the best practice of separating behavior and presentation. This example is shown for completeness, but it is generally better to go the JavaScript route than CSS pseudo-classes for this type of effect.

Tip

Keep in mind that this technique can be used in conjunction with most elements. However, only the anchor tag can be formatted with the anchor pseudo-classes — use other elements to format the anchor tags within the document accordingly. ■

Summary

This chapter covered dynamic content and CSS. You learned how to use JavaScript to modify an element's underlying CSS for a variety of effects and how CSS pseudo-classes can be used to achieve dynamic effects. The next chapter covers how to use CSS to define pages for printing before we venture into niche CSS topics in the last part of the book (Chapters 38 through 41).