CHAPTER 15

Dynamic Effects: Rollovers, Positioning, and Animation

In this chapter we explore the use of JavaScript to add flash and sizzle to Web pages. Starting first with the basic rollover script that changes an image when the mouse hovers over it, we then proceed to more advanced techniques, including target-based and Cascading Style Sheets (CSS)-based rollovers. The manipulation of CSS-positioned regions is also discussed, with attention given to visibility and positioning issues. Finally, we describe how to create basic animation effects by using timers to move and change positioned objects and text. An emphasis is placed on making all introduced effects as cross-browser compliant as possible. The focus is on fundamental techniques you can use to create dynamic pages rather than on demonstrating all that is possible.

Images

The **images[]** collection of the **Document** object was introduced in Netscape 3 and Internet Explorer 4 and has since been adopted by nearly every browser in existence. This collection is a part of the DOM Level 1 standard, so support for it will continue well into the future. The collection contains **Image** objects (known as **HTMLImageElement**s in the DOM1 spec) corresponding to all the **<img>** tags in the document. Like all collections, images can be referenced numerically (**document.images[*i*]**), associatively (**document.images['*imagename*']**), and directly (**document.images.*imagename***).

Image Objects

The properties of the **Image** object correspond, as expected, to the attributes of the **<img>** tag as defined by the (X)HTML standard. An overview of the properties of the **Image** object beyond the common **id**, **className**, **style**, **title**, and DOM1 Core properties is presented in Table 15-1.

The traditional **Image** object also supports **onabort**, **onerror**, and **onload** event handlers. The **onabort** handler is invoked when the user aborts the loading of the image, usually by hitting the browser's Stop button. The **onerror** handler is fired when an error occurs during image loading. The **onload** handler is, of course, fired once the image has loaded.

Property	Description
align	Indicates the alignment of the image, usually "left" or "right."
alt	The alternative text rendering for the image as set by the **alt** attribute.
border	The width of the border around the image in pixels.
complete	Non-standard (but well-supported) Boolean indicating whether the image has completed loading.
height	The height of the image in pixels or as a percentage value.
hspace	The horizontal space around the image in pixels.
isMap	Boolean value indicating presence of the **ismap** attribute, which indicates the image is a server-side image map. The **useMap** property is used more often today.
longDesc	The value of the (X)HTML **longdesc** attribute, which provides a more verbose description for the image than the **alt** attribute.
lowSrc	The URL of the "low source" image as set by the **lowsrc** attribute. Under early browsers this is specified by the **lowsrc** property.
name	The value of the **name** attribute for the image.
src	The URL of the image.
useMap	The URL of the client-side image map if the **<img>** tag has a **usemap** attribute.
vspace	The vertical space in pixels around the image.
width	The width of the image in pixels or as a percentage value.

TABLE 15-1 Properties of **Image** Objects

Under modern browser implementations that support (X)HTML properly, you will also find **onmouseover**, **onmouseout**, **onclick**, and the rest of the core events supported for **Image**.

The following example illustrates simple access to the common properties of **Image**. A rendering of the example is shown in Figure 15-1.

L 15-1

```
<!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>JavaScript Image Object Test</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
</head>
<body>
<img src="sample.gif" width="200" height="100"
     name="image1" id="image1" align="left"
     alt="Test Image" border="0" />
<br clear="all" />
<hr />
<br clear="all" />
<h1>Image Properties</h1>
<form name="imageForm" id="imageForm" action="#" method="get">
Left:
<input type="radio" name="align" id="alignleft" value="left" checked="checked"
```

```
onchange="document.images.image1.align=this.value" />
Right:
<input type="radio" name="align" id="alignright" value="right"
onchange="document.images.image1.align=this.value" />
<br />
Alt:
<input type="text" name="alt" id="alt"
onchange="document.images.image1.alt=this.value" />
<br />

Border:
<input type="text" name="border" id="border"
onchange="document.images.image1.border=this.value" />
<br />

Complete:
<input type="text" name="complete" id="complete" />
<br />

Height:
<input type="text" name="height" id="height"
onchange="document.images.image1.height=this.value" />
<br />

Hspace:
<input type="text" name="hspace" id="hspace"
onchange="document.images.image1.hspace=this.value" />
<br />

Name:
<input type="text" name="name" id="name" />
<br />

Src:
<input type="text" name="src" id="src" size="40"
onchange="document.images.image1.src=this.value" />
<br />

Vspace:
<input type="text" name="vspace" id="vspace"
onchange="document.images.image1.vspace=this.value" />
<br />

Width:
<input type="text" name="width" id="width"
onchange="document.images.image1.width=this.value" />
</form>

<script type="text/javascript">
<!--
function populateForm()
{
  if (document.images && document.images.image1 &&
      document.images.image1.complete)
```

```
    {
     with (document.imageForm)
     {
       var i = document.images.image1;
       alt.value = i.alt;
       border.value = i.border;
       complete.value = i.complete;
       height.value = i.height;
               hspace.value = i.hspace;
       name.value = i.name;
       src.value = i.src;
       vspace.value = i.vspace;
       width.value = i.width;
     }
   }
}
window.onload = populateForm;
//-->
</script>
</body>
</html>
```

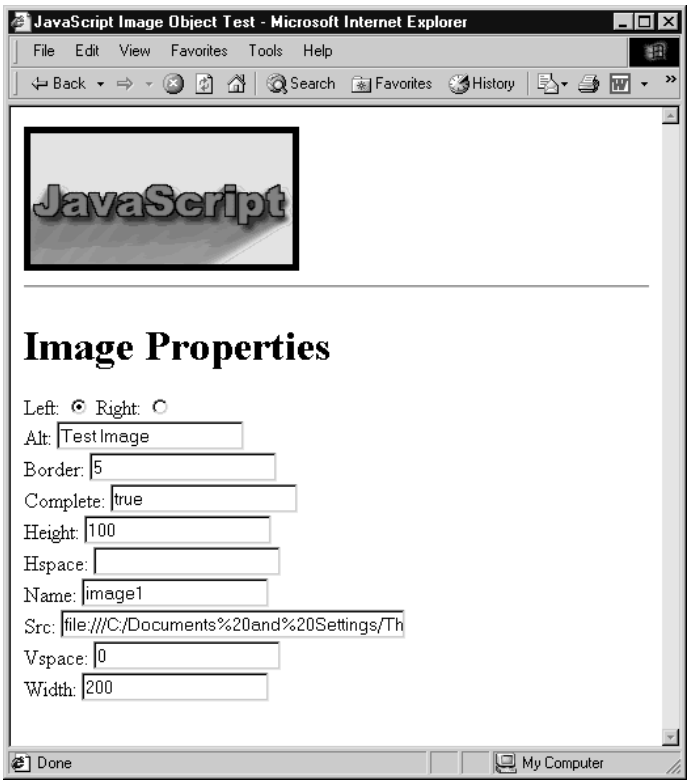


FIGURE 15-1 Manipulating **Image** properties with JavaScript

NOTE *If you try this example under much older browsers such as Netscape 3, you will find that it is not possible to manipulate the properties of the* ***Image*** *object, except for the* ***src*** *attribute.*

Notice in the previous example how it is possible to manipulate the image **src** dynamically. This leads to the first application of the **Image** object—the ubiquitous rollover button.

Rollover Buttons

One of the most common JavaScript page embellishments is the inclusion of rollover buttons. A *rollover button* is a button that changes when the user positions the mouse over it or some other event occurs on it. For example, in addition to changing when the user moves their mouse over it, it can change when it is clicked.

To create a basic rollover button, you first will need two, perhaps even three images, to represent each of the button's states—*inactive*, *active*, and *unavailable*. The first two states are for when the mouse is and is not over the button; the last is an optional state in case you wish to show the button inoperable (e.g., greyed out). A simple pair of images for a rollover button is shown here:

Ill 15-1

The idea is to include the image in the page as normal with an **<img>** tag referencing the image in its inactive state. When the mouse passes over the image, switch the image's **src** to the image representing its active state. When the mouse leaves, switch back to the original image.

Given the following image:

L 15-2
```
<img src="imageoff.gif" name="myimage" id="myimage" />
```

a reasonable implementation of a rollover might be:

L 15-3
```
<img src="imageoff.gif" name="myimage" id="myimage"
     onmouseover="document.myimage.src='imageon.gif';"
     onmouseout="document.myimage.src='imageoff.gif';" />
```

Of course, you could even shorten the example since you do not need to reference the object path but instead use the keyword **this,** as shown here:

L 15-4
```
<img src="imageoff.gif"
     onmouseover="this.src='imageon.gif';"
     onmouseout="this.src='imageoff.gif';" />
```

Rollover Limitations

The previous rollover example works in most modern browsers, but under older browsers like Netscape 4 you cannot capture **mouseover** events on an image in this way, and in very old browsers like Netscape 3 you can't capture them at all. Furthermore, we may find problems

with the script not addressing whether or not the images for the rollover effects have been downloaded by the browser or not. As a gentle introduction to cross-browser problems that emerge as we pursue dynamic effects, we address how to deal with these and other problems.

Event Binding Problems

The first problem we run into with rollovers across browsers and browser versions is that event binding is not supported on the <**img**> tag in many old implementations of JavaScript. So if you want to be backward compatible to Netscape 3 and 4, you can solve the problem by recalling that an image can be surrounded by a link, and links in Netscape 3 and 4 receive **onmouseover** events. So it is therefore possible to use the link's event handlers for control purposes. The following short example illustrates this technique, assuming you had two images called "imageon.gif" and "imageoff.gif."

L 15-5

```
<!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>Quick and Dirty Rollovers</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script type="text/javascript">
<!--
function mouseOn()
{
  document.image1.src = "imageon.gif";
}
function mouseOff()
{
  document.image1.src = "imageoff.gif";
}
//-->
</script>
</head>
<body>
<a href="#" onmouseover="mouseOn()" onmouseout="mouseOff()"><img
   name="image1" id="image1" src="imageoff.gif" border="0"
   width="90" height="90" alt="rollover" /></a>
</body>
</html>
```

Lack of Image Object Support

You will find that the previous example doesn't work in some older JavaScript-enabled browsers, such as Internet Explorer 3 and Netscape 2. In these browsers, images aren't scriptable, and they therefore don't support the **images[]** collection. Thus, regardless of support, we should err on the safe side and try to detect for JavaScript support before trying to modify an image.

The easiest way to make sure the user is running a browser that supports scriptable images is to check for the presence of the **document.images[]** collection:

L 15-6

```
if (document.images)
{
  // do image related code.
}
```

This statement determines whether or not the **document.images** exists. If the object does not exist, **document.images** is **undefined**, so the conditional evaluates to **false**. On the other hand, if the array exists, it is an object and thus evaluates to **true** in a conditional statement. We'll add this check into the next example, which addresses a problem that transcends browser version.

Preloading Images

What will happen if the user starts triggering rollovers when the rollover images haven't been downloaded? Unfortunately, the answer is a broken image will be shown. To combat this we use *JavaScript preloading* to force the browser to download an image (or other object) before it is actually needed and put in cache for later use.

The easiest way to preload an image is, in the **<head>** of the document, to create a new **Image** object and set its source to the image to preload. This forces the browser to begin fetching the image right away. Unless we have deferred the script execution the image must be downloaded before the script continues and thus preloading is ensured. To create an **Image** object, use the object constructor **new**:

L 15-7
```
var myImage = new Image();
```

You can pass in the width and height to the constructor if you wish, but in practice, it doesn't make much difference if the goal is preloading:

L 15-8
```
var myImage = new Image(width, height);
```

Once the object is created, set the **src** property so that the browser downloads it:

L 15-9
```
myImage.src = "URL of image";
```

Consider the following improved example of our image rollovers:

L 15-10
```
<!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>Rollover Example with Preloading</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script type="text/javascript">
<!--
if (document.images)
{  // Preload images
  var offImage = new Image(); // For the inactive image
  offImage.src = "imageoff.gif";
  var onImage = new Image();  // For the active image
  onImage.src = "imageon.gif";
}

function mouseOn()
{
  if (document.images)
    document.images.image1.src = onImage.src;
}

function mouseOff()
```

```
{
  if (document.images)
    document.images.image1.src = offImage.src;
}
//-->
</script>
</head>
<body>
<a href="http://www.pint.com" onmouseover="mouseOn()"
   onmouseout="mouseOff()"><img src="imageoff.gif" name="image1" id="image1"
border="0" width="90" height="90" alt="" /></a>
</body>
</html>
```

This example is closer to what we need. One remaining problem, however, is that the image names are hardcoded into the script, so it will require significant customization should you wish to reuse it (or even if you wish to add more rollover images to the page). We address that next.

Generalizing Rollover Code

One way to generalize this code to make it more reusable is to develop a consistent naming convention for images, and write JavaScript that assumes this convention. You could, for example, always use the words "on" and "off" as suffixes to each image name indicating the state the image is intended for. You could then automatically compute what image is needed through simple evaluation of the name and the appropriate suffix. This is best illustrated in an example:

L 15-11

```
<script type="text/javascript">
<!--

function preloadImage(url)
{
  var i = new Image();
  i.src = url;
  return i;
}



if (document.images)

{  // Preload images
  var homeon = preloadImage("homeon.gif");
  var homeoff = preloadImage("homeoff.gif");
  var productson = preloadImage("productson.gif");
  var productsoff = preloadImage("productsoff.gif");
}

// On input "myimage" this function sets the src of the image with
// this name to the value of myimageon.src

function mouseOn(imgName)
```

```
{
  if (document.images)
    document[imgName].src = eval(imgName + "on.src");
}

// On input "myimage" this function sets the src of the image with
// this name to the value of myimageoff.src

function mouseOff(imgName)
{
  if (document.images)
    document[imgName].src = eval(imgName + "off.src");
}
//-->
</script>
```

Notice how we generalized not only the image swapping function, but also the preloading functionality.

Later on, somewhere in our HTML file we would have appropriately named the images and links with **onmouseover** and **onmouseout** handlers to trigger the appropriate parts of the script:

L 15-12

```
<a href="home.html" onmouseover="mouseOn('home');"
   onmouseout="mouseOff('home');"><img src="homeoff.gif" height="50"
   width="100" name="home" id="home" border="0" alt="Home" /></a>
<br />

<a href="products.html" onmouseover="mouseOn('products');"
   onmouseout="mouseOff('products');"><img src="productsoff.gif" height="50"
   width="100" name="products" id="products" border="0" alt="Products" /></a>
```

The complete working example is shown here:

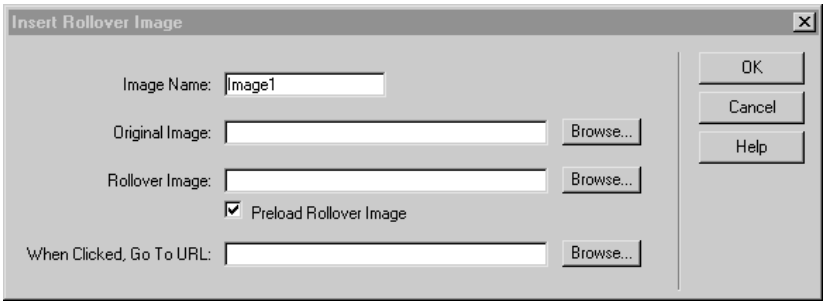
L 15-13

```
<!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>Rollover Example with Preloading</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script type="text/javascript">
<!--
function preloadImage(url)
{
  var i = new Image();
  i.src = url;
  return i;
}

if (document.images)
{  // Preload images
  var homeon = preloadImage("homeon.gif");
  var homeoff = preloadImage("homeoff.gif");
  var productson = preloadImage("productson.gif");
  var productsoff = preloadImage("productsoff.gif");}
```

```
// On input "myimage" this function sets the src of the image with
// this name to the value of myimageon.src
function mouseOn(imgName)
{
  if (document.images)
    document[imgName].src = eval(imgName + "on.src");
}

// On input "myimage" this function sets the src of the image with
// this name to the value of myimageoff.src
function mouseOff(imgName)
{
  if (document.images)
    document[imgName].src = eval(imgName + "off.src");
}
//-->
</script>
</head>
<body>

...Page content here...
<br />

<a href="home.html" onmouseover="mouseOn('home');"
   onmouseout="mouseOff('home');"><img src="homeoff.gif" height="50"
   width="100" name="home" id="home" border="0" alt="Home" /></a>

<br />

<a href="products.html" onmouseover="mouseOn('products');"
   onmouseout="mouseOff('products');"><img src="productsoff.gif" height="50"
   width="100" name="products" id="products" border="0" alt="Products" /></a>

</body>
</html>
```

Given the script shown, rollovers are limited only by one's capability to copy-paste and keep names correct. Rollovers have become so commonplace that most WYSIWYG HTML editors can insert rollover code directly. Notice the dialog shown here from Dreamweaver that requests the items that we used in our script.

Ill 15-2



However, such cut-and-paste or fill-and-go JavaScript is not what we aim to teach. Let's consider going further than the simple rollover.

Extending Rollovers

Canned rollover codes like the one just presented could be improved. With a little ingenuity you could write a rollover script that you do not need to bind **onmouseover** and **onmouseout** code with. Consider making a class name indicating rollovers and having JavaScript loop through the document finding these **<img>** tags and inferring the appropriate images to preload and then dynamically binding the triggering events via JavaScript. This type of very clean rollover could be referenced via an external .js file and cached in all needed pages. This would avoid your need to copy-paste similar rollover code all over your site, which seems to be common practice on the Web and exactly what editors like Dreamweaver create.

Besides improving the coding style of rollovers, we might extend them to perform other functions. For example, a rollover might reveal text or imagery someplace else on the screen as the user moves over a link. A script can be written to reveal a scope note providing information about the destination link. You might even provide an image that users can rollover and learn details about the object by revealing another image. Once you understand the basic idea of rollovers, you're limited only by your imagination (and your users' tolerance for fancy effects!).

The following markup and JavaScript illustrate how one such enhancement might work:

L 15-14

```
<!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>Targeted Rollovers</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script type="text/javascript">
<!--
// Preload all images
if (document.images)
{
  var abouton = new Image();
  abouton.src = "abouton.gif";
  var aboutoff = new Image();
  aboutoff.src = "aboutoff.gif";
  // ... possibly more buttons ...
  var blank = new Image();
  blank.src = "blank.gif";
  var description1 = new Image();
  description1.src = "description.gif";
  // ... possibly more descriptions ...
}
/* Turns the given image on and at the same time shows the description */
function on(imgName, description)
{
  if (document.images)
  {
    imgOnSrc = eval(imgName + "on.src");
    document.images[imgName].src = imgOnSrc;
    document.images["descriptionregion"].src = description.src;
```

```
  }
}
/* Turns the given image off and at the same time blanks the description */
function off(imgName)
{
  if (document.images)
   {
     imgOffSrc = eval(imgName + "off.src");
     document.images[imgName].src = imgOffSrc;
     document.images["descriptionregion"].src = "blank.gif";
   }
}
//-->
</script>
</head>
<body>
<a href="about.html"
onmouseover="on('about', description1);window.status='Company';return true;"
onmouseout="off('about');window.status='';return true;"><img src="aboutoff.gif"
  border="0" alt="About" name="about" id="about" width="159" height="57" /></a>
<!-- ... possibly more buttons ... -->
     
<a href="#"><img src="blank.gif" name="descriptionregion"
   id="descriptionregion" width="328" height="84" border="0" alt="" /></a>
</body>
</html>
```

Figure 15-2 shows the rollover code in action.

FIGURE 15-2 Updating a separate region of the document in response to a rollover

While it would seem from the previous example that JavaScript rollovers are potentially useful, their days are somewhat numbered given that many of these effects are vastly improved with the inclusion of CSS in a Web page.

The End of JavaScript Rollovers?

With the rise of Cascading Style Sheets (CSS), the need for JavaScript-based rollover code has diminished greatly. Already developers have discovered that rollovers are in some sense "expensive" in that they require the download of extra images for the rollover effect. For simple navigation items this penalty is just not worth it and many Web developers are opting instead for simple rollover effects using a CSS **:hover** property, like so:

L 15-15

```
<style type="text/css">
a:hover    {background-color: yellow; font-weight: bold;}
</style>
```

If you take the idea of hover further you might even change the background image of a region to create a more graphical rollover. To do this, set the rollover region to contain a transparent GIF with some alt text and then swap the background-image on hover. With this simple CSS you now have a degradable and accessible graphical rollover effect without any JavaScript! The following example illustrates this idea.

L 15-16

```
<!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>CSS Rollover Example</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<style type="text/css">
a img {height: 35px; width: 70px; border-width: 0; background: top left
no-repeat;}

a#button1 img {background-image: url(button1off.gif);}
a#button2 img {background-image: url(button2off.gif);}

a#button1:hover img {background-image: url(button1on.gif);}
a#button2:hover img {background-image: url(button2on.gif);}
</style>
</head>
<body>
<div id="navbar">
 <a id="button1" href="http://www.javascriptref.com/"><img src="blank.gif"
alt="JavaScript Ref"></a>

 <a id="button2" href="http://www.google.com/"><img src="blank.gif"
alt="Google"></a>
</div>
</body>
</html>
```

With CSS you can go even further and address the multiple image download problem that plagues rollovers. For example, we might create one large image of navigation buttons in a menu in their on state and one large image of the buttons in their off state, as shown here:

Ill 15-3

Then we would use CSS clipping regions in conjunction with either **:hover** rules or JavaScript to reveal and hide pieces of the image to create the rollover effect. With a simple approach like this we would cut down eight image requests if the buttons were separated to two since they are together. While CSS is quite powerful by itself and it can be used to replace some simple visual effects like rollovers, we'll see that it is even more powerful when combined with JavaScript to create DHTML effects.

Traditional Browser-Specific DHTML

We've seen how JavaScript can be used to dynamically update images in the page in response to user actions. But if you consider that almost all parts of the page are scriptable in modern browsers, you'll realize that manipulating images is only the tip of the iceberg. Given browser support, you can update not just images but also text and other content enclosed in tags, particularly ** **'s, embedded objects, forms, and even the text in the page. You're not just limited to changing content, either. Because most objects expose their CSS properties, you can change appearance and layout as well.

Three technologies come together to provide these features: (X)HTML provides the structural foundation of content, CSS contributes to its appearance and placement, and JavaScript enables the dynamic manipulation of both of these features. This combination of technologies is often referred to as "Dynamic HTML," or *DHTML* for short, particularly when the effect created appears to make the page significantly change its structure. We start first with the traditional example of DHTML, positioned regions, and address how developers have addressed the troublesome cross-browser issues they have encountered. Once we have clearly demonstrated the problems with this approach to DHTML, we will present DOM Standard-oriented DHTML with a smattering of Internet Explorer details where appropriate.

Cross-Browser DHTML with Positioned Regions

For many, a major perceived downside of DHTML is that, because traditional object models are so divergent, doing anything non-trivial requires careful implementation with cross-browser issues in mind. Even when the interfaces by which DHTML is realized are uniform, browsers are notorious for interpreting standards in slightly different ways, so you'll need to carefully

test your scripts to ensure their behavior is as desired in the browsers used by your demographic. In this section we provide a brief example of the cross-browser headache by exploring how to create simple DHTML effects with positioned regions that work in both standards-aware and non-standards–aware browsers. Hopefully, the inconvenience of the workarounds and various arcane issues presented will encourage readers to spend time focusing on the standards-oriented DHTML that follows this section.

CSS Positioning Review

Given how important positioning is for DHTML, we present here a brief review of the related CSS. Positioning is generally controlled with the combination of the **position**, **top**, **bottom**, **right**, and **left** properties. Table 15-2 lists these and other relevant properties.

There are three primary types of positioning. An element with **static** positioning is placed where it would normally occur in the layout of the document (also called flow positioning). An element with **relative** positioning is positioned at the offset given by **top**, **bottom**, **left**, and/or **right** from where it would normally occur in the layout. That is, the document is laid out and then elements with relative positioning are offset from their position by the indicated amount. The final type of positioning is **absolute**, meaning the element is not laid out as a normal part of the document but is positioned at the indicated offset *with respect to its parent (enclosing) element*.

CSS Property	Description
position	Defines the type of positioning used for an element: *static* (default), *absolute*, *relative*, *fixed*, or *inherit*. Most often *absolute* is used to set the exact position of an element regardless of document flow.
top	Defines the position of the object from the top of the enclosing region. For most objects, this should be from the top of the content area of the browser window.
left	Defines the position of the object from the left of the enclosing region, most often the left of the browser window itself.
height	Defines the height of an element. With positioned items, a measure in pixels (**px**) is often used, though others like percentage (**%**) are also possible.
width	Defines the width of an element. With positioned items, a measure in pixels (**px**) is often used.
clip	A clipping rectangle like **clip: rect (*top right bottom left*)** can be used to define a subset of content that is shown in a positioned region as defined by the rectangle with upper-left corner at (*left*,*top*) and bottom-right corner at (*right*,*bottom*). Note that the pixel values of the rectangle are relative to the clipped region and not the screen.
visibility	Sets whether an element should be visible. Possible values include *hidden*, *visible*, and *inherit*.
z-index	Defines the stacking order of the object. Regions with higher **z-index** number values stack on top of regions with lower numbers. Without **z-index**, the order of definition defines stacking, with last object defined the highest up.

TABLE 15-2 Position-Related Properties of **Style** Objects

NOTE *CSS2 also supports the idea of fixed positioning, which allows an object to stay pegged to a particular location regardless of window scrolling. However, it is not supported in IE6 or before and should be avoided.*

Absolutely positioned elements not contained within any other elements (save the **<body>**) are easy to move about the page in a dynamic way using JavaScript because their enclosing element is the entire document. So any coordinates assigned to their positional properties become their position on the page. We can also hide positioned regions by setting their **visibility**, change their size by setting their **height** and **width** values, and even change their content using the commonly supported **innerHTML** property or resorting to DOM methods as discussed in Chapter 10. However, while it sounds easy in practice, there are many different ways positioned objects are accessed with JavaScript in browsers.

Netscape 4 Positioned Regions: Layers

Netscape 4 did not provide excellent support for CSS1. However, it does support the **<layer>** tag, which provides the equivalent of positioned regions in style sheets. For example:

L 15-17
```
<layer name="test" pagex="100" pagey="100" width="100" height="50"
bgcolor="#ffff99">
      This is a layer!
</layer>
```

produces the same region as:

L 15-18
```
<div id="test" style="position: absolute; top: 100px; left: 100px; width: 100px;
height: 50px; background-color: #ffff99;">
      This is a layer!
</div>
```

Based on the previous example you might guess that you then have to include both ** ** and **<layer>** tags in a document in order to achieve proper layout across browsers. Fortunately, just before release, Netscape 4 adopted support for positioned ** ** tags. Note though that this support is actually through a mapping between ** ** regions and **Layer** objects. In fact, to access a positioned ** ** object under Netscape 4, you use the **layers[]** collection. To demonstrate this, consider that to access a region defined by:

L 15-19
```
<div id="region1" style="position: absolute; top: 100px; left: 100px; width:
100px; height: 100px; background-color: #ffff99;">
      I am positioned!
</div>
```

we would use **document.layers['region1']**. However, once accessed, we cannot unfortunately modify the **style** property of a region. Yet we can modify important values such as position, size, or visibility under Netscape 4. For example, to change the visibility we would use **document.layers['region1'].visibility** and set the property to either *hide* or *show*. The various modifiable aspects of a positioned region map actually map directly to the properties of the **Layer** object. The most commonly used properties for this object are shown in Table 15-3.

Property	Description
background	The URL of the background image for the layer.
bgColor	The background color of the layer.
clip	References the clipping region object for the layer. This object has properties **top**, **right**, **bottom**, and **left** that correspond to normal CSS clipping rectangles as well as **width** and **height**, which can be used similarly to normal **width** and **height** properties in CSS.
document	A reference to the **Document** object of the current layer.
left	The x-coordinate position of the layer.
name	The name of the layer.
pageX	The x-coordinate of the layer relative to the page.
pageY	The y-coordinate of the layer relative to the page.
src	The URL to reference the layer's content when it is not directly set within the **<layer>** tag itself.
top	The y-coordinate position of the layer.
visibility	Reference to the current visibility of the layer. Values of *show* and *hide* for **<layer>** are equivalent to *visible* and *hidden* under CSS. Later versions of Netscape 4 map the two values so either can be used.
window	Reference to the **Window** object containing the layer.
x	The x-coordinate value for the layer.
y	The y-coordinate value for the layer.
zIndex	Holds the stacking order of the layer.

TABLE 15-3 Useful **Layer** Object Properties

Of course, **<layer>** is an extremely proprietary tag and is not supported outside Netscape 4. In fact, in the 6.*x* (and later) release of the browser, Netscape removed support for this tag. We'll see in the next few sections how Internet Explorer and DOM-compatible browsers access positioned regions.

Internet Explorer 4+ Positioned Regions

As mentioned in Chapter 9, Internet Explorer exposes all objects in a page via the **all[]** collection. So to access a positioned region defined by:

L 15-20
```
<div id="region1" style="position: absolute; top: 100px; left: 100px; width:
100px; height: 100px; background-color: #ffff99;">
      I am positioned!
</div>
```

under Internet Explorer 4 and greater, you would use **document.all['region1']** or **document.all.region1** or simply **region1**. Once the particular object in question was accessed we could manipulate its presentation using the **Style** object. For example, to set the background

color of the region to orange as set by the CSS property **background-color,** we would use **document.all['region1'].style.backgroundColor = 'orange'** or simply **region1.style.backgroundColor='orange'**. To set visibility, we would use **region1.style.visibility** and set the value to either *visible* or *hidden*.

The style property to JavaScript property mapping was presented in Chapter 10, but recall once again that in general you take a hyphenated CSS property and uppercase the first letter of the hyphen-separated terms, so the CSS property **text-indent** becomes **textIndent** under IE and DOM-compatible JavaScript. The next section shows a slight variation to the scheme presented here since the standard DOM supports different syntax to access a positioned region. Fortunately, since Internet Explorer 5 and beyond we can really use either syntax interchangeably.

DOM Positioned Regions

Access to positioned regions under a DOM-compliant browser is pretty much nearly as easy as using Internet Explorer's **all[]** collection with ** ** tags. The primary method would be to use the **document.getElementById()** method. Given our sample region specified with a ** ** called 'region1', we would use **document.getElementById('region1')** to retrieve the region and then we can set its visibility or other style-related properties via the **Style** object in a similar fashion to Internet Explorer. For example, to change visibility of an object to hidden we use **document.getElementById('region1').style.visibility='hidden'**. Of course the question then begs: How do we get and set style properties related to layer positioning in the same way across all browsers. The next section presents one possible solution to this challenge.

Building a Cross-Browser DHTML Library

As we have just seen, as well as in many other examples in the book, significant differences exist in technology support between the popular Web browsers, particularly those that are not up to date with standards. For some developers, authoring for one browser (Internet Explorer) or the standard (DOM) has seemed the best way to deal with these differences. But sometimes one must address cross-browser compatibility head-on and write markup and script that works under any browser capable of producing the intended result. This section explores this approach by creating a sample cross-browser layer library. While it is by no means the only way to implement such a library, it does illustrate common techniques used for such tasks.

From the previous sections, we can see that for layer (content region) positioning and visibility we will need to support three different technologies:

- Netscape 4 proprietary **<layer>** tags
- Internet Explorer 4+ **all[]** collections with positioned ** ** tags
- DOM-compatible browsers with positioned ** ** tags

Given these tractable requirements, we can create a suite of JavaScript routines to change visibility and move, modify, size, and set the contents of positioned regions in major browsers fairly easily.

The first thing such a library needs to do is identify the browser of the current user. The easiest way to do this is by looking at the **Document** object. If we see a **layers[]** collection,

we know the browser supports Netscape 4 layers. We can look at the **all[]** collection to sense if the browser supports Internet Explorer's **all[]** collection syntax. Lastly, we can look for our required DOM method **getElementById()** to see if we are dealing with a DOM-aware browser. The following statements show how to set some variables indicating the type of browser we are dealing with:

L 15-21
```
var layerobject = ((document.layers) ? (true) : (false));
var dom = ((document.getElementById) ? (true) : (false));
var allobject = ((document.all) ? (true) : (false));
```

Once we know what kind of layer-aware browser we are dealing with, we might define a set of common functions to manipulate the layers. We define the following layer functions to handle common tasks:

L 15-22
```
function hide(layerName) { }
function show(layerName) { }
function setX(layerName, x) { }
function setY(layerName, y) { }
function setZ(layerName, zIndex) { }
function setHeight(layerName, height) { }
function setWidth(layerName, width) { }
function setClip(layerName, top, right, bottom, left) { }
function setContents( ) { }
```

These are just stubs that we will fill out shortly, but first we will need one special routine in all of them to retrieve positioned elements by name, since each approach does this slightly differently.

L 15-23
```
function getElement(layerName, parentLayer)
{
 if(layerobject)
  {
    parentLayer = (parentLayer) ? parentLayer : self;
    layerCollection = parentLayer.document.layers;
    if (layerCollection[layerName])
      return layerCollection[layerName];

    /* look through nested layers */
    for (i=0; i < layerCollection.length;)
      return(getElement(layerName, layerCollection[i++]));
  }

  if (allobject)
    return document.all[layerName];
  if (dom)
    return document.getElementById(layerName);
}
```

Notice the trouble that the possibility of nested **<layer>** or ** ** tags under Netscape causes. We have to look through the nested layers recursively until we find the object we are looking for or until we have run out of places to look.

Once a positioned element is accessed, we can then try to change its style. For example, to hide and show a positioned region we might write:

L 15-24
```
function hide(layerName)
{
   var theLayer = getElement(layerName);
   if (layerobject)
     theLayer.visibility = 'hide';
   else
     theLayer.style.visibility = 'hidden';
}

function show(layerName)
{
   var theLayer = getElement(layerName);
   if (layerobject)
     theLayer.visibility = 'show';
   else
     theLayer.style.visibility = 'visible';
}
```

The other routines are similar and all require the simple conditional detection of the browser objects to work in all capable browsers.

Of course, there are even more issues than what has been covered so far. For example, under older Opera browsers, we need to use the **pixelHeight** and **pixelWidth** properties to set the **height** and **width** of a positioned region. In order to detect for the Opera browser, we use the **Navigator** object to look at the user-agent string, as discussed in Chapter 17. Here we set a Boolean value to indicate whether we are using Opera by trying to find the substring "opera" within the user-agent string.

L 15-25
```
opera = (navigator.userAgent.toLowerCase().indexOf('opera') != -1);
```

Once we have detected the presence of the browser, we can write cross-browser routines to set height and width, as shown here:

L 15-26
```
/* set the height of layer named layerName */
function setHeight(layerName, height)
{
  var theLayer = getElement(layerName);

  if (layerobject)
    theLayer.clip.height = height;
  else if (opera)
    theLayer.style.pixelHeight = height;
  else
     theLayer.style.height = height+"px";
}

/* set the width of layer named layerName */
function setWidth(layerName, width)
{
  var theLayer = getElement(layerName);
```

```
  if (layerobject)
    theLayer.clip.width = width;
  else if (opera)
    theLayer.style.pixelWidth = width;
  else
    theLayer.style.width = width+"px";
}
```

The same situation occurs for positioning with Opera, as it requires the use of **pixelLeft** and **pixelTop** properties rather than simply **left** and **top** to work. See the complete library for the function for setting position that is similar to the previous example.

We must also take into account some special factors when we write content to a layer. Under Netscape 4, we use the **Document** object methods like **write()** to rewrite the content of the layer. In Internet Explorer and most other browsers, we can use the **innerHTML** property. However, under a strictly DOM-compatible browser, life is somewhat difficult, since we would have to delete all children from the region and then create the appropriate items to insert. Because of this complexity and the fact that most DOM-supporting browsers also support **innerHTML,** we punt on this feature. This leaves Opera versions prior to Opera 7, though we wrote the code in such a manner that simply nothing happens rather than an error message being displayed.

L 15-27

```
function setContents(layerName, content)
{
   var theLayer = getElement(layerName);

   if (layerobject)
     {
       theLayer.document.write(content);
       theLayer.document.close();
       return;
     }

   if (theLayer.innerHTML)
     theLayer.innerHTML = content;
}
```

We skipped discussion of a few routines, but their style and usage follow the ones already presented. The complete layer library is presented here:

L 15-28

```
/* layerlib.js: Simple Layer library with basic
   compatibility checking */

/* detect objects */
var layerobject = ((document.layers) ? (true) : (false));
var dom = ((document.getElementById) ? (true) : (false));
var allobject = ((document.all) ? (true) : (false));

/* detect browsers */
opera=navigator.userAgent.toLowerCase().indexOf('opera')!=-1;

/* return the object for the passed layerName value */
function getElement(layerName,parentLayer)
{
```

PART IV

```
 if(layerobject)
  {
    parentLayer = (parentLayer)? parentLayer : self;
    layerCollection = parentLayer.document.layers;
    if (layerCollection[layerName])
      return layerCollection[layerName];
    /* look through nested layers */
    for(i=0; i < layerCollection.length;)
      return(getElement(layerName, layerCollection[i++]));
  }

  if (allobject)
    return document.all[layerName];

  if (dom)
    return document.getElementById(layerName);
}

/* hide the layer with id = layerName */
function hide(layerName)
{
   var theLayer = getElement(layerName);
   if (layerobject)
     theLayer.visibility = 'hide';
   else
     theLayer.style.visibility = 'hidden';
}

/* show the layer with id = layerName */
function show(layerName)
{
   var theLayer = getElement(layerName);
   if (layerobject)
     theLayer.visibility = 'show';
   else
     theLayer.style.visibility = 'visible';
}

/* set the x-coordinate of layer named layerName */
function setX(layerName, x)
{
   var theLayer = getElement(layerName);
   if (layerobject)
     theLayer.left=x;
   else if (opera)
     theLayer.style.pixelLeft=x;
   else
     theLayer.style.left=x+"px";
}

/* set the y-coordinate of layer named layerName */
function setY(layerName, y)
{
   var theLayer = getElement(layerName);
```

```
   if (layerobject)
     theLayer.top=y;
   else if (opera)
     theLayer.style.pixelTop=y;
   else
     theLayer.style.top=y+"px";
}

/* set the z-index of layer named layerName */
function setZ(layerName, zIndex)
{
   var theLayer = getElement(layerName);

   if (layerobject)
     theLayer.zIndex = zIndex;
   else
     theLayer.style.zIndex = zIndex;
}

/* set the height of layer named layerName */
function setHeight(layerName, height)
{
   var theLayer = getElement(layerName);

   if (layerobject)
     theLayer.clip.height = height;
   else if (opera)
     theLayer.style.pixelHeight = height;
   else
     theLayer.style.height = height+"px";
}

/* set the width of layer named layerName */
function setWidth(layerName, width)
{
  var theLayer = getElement(layerName);

  if (layerobject)
     theLayer.clip.width = width;
  else if (opera)
     theLayer.style.pixelWidth = width;
  else
     theLayer.style.width = width+"px";
}

/* set the clipping rectangle on the layer named layerName
   defined by top, right, bottom, and left */
function setClip(layerName, top, right, bottom, left)
{
   var theLayer = getElement(layerName);

   if (layerobject)
     {
        theLayer.clip.top = top;
        theLayer.clip.right = right;
        theLayer.clip.bottom = bottom;
```

```
        theLayer.clip.left = left;
     }
   else
     theLayer.style.clip = "rect("+top+"px "+right+"px "+" "+bottom+"px "+left+"px
 )";

}

/* set the contents of layerName to passed content*/
function setContents(layerName, content)
{
   var theLayer = getElement(layerName);

   if (layerobject)
     {
       theLayer.document.write(content);
       theLayer.document.close();
       return;
     }

   if (theLayer.innerHTML)
      theLayer.innerHTML = content;
}
```

We might save this library as "layerlib.js" and then test it using an an example document like the one that follows here. If you want to avoid a lot of typing, make sure to visit the support site at **www.javascriptref.com**.

L 15-29

```
<!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>Cross-browser Layer Tester</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script type="text/javascript" src="layerlib.js"></script>
</head>
<body>
<div id="region1" style="position: absolute; top: 10px; left: 300px;
                         width: 100px; height: 100px;
                         background-color: #ffff99; z-index: 10;">
     I am positioned!
</div>

<div id="region2" style="position: absolute; top: 10px; left: 275px;
                         width: 50px; height: 150px;
                         background-color:#33ff99; z-index: 5;">
    Fixed layer at z-index 5 to test z-index
</div>
<br /><br /><br /><br /><br /><br />
<hr />
<form name="testform" id="testform" action="#" method="get">
Visibility:
<input type="button" value="show" onclick="show('region1');" />
<input type="button" value="hide" onclick="hide('region1');" />
```

```
<br /><br />
x: <input type="text" value="300" name="x" id="x" size="4" />
   <input type="button" value="set"
     onclick="setX('region1',document.testform.x.value);" />

y: <input type="text" value="10" name="y" id="y" size="4" />
   <input type="button" value="set"
    onclick="setY('region1',document.testform.y.value);" />

z: <input type="text" value="10" name="z" id="z" size="4" />
   <input type="button" value="set"
    onclick="setZ('region1',document.testform.z.value);" />
<br /><br />

Height: <input type="text" value="100" name="height" id="height" size="4" />
        <input type="button" value="set"
         onclick="setHeight('region1',document.testform.height.value);" />

Width: <input type="text" value="100" name="width" id="width" size="4" />
       <input type="button" value="set"
        onclick="setWidth('region1',document.testform.width.value);" />
<br /><br />

Clipping rectangle:  <br />
top: <input type="text" value="0" name="top" id="top" size="4" />
left: <input type="text" value="0" name="left" id="left" size="4" />
bottom: <input type="text" value="100" name="bottom" id="bottom" size="4" />
right: <input type="text" value="100" name="right" id="right" size="4" />
<input type="button" value="set"
 onclick="setClip('region1',document.testform.top.value,
             document.testform.right.value, document.testform.bottom.value,
             document.testform.left.value);" />

<br /><br />
<input type="text" name="newcontent" id="newcontent" size="40"
 value="I am positioned!" />
<input type="button" value="set content"
 onclick="setContents('region1',document.testform.newcontent.value);" />
</form>
</body>
</html>
```

NOTE *If you type this example into an HTML document, be sure to fix the line wrapping: Neither attribute values nor string literals in JavaScript are permitted to span multiple lines.*

A rendering of the library and example in action is shown in Figure 15-3.

Playing around with this script, you will find that you might encounter problems under Netscape 4 if you position the layer to cover the form elements in the page. You also may encounter a resize bug that causes the page to lose layout on window resize. The first problem is generally not solvable, but we can solve the latter problem by adding a somewhat

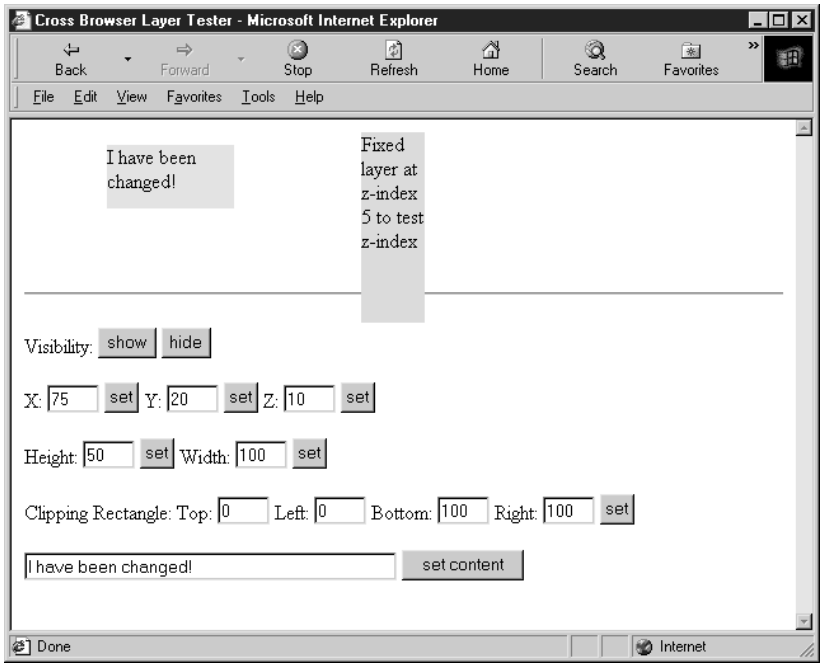


FIGURE 15-3 Testing our cross-browser content region library

clunky fix that reloads the page every time it is resized. It is presented here for readers to add to their library as a fix for this strictly Netscape 4 problem.

L 15-30

```
/* Reload window in Nav 4 to preserve layout when resized */
function reloadPage(initialload)
{
  if (initialload==true)
    {
       if ((navigator.appName=="Netscape") &&
           (parseInt(navigator.appVersion)==4))
        {
          /* save page width for later examination */
          document.pageWidth=window.innerWidth;
          document.pageHeight=window.innerHeight;

          /* set resize handler */
          onresize=reloadPage;
        }
    }
  else if (innerWidth!=document.pageWidth ||
           innerHeight!=document.pageHeight)
          location.reload();
}

/* call function right away to fix bug */
reloadPage(true);
```

In the final examination, the harsh reality of DHTML libraries like the one presented here is that minor variations under Macintosh browsers and the less common JavaScript-aware browsers (such as Opera) can ruin everything. The perfect application of cross-browser DHTML is certainly not easily obtained, and significant testing is always required. The next section explores standards-oriented DHTML, which should soon provide at least some relief from cross-browser scripting headaches.

Standards-Based DHTML

It would seem that for true DHTML we need to employ browsers in which CSS, DOM, and (X)HTML standards are actually well supported. While complete support for CSS1, CSS2, DOM1, and DOM2 cannot be found in all browsers, more often than not there is sufficient support to permit most DHTML applications you can think of using the standard rather than relying on the ideas presented in the previous section.

One of the most fundamental tasks in DHTML is to define a region of the page whose appearance or content you wish to manipulate. In standards-based browsers this is easy: just about any tag such as **<p>**, **<h1>**, or **<pre>** can be used. However, these tags come with a predefined meaning and rendering in most browsers, so (X)HTML provides two generic tags that have no default rendering or meaning: ** ** (the generic block-level element) and **** (the generic inline element). Note that in most of the examples we will stick with the ** ** tag since its support with CSS and JavaScript tends to be the most consistent across browser versions.

Style Object Basics

The appearance of content areas defined by ** ** or other tags for that matter is best manipulated via the object's **style** property (corresponding to the contents of the **style** attribute for the element). The **Style** object found in this property exposes the CSS attributes for that object, enabling control of the content's visual characteristics such as font, color, size. (For a full list of **Style** properties, see Chapter 10 or Appendix B).

Consider the following simple text-based rollover effect:

L 15-31
```
<a href="http://www.google.com" onmouseover="this.style.fontWeight='bold';"
onmouseout="this.style.fontWeight='normal';">Mouse over me!</a>
```

When the user mouses over the link, the font is switched to bold (the equivalent of a **font-weight: bold** CSS binding), and the font is switched back on mouseout. This is similar to the ideas from the section on rollovers but rather than changing the source of the image we instead change the CSS properties of the object. We could, of course, set an arbitrary CSS property if we follow the convention of taking the CSS property name and removing the dash and upper-casing the initial letter of merged words to get its JavaScript/DOM property. So given the CSS property, **font-style** in JavaScript becomes **fontStyle**, **background-image** becomes **backgroundImage**, font-size becomes **fontSize**, and so on.

To illustrate broad-based appearance changes, the following example will change the appearance of a region when it is clicked.

L 15-32
```
<!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>Standards-based DHTML</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script language="JavaScript" type="text/javascript">
<!--
var prevObj;  // So we can revert the style of the previously clicked element
function handleClick(e)
{
  if (!e)
   var e = window.event;
  // e gives access to the event in all browsers

  // If they previously clicked, switch that element back to normal
  if (prevObj)
   {
    switchAppearance(prevObj);
   }
  if (e.target)  // DOM
  {
   prevObj = e.target;
   switchAppearance(e.target);
  }
 else if (e.srcElement)  // IE
  {
    prevObj = e.srcElement;
   switchAppearance(e.srcElement);
  }
}

function switchAppearance(obj)
{
  obj.style.backgroundColor = ((obj.style.backgroundColor == "lightblue") ?
                               ("") : ("lightblue"));
  // IE can't handle a value of inherit so pass it a blank value
  // Avoid messing with the border around form fields
  if (obj.tagName.toLowerCase() != "input")
  {
    if (obj.style.borderStyle.indexOf("solid") != -1)
        {
        obj.style.borderStyle = "none";
        obj.style.borderWidth = "0px";
        }
       else
        {
        obj.style.borderStyle = "solid";
        obj.style.borderWidth = "1px";
        }
   }
}
// Register DOM style events
if (document.addEventListener)
  document.addEventListener("click", handleClick, true);
// Register IE style events
if (document.attachEvent)
  document.attachEvent("onclick",handleClick);
```

```
//-->
</script>
</head>
<body>
<h2>Click anywhere on the page to see the content regions!</h2>
<br /><br />
<p style="float: left;">Some content that floats to the left.</p>
<p style="float: right;clear: none;">Some content that floats to the right.</p>
<br clear="all"/><hr />
<form action="#" method="get">
Here's a form!<br />
<input type="text" /><br />
<input type="text" /><br />
<input type="text" /><br />
</form>
<p>And another paragraph!</p>
</body>
</html>
```

NOTE *To make the example work in Internet Explorer 6, we had to employ the cross-platform event capture ideas presented in Chapter 11 since this browser does not support the DOM Level 2 style of event listeners.*

This example changes the background color and border of the content regions on the screen defined by the (X)HTML. The example is not just useful in that it shows style changes with events, but it illustrates that markup and CSS structure are inherent in any document. A sample rendering in the Mozilla browser after clicking the form is shown in Figure 15-4.

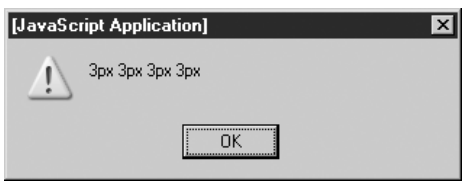
The previous example illustrates two very important points. The first observation is that to employ DHTML to manipulate the appearance of pages requires an intimate knowledge of CSS. Otherwise, you're limited to manipulating elements' (X)HTML attributes rather than their **Style** objects. The second observation is that properties of **Style** objects contain CSS values, and these values might not be what you'd expect. To illustrate, consider the following JavaScript:

L 15-33

```
<p id="mypara">Oy.</p>
<script type="text/javascript">
document.getElementById("mypara").style.borderWidth = 3;
alert(document.getElementById("mypara").style.borderWidth);
</script>
```

The results under Internet Explorer and Mozilla might surprise you:

Ill 15-4



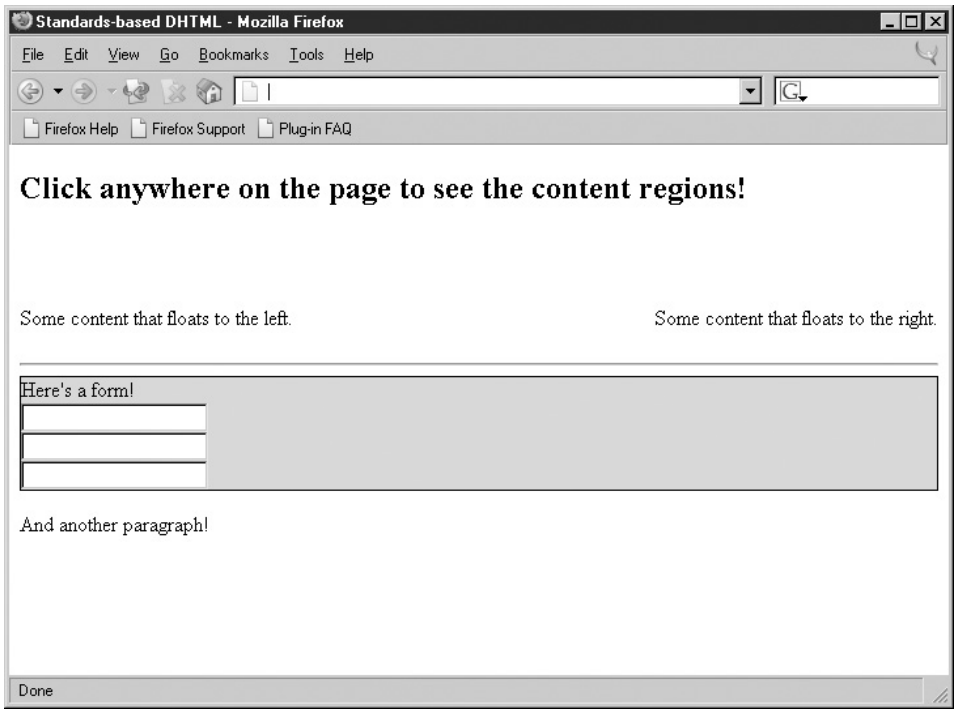


FIGURE 15-4 DHTML in standards-supporting browsers requires knowledge of CSS.

First, notice that we assigned the border width with a numeric value without specifying any units. In the case of **border-width** we should have specified the units directly and passed in a string value rather than employing implicit type conversion like so:

L 15-34
```
document.getElementById("mypara").style.borderWidth = "3px";
```

Even more interesting is that you see under Mozilla-based browsers how **border-width** actually is shorthand for the four sides of the border, thus it shows four values. As you can see when you set a property of the **Style** object, the value is parsed as if it appeared in a style sheet. Thus, the browser generally fills in any missing or implied CSS rules (such as units like "px") you might have omitted or it may just simply ignore the value in some cases. Intimate knowledge of CSS really is required but in case your CSS is a little rusty you might follow these best practices for manipulating **Style** properties to help stay out of trouble:

- Do not use **Style** properties to store state if possible. For example, if you want to keep track that you've set a background to red, use a separate variable (possibly an instance property of the **Style**) instead of inferring state from **style.color**. Doing so will save you from the headaches of dealing with unexpected values filled in by the browser.
- If you must examine **Style** properties, do so using substrings and/or regular expressions rather than direct comparisons with operators like ==. Doing so reduces type-conversion errors and problems related to properties whose implied values are filled in by the browser.

- Set **Style** properties as strings, and always be as specific as possible. For example, instead of using **style.borderWidth = 2**, use **style.borderWidth = "2px"**. This will reduce the risk of error and increase compatibility with less robust CSS implementations.

Effective Style Using Classes

Setting a large number of **Style** properties dynamically can be tiresome and error prone. A better technique is to bind the CSS properties you want to a class, and then swap an object's class dynamically. The following example illustrates the technique by mirroring the value of the **class** attribute from any ** ** the user mouses over into a target content region:

L 15-35

```
<!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>Changing Classes</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<style type="text/css">
#mirror {border-style: solid; border-width: 1px; width: 100%;}
#theStyles {border-style: dashed; border-width: 1px; width: 80%; padding: 5%;}
h1 {text-align: center;}

/* the classes to swap */
.big {font-size: 48pt;}
.small {font-size: 8pt;}
.important {text-decoration: underline; font-weight: bold;}
.annoying {background-color: yellow; color: red;}
</style>
<script type="text/javascript">
<!--
function changeClass(whichClass)
{
  document.getElementById("mirror").className = whichClass;
}
//-->
</script>
</head>
<body>
<h1>Result</h1>
<div id="mirror">Mouse over any of the text below and watch this text mirror
  its CSS properties.</div>
<br /><br />
<h1>Styles to Test</h1>
<div id="theStyles">
 <div onmouseover="changeClass(this.className)" class="big">
  This text is big!
 </div>
<hr />
 <div onmouseover="changeClass(this.className)" class="small">
  This text is small!
 </div>
<hr />
```

```
 <div onmouseover="changeClass(this.className)" class="important">
  This text is important!
 </div>
<hr />
 <div onmouseover="changeClass(this.className)" class="annoying">
   This text is annoying!
 </div>
</div>
</body>
</html>
```

Notice how, in the previous example, we used **className** to access the (X)HTML **class** attribute. We must do so because "class" is a reserved word in JavaScript, and we need to therefore avoid using that identifier whenever we can.

Computed Styles

One subtlety of the **style** property of document objects is that it represents only the *inline* style applied to that element. Inline styles are those specified using the **style** (X)HTML attribute. As a result, there's no guarantee that the values accessed this way represent the style ultimately displayed by the browser. For example:

L 15-36

```
<!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>What's my style?</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<style type="text/css">
  p { text-decoration: underline !important }
</style>
</head>
<body>
<p id="para">This text always appears underlined, even when we try to override it
 by setting its inline style</p>
<script type="text/javascript">
 document.getElementById("para").style.textDecoration = "none";
 alert(document.getElementById("para").style.textDecoration);
</script>
</body>
</html>
```

As you can see in Figure 15-5, the text remains underlined even though we've set the inline style property to "none." The reason is that there is a CSS rule in the document-wide style sheet that overrides the inline setting using **!important**. However, alerting the style value clearly shows that the value for **textDecoration** is none, which is somewhat confusing.

Getting the *actual* style applied to an object can be tricky. In DOM2-compliant browsers you can use the **getComputedStyle()** method of the document's default view. A document's *default view* is its default representation in the Web browser, that is, its appearance once all style rules have been applied. The **getComputedStyle()** method takes two arguments: a node for which style should be gotten and the pseudo-element (e.g., ":hover") of interest (or the

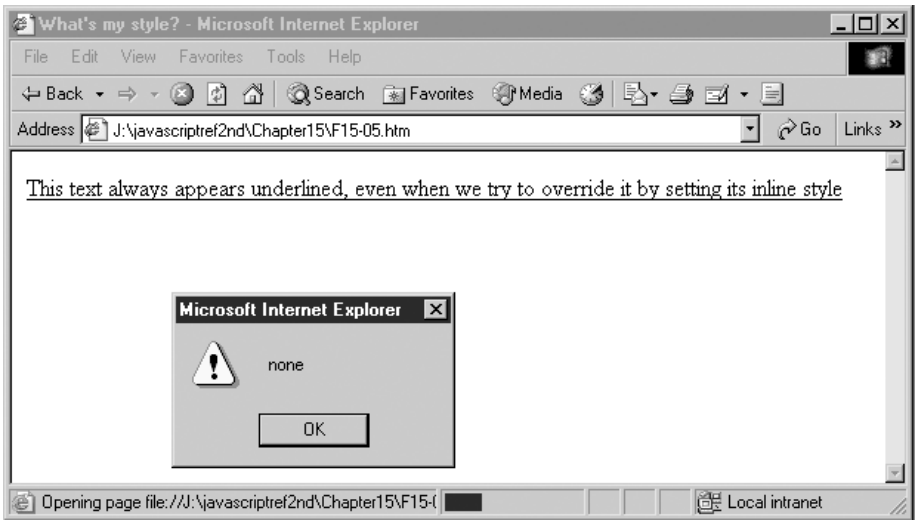


FIGURE 15-5 Computed style and actual style may vary.

empty string for the normal appearance). You might get the style of the paragraph in the previous example with:

L 15-37
```
var p = document.getElementById("para");
var finalStyle = document.defaultView.getComputedStyle(p, "");
```

To examine individual properties, use the **getPropertyValue()** method, which takes a string indicating the property of interest:

L 15-38
```
alert("The paragraph's actual text decoration is: " +
      finalStyle.getPropertyValue("text-decoration"));
```

Unfortunately, as you get into the more esoteric aspects of DOM2, browser support varies significantly from vendor to vendor. Even worse under IE6 and earlier you won't find support for this approach but instead will be required to use **currentStyle** to calculate an object's current property values. We present an example that works both with the proprietary and DOM syntax here.

L 15-39
```
<!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>What's my style? Take 2</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<style type="text/css">
  p { text-decoration: underline !important }
</style>
</head>
<body>
```

```
<p id="para">This text always appears underlined, even when we try to override it
by setting its inline style</p>
<script type="text/javascript">
 document.getElementById("para").style.textDecoration = "none";
 alert("The paragraph's defined text decoration is: "+
document.getElementById("para").style.textDecoration);
 var p = document.getElementById("para");
 if (p.currentStyle)
   alert("The paragraph's actual text decoration is: " +
p.currentStyle.textDecoration);
 else
   {
     var finalStyle = document.defaultView.getComputedStyle(p, "");
      alert("The paragraph's actual text decoration is: " +
      finalStyle.getPropertyValue("text-decoration"));
   }
</script>
</body>
</html>
```

NOTE *Even when computed styles are implemented you may find that browsers have somewhat limited implementation and not all styles defined by CSS2 are exposed.*

In this section we've only touched on the fundamental aspects of dynamic manipulation of objects' **style** properties. Given a solid understanding of CSS, much more is possible. The extent to which DHTML can be realized in modern browsers is quite amazing: it's possible to build, modify, and deconstruct documents or parts of documents on the fly, with a relatively small amount of code. We present a few examples of these effects next.

Applied DHTML

This section provides a brief introduction to some DHTML effects that are possible. The examples focus on maximum cross-browser and backward compatibility and use the layerlib.js presented in the section "Building a Cross-Browser DHTML Library." While the examples should work under the common browsers from the 4.*x* generation on, because of bugs with clipping regions, you may find some of the examples do not work under some versions of Opera or other browsers.

Simple Transition

With positioned layers, you can hide and show regions of the screen at any time. Imagine putting colored regions on top of content and progressively making the regions smaller. Doing this would reveal the content in an interesting manner, similar to a PowerPoint presentation. While you can create such transitions easily with filters under Internet Explorer, this effect should work in most modern browsers. The code for this effect is shown here, and its rendering is shown in Figure 15-6.

L 15-40

```
<!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>Wipe Out!</title>
```

```
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<style type="text/css">
<!--
.intro { position:absolute;
         left:0px;
         top:0px;
         layer-background-color:red;
         background-color:red;
         border:0.1px solid red;
         z-index:10; }

#message { position: absolute;
           top: 50%;
           width: 100%;
           text-align: center;
           font-size: 48pt;
           color: green;
           z-index: 1;}
-->
</style>
<script type="text/javascript" src="layerlib.js"></script>
</head>
<body>
<div id="leftLayer" class="intro"> </div>
<div id="rightLayer" class="intro"> </div>

<div id="message">JavaScript Fun</div>

<script type="text/javascript">
<!--
var speed = 20;

/* Calculate screen dimensions */
 if (window.innerWidth)
       theWindowWidth = window.innerWidth;
 else if ((document.body) && (document.body.clientWidth))
       theWindowWidth = document.body.clientWidth;
  if (window.innerHeight)
       theWindowHeight = window.innerHeight;
  else if ((document.body) && (document.body.clientHeight))
       theWindowHeight = document.body.clientHeight;

      /* nasty hack to deal with doctype switch in IE */
 if (document.documentElement  && document.documentElement.clientHeight &&
document.documentElement.clientWidth)
     {
         theWindowHeight = document.documentElement.clientHeight;
         theWindowWidth = document.documentElement.clientWidth;
     }

/* cover the screen with the layers */
  setWidth('leftLayer', parseInt(theWindowWidth/2));
  setHeight('leftLayer', theWindowHeight);
  setX('leftLayer',0);
```

```
  setWidth('rightLayer', parseInt(theWindowWidth/2));
  setHeight('rightLayer', theWindowHeight);
  setX('rightLayer', parseInt(theWindowWidth/2));

  clipright = 0;
  clipleft =  parseInt(theWindowWidth/2);

function openIt()
{
  window.scrollTo(0,0);
  clipright+=speed;
  setClip('rightLayer',0,theWindowWidth, theWindowHeight,clipright);

  clipleft-=speed;
  setClip('leftLayer',0,clipleft,theWindowHeight,0);

  if (clipleft<0)
    clearInterval(stopIt)
}

function doTransition()
{
  stopIt=setInterval("openIt()",100);
}
window.onload = doTransition;
//-->
</script>
</body>
</html>
```

A point of interest in this example is the **setInterval(*code*, *time*)** method of the **Window** object, which is used to perform the animation. The basic use of this method, which is fully presented in Chapter 12, is to execute some specified string *code* every *time* milliseconds. To turn off the interval, you clear its handle, so that if you have ***anInterval* = setInterval ("alert('hi')", 1000),** you would use **clearInterval(*anInterval*)** to turn off the annoying alert.

Targeted Rollovers (Take 2)

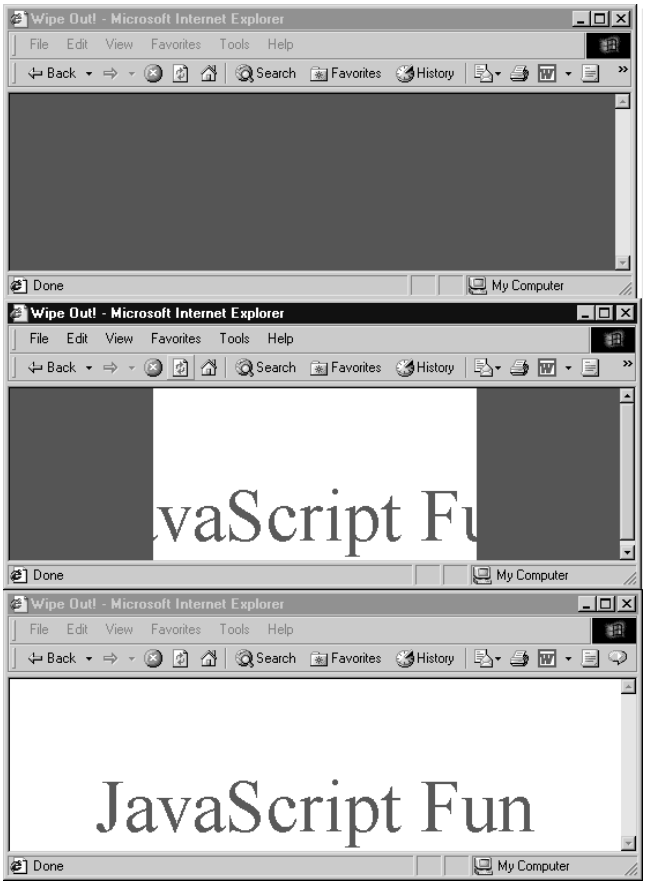
We saw earlier in the chapter how a rollover effect might reveal a region on the screen containing a text description. This form of targeted rollover, often called a *dynamic scope note*, can be implemented without CSS by using images, but with the DOM- and CSS-positioned items we may have a much more elegant solution. As an example, look at the code for simple scope notes presented here.

L 15-41

```
<!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>CSS Rollover Message</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<style type="text/css">
<!--
#buttons {position: absolute; top: 10px;
          background-color: yellow;width: 20%;}
```

```
#description {position: absolute;top: 10px;left: 40%;}
-->
</style>
<script src="layerlib.js" type="text/javascript"></script>
</head>
<body>
<div id="buttons">
<a href="about.html"
   onmouseover="setContents('description',
      'Discover the history and management behind the Democompany.');"
   onmouseout="setContents('description', ' ');">About</a>
<br /><br />

<a href="products.html"
   onmouseover="setContents('description',
                    'If you like our domes, you\'ll love our robots!');"
   onmouseout="setContents('description', ' ');">Products</a>
</div>
<div id="description"> </div>
</body>
</html>
```

FIGURE 15-6
A simple DHTML page transition



NOTE *Without the non-breaking space ( ), you may find that the description layer will collapse under HTML and thus not instantiate the required object for manipulation via JavaScript.*

General Animation

The last example in this chapter presents some very simple animation using JavaScript. We move an object up and down to particular coordinates as well as left to right. The basic idea will be to figure out the current position of an object and then move the object incrementally around the screen using the **setX()** and **setY()** functions in our layer library. First, we add simple **getX(*layerName*)** and **getY(*layerName*)** functions that return the coordinates of the layer passed. These routines are shown here.

L 15-42

```
/* return the X-coordinate of the layer named layerName */
function getX(layerName)
{
   var theLayer = getElement(layerName);
   if (layerobject)
     return(parseInt(theLayer.left));
   else
     return(parseInt(theLayer.style.left));
}

/* return the y-coordinate of layer named layerName */
function getY(layerName)
{
   var theLayer = getElement(layerName);

   if (layerobject)
     return(parseInt(theLayer.top));
   else
     return(parseInt(theLayer.style.top));
 }
```

Next, we need to define some variables to indicate how many pixels to move at a time (*step*) and how quickly to run animation frames (*framespeed*).

L 15-43

```
/* set animation speed and step */
var step = 3;
var framespeed = 35;
```

We should also define some boundaries for our moving object so it doesn't crash into our form controls that will control the animated object.

L 15-44

```
/* set animation boundaries */
var maxtop = 100;
var maxleft = 100;
var maxbottom = 400;
var maxright = 600;
```

Next, we'll add routines to move the object in the appropriate direction until it reaches the boundary. The basic idea will be to probe the current coordinate of the object, and if it isn't yet at the boundary, move it a bit closer by either adding or subtracting the value of

step and then setting a timer to fire in a few milliseconds to continue the movement. The function **right()** is an example of this. In this case, it moves a region called "ufo" until the right boundary defined by *maxright* is reached.

L 15-45

```
function right()
{
  currentX = getX('ufo');

  if (currentX < maxright)
   {
    currentX+=step;
    setX('ufo',currentX);
    move=setTimeout("right()",(1000/framespeed));
   }
  else
    clearTimeout(move);
}
```

The complete script is shown here with a rendering in Figure 15-7.

L 15-46

```
<!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>UFO!</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script src="layerlib.js" type="text/javascript"></script>
<script type="text/javascript">
<!--
/* return the x-coordinate of the layer named layername */
function getX(layername)
{
  var theLayer = getElement(layername);
  if (layerobject)
    return(parseInt(theLayer.left));
   else
    return(parseInt(theLayer.style.left));
}

/* return the y-coordinate of layer named layerName */
function getY(layerName)
{
   var theLayer = getElement(layerName);

   if (layerobject)
     return(parseInt(theLayer.top));
   else
     return(parseInt(theLayer.style.top));
}

/* set animation speed and step */
var step = 3;
var framespeed = 35;
```

```
/* set animation boundaries */
var maxtop = 100;
var maxleft = 100;
var maxbottom = 400;
var maxright = 600;

/* move up until boundary */
function up()
{
  var currentY = getY('ufo');
  if (currentY > maxtop)
  {
    currentY-=step;
    setY('ufo',currentY);
    move=setTimeout("up()",(1000/framespeed));
  }
  else
    clearTimeout(move);
}

/* move down until boundary */
function down()
{
  var currentY = getY('ufo');
  if (currentY < maxbottom)
  {
    currentY+=step;
    setY('ufo',currentY);
    move=setTimeout("down()",(1000/framespeed));
  }
  else
    clearTimeout(move);
}

/* move left until boundary */
function left()
{
  var currentX = getX('ufo');
  if (currentX > maxleft)
  {
    currentX-=step;
    setX('ufo',currentX);
    move=setTimeout("left()",(1000/framespeed));
  }
  else
    clearTimeout(move);
}

/* move right until boundary */
function right()
{
  var currentX = getX('ufo');
  if (currentX < maxright)
  {
```

```
    currentX+=step;
    setX('ufo',currentX);
    move=setTimeout("right()",(1000/framespeed));
  }
  else
    clearTimeout(move);
}
//-->
</script>
</head>
<body background="space_tile.gif">

<div id="ufo" style="position:absolute; left:200px; top:200px; width:241px;
 height:178px; z-index:1;">
 <img src="space_ufo.gif" width="148" height="141" alt="ufo!" />
</div>

<form action="#" method="get">
   <input type="button" value="up" onclick="up();" />
   <input type="button" value="down" onclick="down();" />
   <input type="button" value="left" onclick="left();" />
   <input type="button" value="right" onclick="right();" />
   <input type="button" value="stop" onclick="clearTimeout(move);" />
</form>
</body>
</html>
```

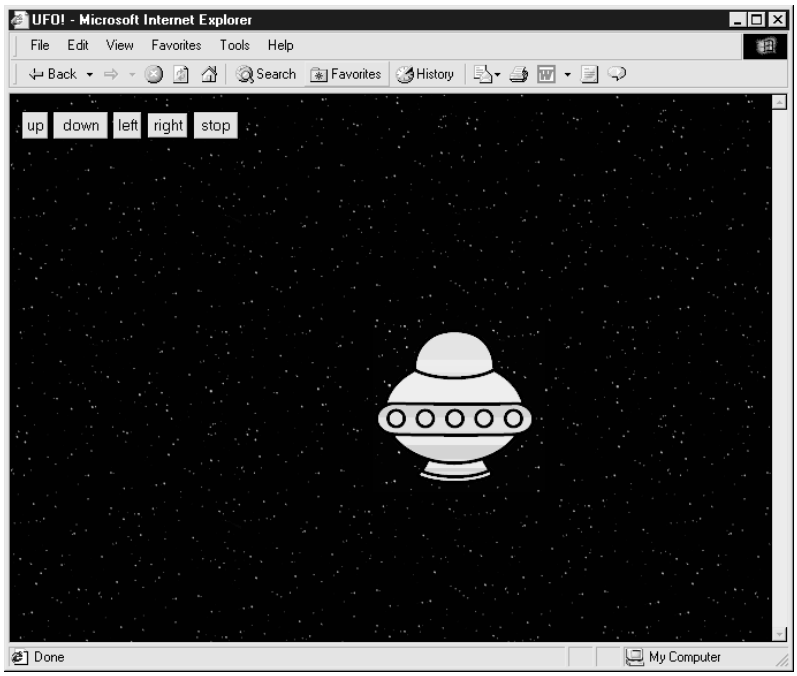


FIGURE 15-7 A JavaScript UFO in flight

We could modify the animation example to multiple regions and to move across a predefined path. Yet the question is: *Should we*?

Practical DHTML

Practically speaking, DHTML effects should be used sparingly. First off, there are many JavaScript bugs associated with positioning objects and manipulating their clipping regions. Careful testing and defensive coding practices (as discussed in Chapter 22) would need to be applied. Second, many of these effects, as we saw with rollovers, can be created in technologies other than JavaScript such as CSS or Flash. Animations in particular raise many questions. While you can implement them with JavaScript, you may find that the animations strobe or move jerkily. Without significantly complex programming, you won't have perfect animations under JavaScript. However, by using Flash or even simple animated GIFs, you can achieve some very interesting effects—often with far less complexity. We're big fans of picking the most appropriate technology in which to implement any particular solution. For fancy effects, the appropriate solution is rarely JavaScript. A wonderful rule of thumb is that effects for the sake of effects are not worth the effort. JavaScript should be used to add functionality—not glitz—to your site.

If you're dead set on using JavaScript, there are many interesting effects that can be achieved. A few examples are presented at the support site at **www.javascriptref.com** as well as at the numerous JavaScript library sites online, such as DynamicDrive (**www.dynamicdrive.com**).

Summary

This chapter presented some common applications of the **Image** object as well as other visual effects commonly associated with JavaScript. We saw that while many of these effects are relatively easy to implement, the scripting and style sheet variations among the browsers require defensive programming techniques to prevent errors from being thrown in browsers that do not support the required technology. DHTML effects, such as animations, visibility changes, and movement, demonstrate the high degree of effort required to make cross-browser–compliant code. While all the effects demonstrated in this chapter are relatively simple, developers should not necessarily add them to their site.