Table of Contents

Online Sections I–VII: Applying Dynamic HTML

ONLINE SECTIONS I-VII

Applying Dynamic HTML

Online Sections I through VII try to make sense of the alphabet soup of industry standards surrounding DHTML and demonstrates the use of cascading style sheets, element positioning, dynamic content, and scripting events. These sections explain how Microsoft Internet Explorer and Netscape Navigator implement the various DHTML technologies, and they discuss how to develop cross-browser web applications.

Online Section I, *The State of the Art: Standards*

Online Section II, *Cross-Platform Compromises*

Online Section III, *Adding Cascading Style Sheets to Documents*

Online Section IV, *Changing Page Content and Styles*

Online Section V, *Adding Dynamic Positioning to Documents*

Online Section VI, *Scripting Events*

Online Section VII, *XMLHttpRequest and Ajax*

ONLINE SECTION I

The State of the Art: Standards

When viewed from today's media-centric world, the old days of the World Wide Web—way back in the early to mid-1990s—are mere sepia-toned recollections. Today, with so many personal computers and portable devices having always-on access to the Web at home, work, and leisure, the world's users expect not only quick answers, but efficient and entertaining displays of data, akin to the rich experience offered by today's videogame-charged e-atmosphere.

Although coined by Macromedia in 2002, the term "Rich Internet Application" has come to mean a combination of software and content that, once delivered to a client computer (typically in a web browser), has a life of its own. Users interact with content and interface elements just as they do in standalone applications installed on their computers. But if the user creates or modifies information, the data is typically stored on the server, not the client computer. The next time the user accesses the application—even from a different browser on a different computer—the user expects the application to behave the same way, with previously preserved data ready for use.

A variety of technologies have allowed the creation of rich Internet applications for nearly 10 years. Java applets, available in popular browsers since 1997, didn't take the world by storm as predicted. In contrast, Macromedia's Flash technology has attracted a dedicated following. Even so, the universal browser support of the standards-based Hypertext Markup Language (HTML), ease of content creation in that medium, and the fact that content composed as HTML becomes easily searchable through web search engines have contributed to the overwhelming popularity of HTML as a fundamental content medium. Then, thanks to a few browser technology and standards developments over the years, even the once stodgy and static HTML content can become *dynamic*—web pages can "think and do" on their own, with little or no help from the server once they have been loaded in the browser.

The allure of the Web—in theory anyway—is that publishers and application developers can rely on well-known standards that facilitate the rendering of, and interaction with, data. Freed from details of painting dots on monitor screens, managing

memory, and controlling internal data flows on dozens of operating systems, publishers and developers can focus on their content and server-side data handling. Browsers do all the operating-system-specific dirty work by interpreting Hypertext Markup Language (HTML) and other directives embedded in the content.

Although dynamic web pages are implemented under the umbrella of Dynamic HTML (DHTML), successful deployment requires knowledge of several technologies and standards that exist well outside the charter of the original HTML Working Group. In this chapter, I'll discuss the applicable standardization efforts, including some proprietary items that have become de facto standards. As disparate as this collection may appear at first, they all magically come together as a system to let creative designers implement engaging DHTML content.

The Standards Alphabet Soup

There is no such thing as a single Dynamic HTML standard. DHTML is an amalgam of specifications that stem from multiple standards efforts and proprietary technologies built into recent versions of popular browsers.

As a savvy web content author these days, you must know the acronyms of all relevant standards, such as HTML, XHTML, CSS, DOM, and ECMA for starters. Bear in mind, however, that browser makers and standards bodies work according to their separate release schedules. Sometimes a browser implements a new feature not yet part of a standards track; and sometimes a standard specifies items that are not yet built into the latest browser. As righteous as it sounds to compose content that is strictly "standards-compliant," you must cede the final power to what is implemented in browsers you intend to support for your web site or application. Users want pages to work in their browsers, regardless of what lies within the source code. Any plaudits you receive for adhering as closely as possible to standards will come from your fellow content authors and whatever personal satisfaction you derive from the effort.

Version Headaches

As a further complication, there are the inevitable prerelease versions of browsers and standards. Browser prereleases are sometimes called "preview editions" or "beta" versions. While not officially released, these versions give you a chance to see what new functionality will be available for content rendering and control in the next-generation browsers. Authors who follow browser releases closely sometimes worry when certain aspects of their current pages fail to work properly in prerelease versions. The fear is that the new version of the browser is going to break a carefully crafted masterpiece that runs flawlessly in earlier released versions of the browser.

Prerelease browsers are valuable resources for content developers. On the one hand, testing existing content on a preview release allows you to uncover bugs in the browser before the browser is released. Report those bugs! Wherever possible, provide a simple test case (or link to a test case) that proves the bug so that the browser engineers can see exactly what you mean and test their fixes against real code. On the other hand, testing may allow you to learn about the rare case in which a feature you rely on is removed or changed (usually to meet a standard definition). While browsers tend to be backward compatible with previous versions, many developers were caught unaware, for instance, when the Mozilla browser (successor to Netscape 4), in its effort to adopt industry standards, completely dropped support for a couple of popular, but non-standard, features implemented in Netscape 4. Similarly, Internet Explorer 7 fixed a bug that many content authors had previously relied on to help differentiate browser brands. Developers who tested preview versions of the browsers in question would have learned of these important changes early in the process, and planned for the new deployment ahead of time.

Avoid the urge, however, to modify your public HTML or scripting code to accommodate what may be a temporary bug in a prerelease version of a browser. Any page visitor who uses a prerelease browser does so at his or her own risk. If your pages are breaking on that browser, they're probably not the only ones on the Web that are breaking. A user of a prerelease browser must understand that using such a browser for mission-critical web work is as dangerous as entrusting one's Great Novel to a beta version of a word processing program.

On the standards side, working groups usually publish prerelease (draft) versions of their standards. These documents are very important to the people who build browsers and authoring tools. The intent of publishing a working draft is not much different from making a prerelease browser version public: to get as many concerned netizens as possible looking over the material to find flaws or shortcomings before the standard is finalized.

Speaking of standards, it is important to recognize that the final releases of these documents from standards bodies are called not "standards" but "recommendations." Content authors find, to their dismay, that browser makers don't always interpret details of recommendations the same way. You will also find details within a recommendation that are optional, thus allowing a browser maker to claim full compliance, even when not every feature is implemented.

No law or contract forces browser makers to implement recommendations in full. Fortunately, from a marketing angle, it plays well to the web development audience that a company's browser adheres to the "standards." Eventually—after enough release cycles of both standards and browsers allow everyone to catch up with each other, and older, less standards-compliant browsers fall into disuse—our lives as content creators should become easier. Unfortunately, the fulfillment of that dream seems eternally to be five years in the future.

In the meantime, the following sections provide brief evolutionary histories of the various standards, and their implementation in major browsers, as they relate to the technologies that affect DHTML.

HTML

In an otherwise fast-moving environment, it's hard to believe that the current Worldwide Web Consortium (W3C) document content markup standard, Version 4.01, was published in December of 1999. Since then, most attention has been paid to the XHTML effort (described below) as a replacement. In late 2006, however, W3C cofounder, Tim Berners-Lee, signaled the revitalization of HTML in parallel with XHTML. Whatever the future might hold, we must recognize the critical role that HTML 4.x has played in the evolution of content delivery on the Web and how authors construct content.

Many of the features that were new to HTML 4 were designed for browsers that make the graphical user interface of a web page more accessible to users who cannot see a monitor or use a keyboard (see "Web Accessibility Initiative (WAI)," later in this Online Section). The new tags and attributes also acknowledge that a key component of the name World Wide Web is World. Users of all different written and spoken languages need equal access to the content of the Web. Thus, HTML 4 included support for the alphabets of most languages and provided the ability to specify that a page be rendered from right to left, rather than left to right, to accommodate languages that are written that way.

Perhaps the most important long-term impact of HTML 4, however, was distancing a web page's content from its formatting (presentation). Strictly speaking, the purpose of HTML is to provide structural meaning to the content of a document. That's what each tag does: this blurb of text is a paragraph, another segment is labeled internally as an acronym, and a block over there is reserved for data loaded in from an external multimedia file. HTML 4 sought to wean authors from the once-familiar tags that make text larger, bold, and red, for example. That kind of information is formatting information, and it belongs to a separate standardization effort related to content style.

In the HTML 4 world, a content author indicates that a chunk of text in a paragraph is to receive emphasis based on that text's context within the document. The HTML standard, however, does not dictate whether the browser conveys emphasis through a bold or italic or green font. Instead, a separate style definition controls the formatting for an emphasized string of text. This separation of content and style allows the same content to be rendered differently for a variety of output devices. When emphasized text is viewed in a browser on a video monitor, the color may be green and the style italic, but when the same HTML markup is viewed through a projection system, it may be a different shade of green, to compensate for the different ambient

lighting conditions, and bold, so it is more readable at a distance. And when the content is being read aloud electronically for a blind user, the synthesized voice speaks the tagged words with more vocal emphasis. The key point here is that the content—the words in this case—is written and tagged once. Style definitions, either in the same document or maintained in separate files that are linked into the document, can be modified and enhanced independently of the content.

HTML 4 was also the first version of HTML to account for the role that client-side scripting was playing in the real world. Not only did `<script>` and `<noscript>` tags become part of the specification, but most elements that get rendered on the page had a basic set of scripting event handler attributes explicitly defined for them (`onclick`, `onmouseover`, `onkeypress`, and the like). If nothing else, these acknowledgments validated the idea of client-side processing instructions delivered as part of the document. It also allowed HTML-validating programs to accept attributes that link elements to script actions.

The long-term stability of the HTML 4.01 recommendation means that virtually every popular browser released since 2001 supports (or claims complete support for) the standard. Therefore, you will find satisfactory or better HTML 4.01 support starting with Internet Explorer 6 (Windows), Internet Explorer 5 (Macintosh), Mozilla 0.9.4, Safari (Macintosh), and Opera 4.

XHTML

The industry-wide drive to embrace the Extensible Markup Language (XML) way of describing electronic information had a big impact on the W3C HTML Working Group. For example, while HTML has a rigid set of elements and attributes, XML supports the creation of new elements and attributes that accommodate a new or specialized (discipline-specific) kinds of data. It was a natural extension, therefore, to reconfigure the HTML 4 standard so that it also adhered to fundamental XML tagging practices. The result is an XML-ized HTML recommendation, or XHTML for short.

XHTML 1.0/1.1

Version 1.0 of the XHTML recommendation is a very thin document because it encompasses all HTML 4.01 elements and attributes by reference. The bulk of the document describes how to compose HTML in a way that conforms to XML markup. Most of the XML-oriented features of XHTML govern how authors should format source code and structure elements and attributes in a document.

In concert with the trend of many W3C recommendations, XHTML evolved to embrace the idea of modularity. In place of one giant XHTML recommendation, the group divided elements into 20 modules of related elements. The first version of the standard that embraced modularization is known as XHTML 1.1. The rationale

behind modularization is to allow specialized devices to support subsets of the XHTML standard that make the most sense for the content they intend to convey or the rendering facilities of the device. For example, an HTML-rendering engine embedded into a cellular telephone might not need to support the loading of external code objects. Such a phone-based "browser" could support all XHTML modules except for the Object Module, which contains the `object` and `param` HTML elements.

It is important to bear in mind that as of XHTML 1.1, no new elements or attributes were added to what had been in the HTML 4.01 recommendation. Some long-standing items were deprecated, however. With only a few exceptions, the source code structure and formatting requirements of XHTML coincide with many coders' existing styles. The requirements simply impose a more rigorous style in order to pass XML source code validation. The key requirements are:

- HTML tag and attribute names must be spelled only in lowercase characters.
- All attribute values must be quoted.
- All elements that can contain other elements require an end tag.
- All empty elements, such as `img` and `hr`, require either an end tag or must include a forward slash immediately before the right angle bracket (as in `<hr/>`).
- All attributes must be stated as name-value pairs, including those that had previously been defined as one-word, minimalized attributes (e.g., `checked="checked"`).
- The `name` attribute is deprecated in favor of the `id` attribute.
- The `target` attribute (of the `a` and `form` elements, for instance) is not permitted in Strict validation.

Moreover, in recognition of compatibility issues with HTML code that must also run on non-XHTML-aware browsers, the XHTML 1.x recommendations include provisions for backward compatibility. For example, because older browsers may become confused by an end tag for an empty element (a `</br>` tag, for example), you can use the XML internalized slash technique, but with an extra space before the end slash, as in `<br />` or `<hr />`, which older browsers accommodate. In the case of forms, where browsers continue to need the `name` attribute to convey the form or form control's identifier with a submission, you can assign the same identifier to the `name` and `id` attributes of a `form` element or form control.

Thanks to the backward-compatible design of XHTML 1.x, most HTML 4 browsers experience no difficulty with XHTML markup served as an HTML file (i.e., a file served with the `text/html` MIME type). A problem does arise, however, in Internet Explorer 6 or 7 and a host of earlier browsers, which do not understand the preferred MIME type for XHTML: `application/xhtml+xml`. The preferred MIME type is supported by Mozilla, Opera 7.1 (Windows), Opera 6 (Macintosh), and Safari 1.2.

XHTML 2.0

Based on the working draft of the XHTML 2.0 recommendation, numerous new markup elements and attributes are on the way. For example, any element can be made a hyperlink by simply including an `href` attribute whose value is a target URL. A new type of list element, `nl`, provides a context for navigation lists (commonly marked up today as `ul` and `li` elements) whose appearance and interaction can be independently governed by style sheet rules and/or scripts. Support for W3C activities such as XFORMS and XMLEVENTS are incorporated into this new XHTML version. Full details and the current status are available at *http://www.w3.org/TR/xhtml2*.

Mozilla 1.8 offers support for several XHTML 2.0 features, most typically to enhance accessibility of content and user interface elements. Because XHTML 2.0 is not fully backward-compatible with previous versions and non-supporting browsers, it is unlikely to find its way into your repertoire for several years to come if you develop content for public use.

HTML or XHTML?

Because even the latest browsers (and many more versions to come) must be compatible with a Web full of HTML content in the real world, the decision to adhere to XHTML markup practices is up to each content developer. If you take comfort in having your source code successfully pass XHTML validation, you can specify the XHTML document type at the top of your documents. Be aware, however, that such validation will reject proprietary attributes (such as IE-specific event handler attributes) unless they are labeled with XML namespaces. But even if you are not overly concerned with following the XHTML recommendation, you should nevertheless gravitate toward its formatting requirements, especially the top three bullet points in the list above; they will become the norm as automated authoring tools begin generating code to meet that standard.

Example listings throughout this book focus more on the scripting than the tagging. Therefore, HTML code examples in the present edition of this book follow a middle ground in the HTML/XHTML discussion. They adhere to XHTML formatting for tags and attributes. Except where intended for full backward compatibility, examples try to avoid elements and attributes deprecated in HTML 4 to instill good practice as authors move toward XHTML. Also, for the sake of demonstration clarity and brevity, tags occasionally include event and other attributes that go beyond the limited set that validates under XHTML-Strict. Unless otherwise specified, however, all listings assume a browser's default document type. If you are an experienced XHTML author, you won't have to go far to adapt these listings to more stringent XHTML document types if you so desire.

Cascading Style Sheets

A style sheet contains specifications for the rendered appearance of content on the page. The link between a style sheet and the content it influences is either the tag name of the HTML element that holds the content or an identifier associated with the element by way of an attribute (notably the `id` or `class` attribute). When a style sheet defines a border, the style definition doesn't know (or care) whether the border will be wrapped around a paragraph of text, an image, or an arbitrary group of elements. All the style knows is that it specifies a border of a particular thickness, color, and type for whatever element or elements are associated with the style. That's how the separation of presentation from content works: the content is ignorant of the style and the style is ignorant of the content.

A W3C working group undertook the task of creating a supplementary markup syntax that allowed styles to be associated with HTML content (*http://www.w3c.org/Style/*). The technology, called Cascading Style Sheets (CSS), matured relatively quickly during a time when mainstream browser versions had difficulty keeping up with the latest standards. The W3C document that contains the most detailed information about CSS is the second version of the recommendation, called CSS2. This version includes the original CSS1 standard, special features for element positioning (initially released separately as CSS-P), and a large number of features that are new with CSS2. A modified version of the recommendation, CSS2.1, includes numerous corrections and clarifications.

Just as the current XHTML effort embraces modularization, so does CSS. The recommendation known as CSS3 is a modularized version of CSS2.1, but also with many additional properties. Given the enormous size and range of style properties in CSS2.1, modularization provides browsers an opportunity to claim support for well-defined subsets of the CSS recommendation, without supporting features that don't apply to the devices they support.

CSS Rationale

The Cascading Style Sheets recommendation lets authors define style rules that are applied to HTML elements. A rule may apply to a single element, a related group of elements, or all elements of a particular tag (such as all `p` elements). Style rules influence the rendering of elements, including their color, alignment, border, margins, and padding between borders and content. Style rules can also control specialty items, such as whether an `ol` element uses letters or roman numerals as item markers.

Theoretically, CSS frees you from the anarchy behind the arbitrary way that each browser measures fonts and other values. Font sizes can be specified in real pixel or point sizes, instead of the inexact 1-through-7 relative scale of HTML. If you want a paragraph or a picture indented from the left margin, you can do so with the preci-

sion of ems or picas, instead of relying on hokey arrangements of tables and transparent images. (Of course, in practice, a browser's default style sheet and user preference settings can still prevent text styles from appearing identical everywhere. We're still a long way from replicating the precision of print publishing on pages viewed through a web browser.)

Many of the style specifications that go into CSS rules derive their inspiration from now-deprecated (that is, soon-to-be-deleted) HTML tag attributes that used to be the only way to control visual aspects of elements. Visual properties, such as element alignment, belong in style sheet rules, rather than `align` or `valign` attributes inside an element tag. In some cases, style sheet rules even supplant entire HTML elements. For example, in the world of CSS, you do not direct font changes for a string of text within a paragraph by way of `<font>` tags. Instead, you define the font characteristics for that special text in a style sheet rule and then associate the rule with a structural HTML element that surrounds the affected content.

The earliest browsers to support a substantial amount of CSS1 were Netscape Navigator 4 and Internet Explorer 4 (IE 3 implemented a smaller set of CSS1 properties). These early implementations exhibited numerous quirks in the ways the more complex style features work. This was especially true in Navigator 4 with respect to form controls and tables (inheritance rules frequently failed) and in all browsers in the area of CSS-produced element borders. You find much more thorough support for CSS1 and a healthy selection of CSS2 properties starting in IE 5 for the Macintosh, IE 5.5 for Windows, and Mozilla-based browsers, as well as Safari and recent versions of Opera. With such broad support among installed browsers, basic style sheet control of content formatting is deployed very commonly around the Web.

Speaking of "Quirks"

While the CSS specs were solidifying, modern browser makers found that the new standards didn't always mesh with their previous interpretations of the evolving standards. But the browsers also had to walk a fine line between supporting the terabytes of legacy HTML published on the Web and promoting the "right" way of marking up text according to present-day standards. To that end, most modern browsers allow your documents to dictate whether the browsers should behave the old, quirky way (so that your old CSS code continues to work the same way in the new browsers), or the new, standards-compatible way (and sometimes one in-between mode). The switch that toggles the browser between modes is the content of the `<!DOCTYPE>` element at the top of your HTML file..

That's not to say that interpretations of the CSS standard are identical across all browsers. Implementations of some details are notoriously problematic, requiring CSS designers to test extensively on a wide variety of browsers. It's not uncommon

for large applications to serve up separately-designed style sheets based on the class of browser requesting a web page.

Element Positioning and Layering

Begun as a separate working group effort, Cascading Style Sheets-Positioning offered script authors much more in the way of interactivity on a page: more of the D in DHTML. Its inclusion into the CSS2 recommendation validated the techniques and user interface possibilities that positioning offers.

The basic notion of positioning is that an element can be placed in its own plane above the main document. The element lives in its own transparent layer, so it can be hidden, shown, precisely positioned, and moved around the page without disturbing the other content or the layout of the document. It was CSS-based positioning that first allowed overlapping of HTML elements.

As remarkable as these features sound, the syntax for turning an element into a positioned element is no more difficult than making an element's text appear in a color or bold font weight. A handful of CSS-P properties, described in Online Section V, follow the same syntax conventions as other CSS properties.

By controlling position-related properties of an element, a script can make elements fly around the page or it can allow the user to drag elements around the page. Content can pop up out of nowhere or expand to let the viewer see more content—all without reloading the page or contacting the server. Scripted positioning with nearly identical cross-browser syntax is possible starting with IE 5 (Windows and Mac), Mozilla, Safari, and Opera 4.

(Implemented only in Netscape 4.x, the `layer` element and associated scripting was abandoned in favor of CSS positioning for the Mozilla project. Reference chapters in this book list these old Netscape 4-only items for historical purposes only.)

Document Object Model

When an HTML page loads into a scriptable browser, the browser creates an internal roadmap of all the elements it recognizes as scriptable objects. This roadmap is hierarchical in nature, with the most "global" object—the browser window or frame—containing a document, which, in turn, contains, say, a form, which, in turn, contains form control elements. For a script to communicate with one of these objects, it must be able to reference the object in order to call one of the object's methods or set one of its property values. Document objects are the "things" that scripts work with.

Without question, the most difficult challenge facing scripters throughout the short history of scriptable browsers has been how each browser builds its internal roadmap of objects. This roadmap is called a *document object model* (DOM). If one

browser implements an object as scriptable but another doesn't, it drives scripters and page authors to distraction. Pioneering scripters felt the sting of this problem when they implemented image-swapping mouse rollovers in Navigator 3, only to discover that images were not scriptable objects in the contemporary Internet Explorer 3. As a result, their IE 3 users were getting script errors when visiting the sites and moving their mice across the hot images. The situation only worsened when Microsoft developed its DOM for IE 4 in one direction, while Netscape took a different path for Navigator 4.

In an effort to standardize this area, a separate working group of the W3C is charged with setting recommendations for a Document Object Model (*http://www.w3c.org/DOM/*). The foundation for the W3C DOM is the Core DOM module, which defines fundamental building blocks (objects, properties, and methods) that apply to any document-oriented web content (whether pure XML or HTML and its descendants). The Core module includes definitions for basic objects, such as the document, an element, element text content, and an element's attributes. Beyond those generic building blocks of the Core, specialized modules define the objects that apply to particular kinds of documents or entities within documents. For example, the working group patterned an HTML module of the W3C DOM after the elements defined for the W3C HTML 4 specification. Every HTML 4 element is represented in the HTML DOM module as a DOM object. Thus, the HTML DOM includes objects for the `p` element, the `form` element, and so on down the line. HTML 4 element attributes, in turn, become properties of HTML DOM element objects. This is how a script can read the value assigned to an attribute in the source code and perhaps modify the value in response to user activity.

A primary goal of this effort is to create a recognized common denominator among browsers (or any document engine). By the time the first round of W3C DOM standards came into being, however, Netscape and Microsoft had already deployed two or more versions of their own rapidly diverging DOMs. With companies that held sometimes radically different philosophies participating in the W3C DOM process, finding a common ground was not easy.

The resulting W3C recommendation, as best described in the modularized DOM Level 2, created a DOM that in many ways resembled none of the existing models. While the recommendation maintains backward compatibility with early object models (as implemented in Navigator 3 and IE 3), and it exposes all HTML elements as objects just as the IE 4 model did, the W3C DOM created an entirely new methodology for working with pieces of a document. Perhaps the greatest impact on DOM coding practices at the time was in the way scripts may reference elements.

DOM Level 0

The first object model, which the W3C informally refers to as DOM Level 0 (but in truth predates the W3C DOM and was never a formal standard), was restricted to

only a handful of element objects. Inside a window, the `document` object was the master container of all content. After a document loaded into the browser, its content was largely static with the exception of forms and form controls (text fields, buttons, and select lists). References to element objects (plus a couple of abstract objects, such as `location` and `history`) entailed a hierarchical name, starting with the `document` object. To refer to a form control, the reference "walked" through an element hierarchy that matched the nested tag hierarchy in the document—at least to the extent that elements were exposed as objects. Thus, a reference to a form control included the document, the form, and the control itself:

```
document.formName.controlName
```

These models also treated multiple instances of elements as arrays of those elements. Using JavaScript array syntax, you could reference the element by way of numeric or named array indexes, as in:

```
document.forms[0].elements[0]
```

or:

```
document.forms["formName"].elements["controlName"]
```

or any combination of reference types:

```
document.forms[0].controlName
```

Given the initial purpose of scriptable browsers—primarily for client-side form validation and dynamic navigation—the limited object model was sufficient. Despite its limitations, DOM Level 0 from Navigator 2, Navigator 3, and IE 3 was intriguing enough to attract a wide audience hungry for more flexibility.

Microsoft IE 4 DOM

In advance of the W3C's DOM activity, Microsoft boosted the powers of IE 4 to allow scripts to modify any piece of a document's content after the page had loaded. The key to this feature was automatic reflow of the page to accommodate changes in an element's dimensions due to the scripted change.

With the luxury of being able to render any modified content on the fly, it was meaningful to expose all HTML elements as scriptable objects for the IE 4 DOM. What scripters needed, however, was a quick way to reference those elements without having to take the structure of the document into account. A property of the IE 4 `document` object (or any other container object for that matter) flattened the hierarchy of nested elements so that as long as the script had the `id` of the desired element, the `document.all` array (collection) offered instant access to the element in multiple syntax approaches, such as:

```
document.all.elementID
document.all("elementID")
document.all["elementID"]
```

In addition to this referencing syntax, the IE 4 DOM empowered all elements with properties and methods that facilitated the reading and writing of plain text and HTML content. You could, for example, insert some HTML inside an element by first assembling a string of HTML tags, attributes, and content, and then assigning that string to the `innerHTML` property of an existing element. The inserted content got rendered instantly, as the page reflowed to adjust for the inserted HTML. If the content contained no tags, you could assign the string to the element's `innerText` property.

The nested structure of HTML elements also played a role in the IE 4 DOM. The notion of parent and child elements followed traditional paths. An element's container was its parent element; an element nested inside another was a child of the outer element. The element was king, and an element contained either just plain text or additional HTML elements.

W3C DOM Architecture

The W3C DOM working group based its architecture on an object type that is more fundamental than the element: the *node*. The concept behind this is that a document and its contents can be diagrammed, in a sense, as a hierarchy of items—nodes—of different types. Some nodes are containers (branches from which other nodes hang), while others are self-contained (leaves at the ends of branches).

The root node of a document is the *document node*—the master container of all content. In an HTML document, an item denoted in the source code by a tag is an *element node*. Text content between a matched start and end tag pair is a *text node*. Other types of nodes also occur in a document, such as *attribute nodes*, *comment nodes*, and *document-type nodes*. To visualize the basic node arrangement, consider the following HTML for a simple document:

```
<!DOCTYPE ... >
<html>
    <head>
    </head>
    <body>
        <p>A simple paragraph.</p>
    </body>
</html>
```

Figure I-1 shows a diagrammed version of the node hierarchy of this document.

The tree structure shown in Figure I-1 lends itself to describing relationships among elements in parent-child terms. For example, the `document` node in the illustration has two child nodes, the `DOCTYPE` and `html` elements. These two child nodes are siblings to each other. The `html` element node is, itself, also a parent to the `head` and `body` nodes. (Note that some W3C DOM clients, notably Mozilla-based browsers, also treat line breaks in HTML source code as separate text nodes, not shown in Figure I-1. Examples later in this book take this behavior into account.)

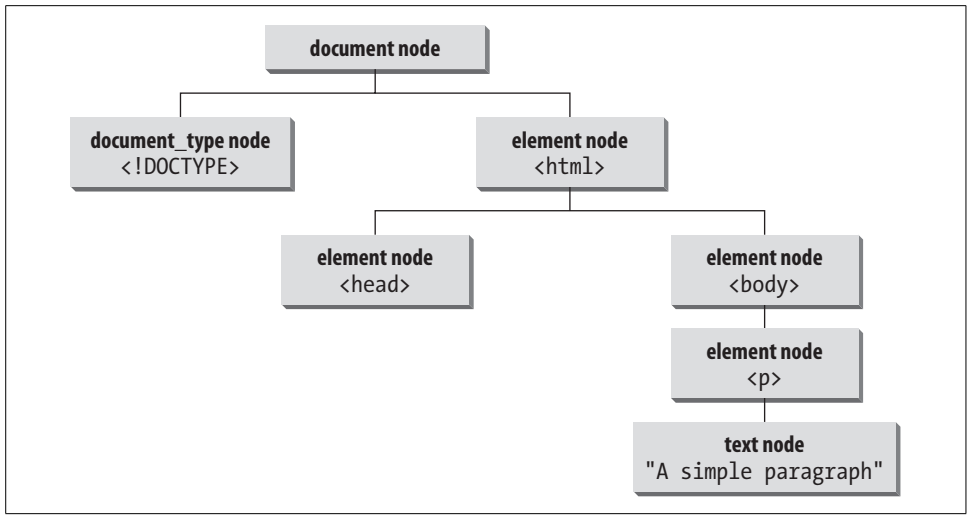


Figure I-1. A simple document's node structure

As you can see, the parent-child relationships are not conceptually different from the containment hierarchy that well-formed HTML documents exhibit. In fact, the IE 4 DOM, at least at the element level, treats relationships the same way. But when it comes to using the W3C DOM to reference elements and what you see in the source code as text content, the approach is different from any DOM that came before it.

In place of the IE 4 `document.all` collection, the W3C DOM implements a root `document` object method, `document.getElementById( )`, that dives through the entire document in search of an element whose `id` attribute value matches the parameter of the method. For example, if a paragraph element's `id` attribute is `myP`, a script can reference the paragraph as follows:

```
document.getElementById("myP")
```

The method returns a reference to the element object, whose properties and methods may be invoked as needed. Unlike the IE 4 DOM, however, a W3C DOM Level 2 element object has no direct knowledge of its child nodes, other than the fact that they exist. For example, the text content of a W3C DOM `p` element is not a property of the `p` element object. In other words, W3C DOM Level 2 provides no analog to the `innerText` or `innerHTML` properties of the IE 4 DOM. To read or modify the content of an element's content using standard syntax, your scripts must work with the child nodes of the element.

An element object has numerous properties and methods that provide scriptable access to information about its parent, child, and sibling nodes. These properties and methods become the primary gateways to reaching an element's nested nodes. In turn, every node has properties that reveal its type (via a constant value for each type) and value. If a child node is of the text node type, you can read or write the

string value of that text node as a property of that text node. If the content consists of a mixture of text and other HTML elements, you must use other W3C DOM facilities to create and accumulate element and text nodes before inserting them into an existing element. In this model, element tags are not represented as string content of a document, but as objects.

Although the modularization of the W3C DOM offered an opportunity to add the IE DOM's `innerText` or `innerHTML` convenience properties to HTML element objects, this did not occur in the HTML module for Level 2. Drafts of DOM Level 3, however, describe a new node property—`textContent`—that simulates `innerText`.

An Object-Oriented Model

The parent-child relationship between nodes in a document depends solely on the nesting and source code order of the tags and related content. This kind of containment hierarchy, however, does not necessarily imply inheritance of DOM properties from parent to child. If you assign a value to the `id` attribute or property of a `table` element, nested rows do not inherit the parent element's ID. The closest that inheritance touches HTML elements is in the style sheet area, where most style properties are inherited by an element's child elements (but CSS inheritance is governed by rules that have nothing to do with object orientation).

The W3C DOM, however, is built upon an abstract model that looks very object oriented. The abstract model defines what constitutes a node, an element, and so on. All content-related objects are derived from the `Node` object, defined in the Core module. This fundamental building block comes with nearly three dozen constants, properties, and methods that define what a `Node` is and what it can do. More specific kinds of nodes, such as the `Document`, `Element`, `Attr`[ibute], and `Text` nodes, inherit all of the `Node` object's constants, properties, and methods, adding properties or methods that are appropriate for whatever kind of node it is. Carrying this inheritance further to the DOM's HTML module, the `HTMLElement` inherits from the generic `Element` object, and each tag-specific element (such as `HTMLBodyElement` or `HTMLInputElement`) inherits from the `HTMLElement` object.

You can also find places in the W3C DOM abstract model where object "classes" act as interfaces to other objects. For example, in the Events module, an important object is called the `EventTarget`. Any node can be a target of an event. Thus, the DOM specification implies that all node objects should implement the `EventTarget` interface.

None of this abstract model construction impacts the way your scripts reference objects within a document. But it does help explain why all HTML element objects share such a large number of properties and methods (inherited from the Core module's `Node` and `Element` objects), as shown in the beginning of Chapter 2 in *Dynamic HTML, Third Edtion*.

Mozilla-based browsers implement a substantial portion of the W3C DOM Level 2 specification, with Mozilla 1.8 implementing some Level 3 Core module features. Support in IE was sketchy in IE 5 (Windows), but IE has gradually embraced more—but certainly not all—of the Level 2 specification through Version 7. In other words, IE 5 and later support both the Microsoft proprietary DOM (from IE 4) and significant portions of the W3C DOM. As of IE 7, however, Microsoft has not implemented the W3C DOM modules governing events and text ranges. Online Sections IV and VI cover the current W3C and IE DOM implementations, while Chapter 2 of of *Dynamic HTML: The Definitive Reference* provides a complete cross-DOM reference.

Web API

Additional working groups in the W3C are tackling the jobs of writing recommendations for items of interest to DHTML developers and browser makers, but which lie outside established realms of document markup and object models. The Web API working group—a subset of the W3C Rich Web Clients Activity—is essentially documenting key items that are implemented in mainstream browsers as de facto standards (albeit sometimes with extensions).

Two recommendations—the Window Object and XMLHttpRequest Object—are fundamental building blocks of a great deal of today's DHTML activity. The `window` object, in particular, has been implemented since Day One of scriptable browsers. But because there is no existing markup that addresses the browser window (except perhaps the frame), its details are not a part of the DOM specification. These Web API efforts are plugging gaping standards holes for all DHTML developers.

Web Accessibility Initiative (WAI)

An important W3C activity that frequently escapes the notice of content developers addresses accessibility of web content by users with a variety of disabilities. The Web Accessibility Initiative (*http://www.w3.org/WAI*) publishes recommendations for both technology producers and content authors. Because overuse of DHTML techniques can render a site's content unavailable to users with mobility, vision, hearing, or reading disabilities, authors should be aware of the guidelines while a site or application is still in the design stage. In truth, many of the existing HTML, CSS, and DOM standards include features aimed at improving accessibility. Until recently, mainstream browsers tended to assign low priority to implementing accessibility features. Do not, however, consider that a license to abuse DHTML at the expense of a sizable potential audience for your content. In some countries, web sites representing organizations of legislated minimum sizes may be legally obligated to ensure that they are accessible to all visitors. All web content and application developers should

read Roadmap for Accessible Rich Internet Applications (*http://www.w3.org/TR/aria-roadmap/*).

Web Hypertext Application Technology Working Group (WHATWG)

Impatient with the progress of the W3C in reaching agreement about standards for rich Internet applications, several industry participants started a separate group to develop standards that extend HTML for practical usage. The Web Hypertext Application Technology Working Group (*http://www.whatwg.org*) has created a specification called Web Applications 1.0 (sometimes informally referred to as HTML 5). Although the specification is still in working draft stage, several parts are sufficiently finalized to have been implemented in a few recent browser versions.

As the name implies, Web Applications 1.0's goal is to define new or improved features built into browsers that make the job of creating rich Internet applications easier for the developer. The standard builds on existing HTML, XHTML, and DOM standards to add features such as self-validating forms (codified under the banner of Web Forms 2.0), user-modifiable tables (e.g., adding rows to a dynamic order form), dynamically drawable canvas regions, script-controllable audio, and more. The `canvas` element, for example, is now built into Mozilla, Safari, and Opera browsers, while support for Web Forms 2.0 and the `Audio` object exist in Opera 9.

ECMAScript

When Navigator 2 made its debut, it provided built-in client-side scripting with a new language called JavaScript. Despite what its name might imply, the language was developed at Netscape, originally under the name LiveScript. It was a marketing alliance between Netscape and Sun Microsystems that put "Java" into the JavaScript name. Yes, there are some striking similarities between the syntax of JavaScript and Java, but those existed even before the name changed.

Internet Explorer 3 introduced client-side scripting for that browser. Microsoft provided language interpreters for two languages: VBScript, with its syntax based on Microsoft's Visual Basic language, and JScript, which, from a compatibility point of view, was virtually 100% compatible with JavaScript in Navigator 2. The name variation had more to do with licensing and corporate politics than it did with programming syntax.

It is important to distinguish a programming language, such as JavaScript, from the document object model that it scripts. It is too easy to forget that document objects are not part of the JavaScript language, but are rather the "things" that programmers script with JavaScript (or VBScript). The JavaScript language is actually more mundane in its scope. It provides the nuts and bolts that are needed for any program-

ming language: data types, variables, control structures, and so on. This is the *core* JavaScript language.

From the beginning, JavaScript was designed as a general-purpose language that could be applied to any object model, and this has proven useful. Adobe Systems, for example, uses JavaScript as the core scripting language for many of its imaging and publishing products. The same core language you use in HTML documents is applied to completely different object models in the Adobe programs.

To head off potentially disastrous incompatibilities between the implementations of core JavaScript in different browsers, several concerned parties (including Netscape and Microsoft) worked with a European computer standards group now known only by its acronym: ECMA. The first published standard, ECMA-262 (*http://www.ecma.ch/stand/ecma-262.htm*), also known as the politically neutral ECMAScript, is essentially the version of JavaScript found in Navigator 3. A second edition of the standard repaired small errors in the first edition. But the third edition added some new language features that had already been implemented in cross-browser compatible fashion. That version of ECMAScript has essentially been a part of mainstream browsers since 1998. More recent browsers implement features of the core language that will find their way into future versions of the standard.

The language continues to evolve and even branch into new areas. For example, Mozilla 1.8 was the launchpad for support for a related standard, ECMA For XML, or E4X for short. These extensions simplify working with XML data within scripts by filling in some of the practical gaps left by the DOM standard.

The good news for scripters is that browsers such as IE, Mozilla, Safari, and Opera display a high degree of cross-browser compatibility with respect to the core ECMA-based scripting language. After the dissonance in the object model arena, it is comforting for web authors to see so much harmony in the core language implementation. For the objects and syntax of the core JavaScript language, see Chapter 5 of *Dynamic HTML: The Definitive Reference*, Third Edition.

De Facto Standards

As helpful as the W3C standards efforts are, they don't always provide the full range of features that could be helpful to the daily work of client-side programmers. This is especially true of the DOM. Content authors and scripters got accustomed to numerous practical features that Microsoft had built into the IE 4 DOM, such as the `innerHTML` property of element objects mentioned earlier. When it appeared that the W3C had no interest in adding much-appreciated properties to its DOM recommendation, browser makers other than Microsoft adopted the features on their own. Such features are now so prevalent in Mozilla, Safari, and Opera that they have become de facto standards that cross-browser developers rely on.

This development became crucial for the widespread adoption of an object known as the `XMLHttpRequest` object (prior to the W3C stepping in). First available in IE 5, this object allows scripts to communicate with servers asynchronously (i.e., silently in the background and independent of other running scripts). Data returned from the server (typically in XML format) could then be extracted with JavaScript and inserted into the current web page view without having to reload the whole page. Outside of Microsoft, first Mozilla, then Safari, and more recently Opera implemented this object so that those browsers now participate in functionality that has been coined Ajax (Asynchronous JavaScript with XML). In a rare case of a de facto standard influencing its originator, Microsoft made this object a native (non-ActiveX) object in IE 7 so that scripters could use the same syntax across browsers to create an instance of the object.

A Fragmenting World

As you will see throughout this book, implementing DHTML applications that work equally well in Internet Explorer, Mozilla, Safari, and other modern browsers can be a challenge unto itself. Understanding and using the common-denominator functionality among the various pieces of DHTML will lead you to greater success than plowing ahead with a design for one browser and crossing your fingers that things will work in the others.

It is equally hazardous to use the W3C, ECMA, and other standards documents as your sole guides to implementing DHTML features in your applications. Browser support for every last detail of a standard is uneven at best—more so if visitors to your pages use browsers even one generation old.

If the inexorable flow of new browser versions, standards, and authoring features teaches us anything, it is that each new generation only serves to fragment further the installed base of browsers in use throughout the world. While I'm sure that every reader of this book has the latest subversion of at least one browser installed, the imperative to upgrade rarely trickles down to all the users of yesterday's browsers. In fact, many corporate users are prohibited by their IT departments from upgrading or changing browers. If you are designing web applications for public consumption, coming up with a strategy for handling the ever-growing variety of browser versions should be a top priority. It's one thing to build a DHTML-based, context-sensitive pop-up menu system into your pages for IE 7 users when that's the browser you use. But what happens to users who visit with IE 5/Mac, Firefox, Safari, or Opera 8, or a pocket computer mini-browser, or Lynx?

There is no quick and easy answer to this question. So much depends on your content, the image you want to project via your application, and the browsers used by your intended audience (analysis of server logs can help here). If you set your sights too high, you may leave many visitors behind; if you set them too low, your competi-

tion may win over visitors with more engaging content and interactivity. Or, you could find yourself in the ideal situation: designing applications aimed at a single, organization-wide browser version.

It should be clear from the sheer size of *Dynamic HTML: The Definitive Reference* that those good old days of flourishing with only a few dozen HTML tags in your head are gone forever. As much as I'd like to tell you that you can master DHTML with one hand tied behind your back, I would only be deceiving you. Using DHTML effectively is a multidisciplinary endeavor. Perhaps it's for the best that content, formatting, and scripting have become separate enough to allow specialists in each area to contribute to a major project. For example, I've been the scripter on many such projects, while other teams handled the content and design. This is a model that works, and it is likely that it will become more prevalent, especially as each new browser version and standards release fatten the following pages in the years to come.

Cross-Platform Compromises

With the World Wide Web Consortium's XHTML, CSS, and DOM standards activities having been on the radar screens of browser makers for so many years now, developers might expect all mainstream browsers to adhere to a sizable chunk of those standards. If that were the case, the benefit to the DHTML developer would be significant: writing one set of code that works on a vast majority of browsers. Indeed, some browser makers have gone to great lengths to implement as many details of the W3C standards as possible. Mozilla-based browsers tend to lead the pack in this regard, but Apple's Safari and the Opera browser are no slackers. In an ironic twist, the latest version of the browser that pioneered the notion of dynamically altering any piece of an already-loaded HTML document, Microsoft Internet Explorer 7 for Windows, lags behind supporting some key standards. At the same time, however, Microsoft also pioneered several features that have proven so popular with developers that they have become de facto standards in other browsers. This chapter begins by comparing categories of DHTML features that are available in a wide range of modern browsers (plus or minus implementation bugs) against important proprietary features with which virtually all DHTML developers must contend. It also explores some overall strategies that you may wish to use for DHTML applications that must run identically across multiple browsers, as well as suggestions on how to accommodate less-capable browsers.

What Is a Platform?

The term *platform* has multiple meanings in web application circles, depending on how you slice the computing world. In its broadest sense, a platform typically denotes any hardware and/or software system that forms the basis for further development. Operating system developers regard each microprocessor family as a platform (Pentium or PowerPC CPUs, for example); desktop computer application developers treat the operating system as the platform (each Windows generation, Mac OS X, Unix, Linux, and the rest); peripherals makers perceive a combination of

hardware and operating system as the platform (for example, a Wintel machine USB 2.0 port or an IEEE 1394 "FireWire" bus).

The universal acceptance of web protocols, such as HTTP, means that a web application developer doesn't have to worry about underlying network transport issues. Theoretically, all client computers equipped with browsers that support web protocols—regardless of the operating system or CPU—should be treated as a single platform. The real world, however, doesn't work that way.

Today's crop of DHTML-capable web browsers are far more than data readers. Each one includes a content rendering engine (either open-source or proprietary), one or more scripting language interpreters, security access mechanisms, asynchronous server data posting and retrieval capabilities (part of a technology commonly called AJAX), and optional connections to related software modules for media playback, Java applets, and the like. The instant you decide to author content that will be displayed in a web browser, you must concern yourself with the capabilities built into the browsers used by visitors. Despite a certain level of interoperability due to industry-wide standards, you must be ready to treat each major browser engine—and sometimes each version of each browser—as a distinct development platform. Writing content to the scripting API or HTML tags known to be supported by one version of a browser does not guarantee support in other browsers or versions. Unlike the "old days," a browser's brand name is less indicative of its platform than is the engine running inside that browser. For instance, the same engine powers browsers named Mozilla, Firefox, Netscape, and Camino.

If you are creating content, you must also be aware of differences in the way some browsers tailor themselves to each supported operating system. For example, even though the HTML code for embedding a clickable button inside a form is the same for any forms-enabled browser, the look of that button may differ when rendered in Windows, Macintosh, and Unix versions of a particular browser. That's because some browser makers observe the traditions of the user interface look and feel for each operating system. Thus, a form whose elements are neatly laid out to fit inside a window or frame of a fixed size in Windows XP may be aligned in an undesirable way when displayed in the same browser on a Macintosh or a Unix-based system.

Despite the potential hassle of distinguishing platforms down to the microscopic level, today's DHTML developer has to be aware of essentially two overall platforms: standards-compliant and Microsoft proprietary. It's not an either-or proposition. In fact, if you intend to have your DHTML content run on a wide range of modern browsers, you will be forced to blend the two platforms together by applying as much standards-compliant code as Internet Explorer can handle, and then provide accommodations for the IE proprietary features.* With this blending in

* Developers who prefer a more Microsoft-centric view of their development would reword this sentence to imply that when other browsers don't support IE-only proprietary features, the developer should write the code to accommodate the other (standards-compliant) ways.

mind, the next sections compare standards-compatible and Microsoft DHTML feature categories.

Standards-Compatible DHTML

The roster of well-known browsers that unreservedly aim for standards compatibility is a long one. There is scarcely a Web developer alive who hasn't heard of brand names such as Mozilla, Firefox, Netscape, Safari, and Opera—or at least you should be seeing these names in web server logs and reports. Although current products bearing these five browser brands are built on three different engines (all but Opera's engine being open source), they share a high degree of DHTML compatibility for common tasks.

Historically, it was the Mozilla effort to replace the once-popular Netscape 4 browser with a more modern implementation that led to support in its Gecko layout engine for more HTML, CSS, and DOM features than any other browser at the time (2000). Mozilla's engineers made some difficult, but ultimately correct, decisions to abandon old ideas that didn't catch on and approach the browser's internal mechanisms anew.

Developers making the transition to the new world of the W3C Document Object Model (DOM) had to learn an entirely new vocabulary and way of thinking about the objects inside a document. Those who had learned only the IE-proprietary way of modifying a page's content on the fly had to take notice of the W3C DOM's new abstract object model. Even though some W3C DOM features had been implemented in varying stages of IE during the Version 5 and 5.5 lifetimes, content authors had little imperative to switch over, because the Microsoft model was the only one needed to work with IE's DHTML capabilities. Even Opera at the time adopted IE's basic DHTML model. But with the W3C DOM being the only route available for scripting Mozilla's Gecko engine and the perceived "correctness" of following standards, developers had to get to know the terminology and concepts that had not existed prior to the W3C standard.

Simple Element Object References

Perhaps the most vital activity that DHTML scripts perform is modifying the content or appearance of an HTML element object in the current document. The specific way to reference an element depends on how much information your script knows about the desired element. Ideally, an element has an identifier assigned to its `id` attribute. Armed with that information, your script can use the core `document` object method whose sole parameter is a string identifier for the desired element:

```
document.getElementById("elementID")
```

This method slices through the node containment hierarchy of the document's content (as discussed at length in Online Section V), and returns a reference to the first element object in source code order whose `id` attribute value matches the method's argument. This syntax is also implemented in IE 5 and later.

It's unfortunate that a method that is likely to get a lot of use in scripts is so long and difficult to type (observe the case of each letter). Some scripters include a helper function in every page whose name is short and sweet, allowing repeated access to the method to save some bytes along the way:

```
function getEl(elemID) {
   return document.getElementById(elemID);
}
```

One other DOM object method can come in handy if you need to reach a series of elements that share the same tag:

```
document.getElementsByTagName("tagname")
```

This method returns an array consisting of references to each element whose tag matches the name supplied as an argument. Importantly, this method can be applied to any HTML element object, whether or not it acts as a container of other elements. Thus, you can use the method to obtain, for example, references to all `p` elements inside a `div` container whose ID is `sidebar`:

```
var elemList = document.getElementById("sidebar").getElementsByTagName("p");
```

Thereafter, you can loop through the array to obtain a reference to each `p` element in turn.

One other point about the W3C DOM specification: it continues to recognize the contribution of the first scriptable browser DOM, with its limited range of objects, such as forms and form controls. The "old" way of referring to these objects, such as `document.forms[n].elements["controlName"]`, is still valid syntax. In practice, however, you should use only this "Level 0" syntax when scripts need to be backward-compatible with earlier browsers.

Cascading Style Sheets

CSS supporters in the developer community have been vocal in their demands of browser makers to support as much of the CSS2 standard as possible (and in a uniform way—not always an easy proposition, it turns out). Mainstream standards-compatible browsers available in 2006 support all of CSS1, most modules of CSS2, and, in some cases, preliminary features of CSS3. Script access to style sheet properties occurs via an element object's `style` property. This property contains a `style` object whose properties correspond to style sheet properties. Important: the W3C DOM standard specifies that an element object's `style` property reflects only the `style` attribute values of an element, and not style settings made elsewhere in the document. To reach the details of style properties affecting an element, regardless of

their source (e.g., linked in from an external .css file), the W3C DOM provides a somewhat convoluted construct to read what is known as the *computed style*. This feature, implemented starting with Mozilla 0.9.2, Safari 1.3/2.0, and Opera 7.5, is demonstrated in Online Section IV.

Positioning and Layering

Although CSS-P (positioning) was not strictly a part of the CSS specification until CSS2, the properties associated with this capability have been built into mainstream browsers since the days of Netscape 4 and IE 4. These and later browsers support style sheet properties that place elements in their own layers above the body content. An inline style rule that pulls a graphic out of the rendering sequence of a page and positions it to a specific spot within a document looks as follows:

```
<img src="myFace.jpg" height="60" width="40" alt="Nancy's Mug"
style="position: absolute; left: 200px; top: 100px">
```

Positioning properties include facilities for hiding and showing an element, as well as controlling the stacking order of the layers.

In standards-compatible browsers, dynamic positioning that acts in response to user actions must use the W3C DOM event model, which is different from the IE event model. Even so, the two can be made to work together without too much difficulty, as described in Online Section VI.

Dynamic Content

Drafters of the W3C DOM standard produced a system that provides a high level of conceptual consistency for the way scripters modify portions of a document's content. This was done, however, at the expense of simplicity (compared to the ways scripters were accustomed to in Internet Explorer's proprietary object model).

The easiest content to modify is text that is contained by an element. Such text is represented in the DOM as the value of a *text node* nested with the element, and is handled through strings in JavaScript. But when it comes to modifications involving elements, the W3C DOM approach gets a bit wordy. To create a new HTML element and its content in pure W3C DOM syntax requires the following sequence:

1. Create an empty element for the desired tag with the `document.createElement( )` method.
2. Assign values to its individual attributes one at a time, preferably via the element's `setAttribute( )` method.
3. Create a text node for the element's content with `document.createTextNode("newtext")`.

4. Use a variety of node methods to construct the node tree of the new element and its content.
5. Use another method to insert the new node group into a position within the document's existing node tree.

If the content your scripts need to generate has lots of elements and text nodes, the sequence requires many more statements. The concept of creating an empty object, populating its attributes or properties, and then inserting the object into its rightful place permeates the W3C DOM, and not only for document content. The phrase "Create, Populate, Insert" will become your mantra.

Engineers at Mozilla and elsewhere recognized, however, that developers found some Microsoft proprietary DOM features to be very convenient. As a compromise to practicality over blind adherence to the standards, most modern browser engines implement the IE `innerHTML` property for any element. This allows scripts to assemble new content as if it were a string of HTML source code to be inserted where desired. Online Section Vwill compare these approaches in detail.

The XMLHttpRequest Object

In response to the popularity, utility, and broad browser support of Microsoft's `XMLHttpRequest` object (originally implemented in IE as an ActiveX object), the W3C opened up a working group to standardize the operation of this object. This object, at the core of what has become a new generation of web-based applications (e.g., Google Maps and highly interactive web email pages), allows a web page to communicate with a server asynchronously in the background. That is, after the page has loaded, a script can request subsequent data from the server (and submit data, as well) without interfering with the content of the current page. Data returned from the server (which can be in XML format) can then be parsed by scripts to modify existing content on the page.

Mozilla-based browsers were the first outside of IE to add this object for mainstream browsers. Safari 1.2 and Opera 8 eventually followed. Because all of these implementations operate on non-Windows operating systems, the `XMLHttpRequest` object is a full global object, rather than an ActiveX object. In fact, Microsoft even went so far as to build the object into its global objects for Internet Explorer 7, thus allowing the same basic code to work across browsers.

The Event Model

Events are the critical bridge between user action and scripted activity. Virtually every element object has events that can be scripted to respond to user and system actions. For example, it is possible to associate different actions with user clicks over different headings (even if the text blocks don't look like links) by assigning a different script statement to each heading's `click` event.

The W3C DOM Events module introduced some new terminology for scripters already experienced with DHTML scripting in earlier browsers. Elements are instructed to respond to a type of event by assigning an *event listener* to it, meaning that scripts instruct elements to "listen" for events of particular types, such as mouse clicks, key presses, and so on. When an element "hears" that event type, processing shifts to a function, just like the event handler functions you are used to.

An `event` object contains numerous properties about the details of the current event being processed. An event listener's function receives the `event` object as a function argument.

An important aspect of the event model you need to understand is *event propagation*. An event, unless otherwise instructed by script, continues to "bubble up" through the HTML element containment hierarchy of the document. Consider the following simple HTML document:

```
<html>
  <body>
    <div>
      <p>Some Text:</p>
      <form>
        <input type="button" value="Click me" onclick="alert('Hi!')">
      </form>
    </div>
  </body>
</html>
```

When the user clicks on the button, the `click` event is first processed by the `onclick` event handler in the button's own tag. Then the `click` event propagates through the `form`, `div`, and `body` elements. If the tag for one of those elements were to have an `onclick` event handler defined in it, the `click` event from the button would trigger that handler, too. Event bubbling can also be programmatically canceled at any level along the way.

Events also trickle downward through the hierarchy in the W3C DOM event model. To process an event on its way to its actual target, however, a script must instruct an event listener to *capture* the event. Online Section VIdescribes event propagation in more detail.

One area in which the W3C DOM model is much more conservative than the IE model is in the breadth of event types. Because the W3C model is not operating-system-dependent, it has so far settled on a basic set of events that let scripts work with common mouse and system events. The module specifying keyboard event details in DOM Level 3 has not been finalized as of late 2006, but browsers support a preliminary version that was originally part of Mozilla.

Operating System Support

It's not uncommon today to find browser engines designed outside of operating system companies to be written such that the same underlying engine is used for all OS versions of the browser. Such is the case with Mozilla's Gecko engine and Opera's Presto engine. All operating system versions of Firefox, for instance, are derived from the same core browser-engine code base. A principal benefit of this approach is that all DHTML-related rendering and activity tend to operate identically, regardless of operating system.

Another aspect of this consistency is that web page user interface elements are not as operating-system-dependent as previous versions. Designs for buttons and other standard UI elements are not controlled entirely by the operating systems, but, in the case of Mozilla-based browsers, rather by definitions associated with the current theme (or "skin") in force at any moment. Default buttons, for instance, generally render with the same dimensions and proportions in all operating system versions. Unlike the earlier days of graphical user interfaces on personal computers, the multiplicity of web designs seems to have reduced the clamor for absolute UI consistency across applications for a given operating system (Macintosh users were particularly sensitive to this type of consistency). Today, as long as users can distinguish a checkbox from a radio button, or intuitively detect a clickable button, the design passes muster. The upside for web developers is that pages using standard elements are more likely to resemble each other on all operating system platforms.

Internet Explorer DHTML

The browser that first inspired extensive dynamic content was Microsoft Internet Explorer 4. Although they are considered basic features of today's DHTML browsers, two groundbreaking characteristics of IE 4 fired developers' imaginations at the time:

- Exposing practically every HTML element as a scriptable object
- Automatically reflowing the page after content modification

In the absence of an industry standard for its document object model during IE 4's development, Microsoft invented its own model, along with a vocabulary of objects, properties, methods, and object collections (arrays). Then, as the radically different W3C DOM recommendation started taking shape, and with so many scripts "in the wild" relying on the IE 4 model and syntax, Microsoft faced the unenviable task of blending the W3C model into its IE 4 model.

The transition began slowly in IE 5 (although the independently developed IE 5 for the Macintosh embraced more of the W3C DOM from the start) and gradually picked up the pace through Version 6 (with little new in the DOM department for Version 7). This means that for many object-scripting tasks in IE-only applications, you have your choice of the Microsoft or W3C DOM approach in your code—an

Navigator 4 DHTML—A Dim Memory

Designed well before the first CSS and DOM standards crystallized, Netscape Navigator 4 became a victim of a sea-change in approaches to style sheets and document object models. Its efforts to bring JavaScript syntax to style sheet rules, to add the `<layer>` tag to the HTML vocabulary, and to place the bulk of a page's dynamism into a separate `layer` object all failed to gain acceptance in the W3C working groups. At the last minute, some of the nascent CSS language made it into Navigator 4, but it was fairly fragile when applied to complex pages involving tables and forms.

For the brief time period during which Navigator 4 was the only DHTML game in town, the lack of standards support was not a problem. But when Microsoft released Internet Explorer 4, with its radically different object model approach, developers had to jump through hoops to write DHTML code that accomplished the same tasks in both browsers. Even then, the power of dynamic page reflow in IE 4 made many DHTML effects essentially impossible to duplicate in the mostly static body content of Navigator 4 pages.

A lot of JavaScript code tailored to Navigator 4 runs in no other browser. In other words, Navigator 4 has become a dead-end development platform, whose installed base is tiny and only getting smaller. Even so, some organizations still standardize on Navigator 4, requiring that some developers acknowledge the browser in their development plans. Although reference sections of this book still supply Navigator 4-specific data for the sake of completeness, consult the first edition of this book if you need detailed coding examples for that platform.

enormous syntactic palette from which to choose. As yet, there are no signs of Microsoft deprecating its own features. If you choose the W3C DOM route as your primary focus, however, you'll find that Microsoft has so far elected to bypass some modules (as described shortly), forcing you to use the Microsoft syntax for vital services, such as events. On the flip side, if you start your DHTML authoring life exclusively in the world of IE for Windows, you will find some features not available in other browsers, including IE for the Mac. This will lead to pages and applications that won't perform as expected when you expand your audience to other browsers.

Element Object References

All IE browsers starting with Version 4 (including those for the Mac) implement the `document.all` collection, which provides a gateway to any element for which you have assigned an identifier to the `id` attribute. Statements that refer to elements can reference the element ID as either a property reference or string, as in the following forms:

```
document.all.elementID
document.all("elementID")
```

```
document.all["elementID"]
document.all.item("elementID")
```

The string versions are helpful when you define generic functions that receive an ID string as an argument, allowing a single statement to reference any element.

IE also lets you omit the `document.all` part of a reference so that you can reference an element simply by its ID. Although this practice makes for compact code, it also makes it very difficult to go back to the code to find statements that reference elements. This capability (also supported in Mozilla 1.7 only in quirks mode, Safari 1.3/2.0, and Opera) has another potentially dangerous side effect: Because element ID names become global object identifiers, it means that you cannot use any element ID on the page as a scripting identifier for variables, functions, arrays, or other objects defined in the global space.

Unless you must explicitly support IE 4, you should avoid the `document.all` reference format in your DHTML code. The W3C DOM `document.getElementById()` method is supported by an overwhelming majority of scriptable browsers in use today, including IE 5 and later. Most code examples in this book assume the W3C DOM style of element object referencing.

Cascading Style Sheets

Some CSS functionality was introduced in IE 3, but IE 4 was considered the first browser to make a serious attempt at supporting the CSS Level 1 standard. Even so, the support was far from bug-free. Microsoft claims full CSS1 support for IE 6 and substantial CSS2 support for IE 7. Web developers frequently criticized Microsoft for its CSS implementations through IE 6. In response, the IE 7 team focused intently on CSS compatibility.

Script access to CSS properties occurs via an element object's `style` property. This property contains a `style` object whose properties correspond to CSS style sheet properties. Element object properties provide scripted access to the element's attribute values, and the `style` property is no different. In other words, the `style` property reports only those values assigned to an inline style sheet rule. To read the actual style being applied to the element (from a style sheet defined in the head or imported from an external file), IE 5 and later provide a proprietary `currentStyle` property for all element objects.

Dynamic Content

Rendering engines in IE 4 and later respond quickly to changes in content, by reflowing the page after any such change (without reloading the page from the browser or cache). Regardless of your choice of element referencing syntax, the DOMs in IE provide properties and methods that allow adding, removing, or modifying content within an element or whole elements and their content. Starting with IE 5, you have

your choice of using the IE 4 DOM or W3C DOM properties and methods for modifying elements and their content. The original Microsoft approach was to treat an HTML document as a character string that could be modified by inserting, deleting, or replacing chunks of text (including tags, if necessary) within the document. Some developers continue to find it easier or more convenient to use IE's string manipulation properties (such as `innerHTML`) to modify elements and their content, rather than the W3C DOM node notation. In fact, the `innerHTML` property has become a de facto standard that is supported by most DHTML-capable browsers, including Safari, Opera, and those based on Mozilla. Be prepared, however, to have your code criticized by developers who prefer a purer implementation using only W3C DOM syntax and node concepts. Also be aware that excessive string manipulation tends to be a performance hog in JavaScript.

The XMLHttpRequest Object

Internet Explorer 5 offered the first opportunity for a web page to retrieve XML data from a server, and make the XML document available to scripts—all in the background. For IE 5, 5.5, and 6, the capability was loaded into the document as an ActiveX object. Once an instance of the object was created, scripts could send data to a server (with GET or POST methods), including data in the form of URL extensions or separate data (e.g., a SOAP call consisting of XML). Asynchronous interactivity with a server, coupled with a fully dynamic document object model, allowed for the creation of web pages that more closely simulated working with a standalone application. For example, if a user wants to "drill down" to view more details about an item in a table, a script can request just that little extra data, receive it in XML form, and let a script quickly display that data in a pop-up box or expanded table cell. No more retrieving an entirely rewritten page just to get a tiny bit of updated data.

Other browser makers implemented non-ActiveX versions of the object, calling it the `XMLHttpRequest` object. Microsoft even added support for the native object to IE 7. Fundamental `XMLHttpRequest` object operations work with the same syntax across all supporting browsers once the instance of the object is created. See Online Section VII for more details on working with the `XMLHttpRequest` object.

The Event Model

Microsoft was certainly entitled to develop its own event model for IE 4 because there was no industry standard at the time. But unlike its adoption of W3C DOM modules in other areas, Microsoft—at least through Internet Explorer 7—has refused to implement the W3C DOM event model alongside its proprietary version. Developers have no choice but to code for the Microsoft event model if they want their pages to run in IE.

The IE event model, which works hand-in-hand with the object model, is the critical bridge between user action and scripted activity. Virtually every element object can be scripted to respond to events triggered by user and system actions. IE for Windows, especially starting with IE 5, defines a large number of events that can trigger scripts (as described in Chapter 3 of *Dynaminc HTML: The Definitive Reference*, the Third Edition). Many of these events are patterned after the kinds of events application programmers use for manipulating Windows-based data and user interface behaviors. As a result, many of these events are available only in Windows versions of IE.

A key part of the IE event model is an `event` object. This abstract and short-lived entity—accessed as a property of the `window` object, and thus available to any function processing the event—contains details about each event that occurs. Scripts operating in response to events can inspect properties of the `event` object to determine the element responding to the event, the event's screen position, keyboard key, and so on.

Most events "bubble up" through the HTML element containment hierarchy of the document. Scripts can stop an event from bubbling past any element in the hierarchy.

As you will see in Online Section VI, cross-browser applications must be built to support both the proprietary Microsoft event model and the W3C DOM event model (and, if necessary, the Navigator 4 event model, which shares some features with the W3C model).

Transitions and Filters

Building atop the syntactical conventions of CSS1, IE 4 and later for Windows includes a proprietary style property called `filter` (implemented as an ActiveX module). This property serves double duty. One set of parameters supplies extra display characteristics for certain types of HTML content. For example, you can set a filter to render content with a drop shadow or with its content flipped horizontally. The other set of properties lets you define visual transition effects for when an object is hidden or shown, much like the transition effects you set in presentation programs such as PowerPoint.

Downloadable Fonts

A document to be displayed in IE 4 and later for Windows can embed TrueType font families downloaded from the server. You download the font via CSS style properties:

```
<style type="text/css">
@font-face {
    font-family: familyName;
```

```
        font-style: normal;
        font-weight: normal;
        src: url("someFont.eot")}
    </style>
```

With the basic font family downloaded into the browser, the family can be assigned to content via CSS styles or old-fashioned `<font>` tags.

Note that a font must be available on the server in a special format generated by a "font object" creation tool. The tool for IE is called Web Embedding Fonts Tool (WEFT), which you can download and learn about at (*http://www.microsoft.com/typography/*).

Data Binding

IE 4 and later for Windows provides hooks for ActiveX controls that communicate directly with text files or databases on the server. Elements from these server-based data sources can be associated with the content of HTML elements, essentially allowing the document to access server data without processing a CGI script. IE 5 for the Mac supports this feature when the server data source is a text file (such as a comma-separated or tab-delimited database file). Data binding was, in many respects, a forerunner to the asynchronous server access commonly known today as AJAX. Data binding is, however, specially tailored to accessing server databases directly, whereas AJAX implementations performing the same task typically require further server programming to act as an interface between client requests and the database. While data binding is not covered in depth in this book, I mention it here because it is one of Microsoft's dynamic content features.

Additional Windows-Only Features

IE for Windows is largely a collection ActiveX controls. As such, the browser can take advantage of numerous ActiveX facilities that come with the browser and some that are components of the operating system. That's the case with filters and data binding with remote databases (through ODBC), described previously. Script control of the Windows Media Player is restricted to IE for Windows, as is the capability to turn the browser into a content-editing application and (with the user's permission) access the filesystem. Developers who learn DHTML initially in the IE for Windows environment often fail to understand which capabilities will not translate to other browsers.

Macintosh Versions

The Macintosh applications group at Microsoft (based in Silicon Valley, rather than Redmond, Washington) controlled the development of the IE browser for the Mac, separate from the Windows version. One result of the separate development efforts

was that the IE/Mac browser had been free to embrace more W3C DOM features in its Version 5 than IE 5 for Windows. Eventually, however, newer browsers available for the Mac, including Firefox and especially Apple's own Safari, overtook IE/Mac in their support for W3C DOM. With Apple devoting so much effort to improving Safari internally, Microsoft stopped further development of IE/Mac.

IE/Mac is now rather outdated, but a fair number of Mac users, especially those who cling to Mac OS 9, use that browser. Because of some of its unique behaviorial quirks, that browser can be a problematical one to support with DHTML features. The key point to take away is that IE/Mac was a browser unto itself, and should, therefore, be treated almost as a separate platform—one that needs testing of your DHTML work if you intend to support its users.

Cross-Platform Strategies

The more browser brands, versions, and operating systems you wish to support with your DHTML applications, the greater your challenge to write one code base that works with them all. Before undertaking any project intended for more than a single browser, you must make difficult decisions about not only which browsers to support but also how users of other browsers will be treated by your site. Consumer-oriented e-commerce sites, for example, can rarely afford to turn away even a small percentage of potential customers because the visitors' browsers don't measure up to a lofty design. Specialized or personal sites that are not as concerned about competitive pressures may choose to require browsers of a certain minimum functionality to pass beyond the home page, but requiring only one type of ultra-modern browser or operating system will not win you many friends.

Adding Value with DHTML

An important question to ask yourself about your design goals is whether the DHTML features of your site add value to the content that is otherwise accessible to all, or are those DHTML features essential to the site design. For example, a DHTML-assisted hierarchical menu system adds value by speeding direct access to a nested area of your site, yet users of DHTML-challenged browsers can still reach those areas (albeit with more clicks and intermediate stops en route), and search engine web crawlers will pursue the links. Conversely, if navigation absolutely requires DHTML powers, some visitors will be locked out—not a good thing, and perhaps even illegal if government regulations require that your site be accessible to all visitors. Similarly, search engine web crawlers, which don't execute scripts, will not know to follow links that are rendered only by script. This could reduce the chances that deeper pages of your site will be catalogued and indexed.

As noted in Online Section 1, the standards-based emphasis of HTML markup since HTML 4 has been to separate a document's structure from its presentation details.

Thus, CSS is largely deployed by way of style rules either imported from external *.css* files or specified in the document's head. Even though the `style` attribute exists for most HTML elements, specifying style rules within an element's tag is fraught with maintenance peril and associates presentation too closely with a specific element's structural purpose. On the scripting side, getting the scripting out of markup (e.g., assigning event handlers through means other than, say, `onclick` or `onchange` event handler attributes inside tags) offers key benefits:

- It encourages design of a basic, unscripted page (and essential server support for tasks such as form validation) that conveys vital information for all visitors.
- Initialization routines assign event handlers based on scripted thresholds and criteria under your control (e.g., through object detection).
- After the page loads, scripts can modify the document's elements to add DHTML features (e.g., inserting enhanced links that navigate to areas requiring a scriptable browser).
- It encourages design of reusable, generic script functions and objects that rely on `event` object data to convey references to affected elements, instead of designing functions hard-wired to specific elements.
- It leads to ease of maintenance or upgradability without modifying document markup.

Don't forget that a sizable portion of your audience may be using browsers designed for those with vision and motor skill impairments. And don't forget visitors using cell phones containing modest web browsers. While some of these browsers may support aspects of JavaScript, your DHTML goodies may be of no use to those visitors. This is yet another reason to treat DHTML as an added value proposition to a basic design that works without scripting or DHTML. You may have to build more of your application on the server to provide basic functionality for all visitors, while the client-side DHTML additions (which bypass some of the server stuff) give DHTML-equipped browser users a faster or more interactive experience.

The technique of separating a document's scripted behavior from its structure goes by names such as "unobtrusive" (meaning that DHTML doesn't get in the way of basic content delivery) or "progressive" (meaning that scripting features are added to a page incrementally if the browser supports them). Whatever the label, it is rooted in the notion of using DHTML scripting judiciously to make a solid page even more inviting and engaging for visitors equipped with modern DHTML-capable browsers.

Serving Pages by Browser Type

It's not uncommon for server programs to assemble parts of HTML pages based on information it receives from the `HTTP_USERAGENT` string during the client request. For example, if your site's designer has carefully crafted external style sheet files tailored

to Macintosh and Windows idiosyncracies, then the server can assemble the OS-specific *.css* file URL for the page's `<link>` tag as the page is being served. The browser performs no decision making at all.

Microsoft Conditional Comments

To let page authors include HTML code (including `<link>` and `<script>` tags) for multiple versions of Internet Explorer, yet have sections activate only for specific IE versions, Microsoft devised a system called conditional comments. First supported in IE 5, the system provides a small language whose statements exist only inside HTML comment tags (and sometimes a variation of comment tags). Each statement defines a condition under which the content inside the comment tag is interpreted by the browser. Typical conditions are whether the current IE version is less than, greater than, or equal to a specified number.

As an example, if you wish to have one set of scripts work for all versions of Internet Explorer earlier than IE 7 (including versions 3 and 4) and another set for all browsers beginning with IE 7, you can isolate those sections by way of the following conditional comment sections:

```
<![if lt IE 7]>
<script type="text/javascript" language="JavaScript">
    // statements for pre-IE7 here
</script>
<![endif]>
<!--[if gte IE 7]>
<script type="text/javascript">
    // statements for IE7 and later here
</script>
<![endif]-->
```

Markup inside the first comment (without the expected double hyphens) is interpreted by IE 3 through IE 6, as desired here, whereas the markup in the second comment is interpreted only by IE with major version numbers of 7 or greater. Browsers other than IE/Windows interpret contents of the first comment type, but not the second. Therefore, you could use conditional comments to include markup that is to be interpreted only by a specific IE version (starting with 5) but no other browsers. You can find more details in the `<!--comment-->` element at the end od Chapter 1 in *Dynamic HTML: The Definitive Reference,* Third Edition.

Object Detection over Browser Sniffing

Regardless of the approach you use to accommodate multiple browsers, it will at some point entail code branching or other equalization tactics that are dependent upon the scriptable features of the browser. Back when the matrix of browser versions was small, it was common practice to use browser "sniffing" with the aid of information gleaned from the `navigator.userAgent` and related properties—

information about the browser brand and version. But the matrix of installed browser brands and versions has expanded beyond a sensible number of variations. Except in rare circumstances, browser sniffing has become virtually extinct. In its place is a more viable technique known as *object detection*.

In case you're skeptical about the shortcomings of browser sniffing, it will be helpful to observe how DHTML developers from the Version 4 browser days found themselves in trouble at some point as new browser versions came on the scene. Consider the following typical global variable declaration from the era of IE 4 and Navigator 4:

```
// code unknowingly doomed to failure
var isNav, isIE;
if (parseInt(navigator.appVersion) >= 4) {
     isNav = (navigator.appName == "Netscape");
     isIE = (navigator.appName.indexOf("Microsoft") != -1);
}
```

Hereafter, various functions would branch their code based on the Boolean values of `isNav` and `isIE`, with browser-specific code in each branch. Unforeseen at the time, however, subsequent versions of Netscape abandoned the `layer` object—usually the primary need for branching in the first place back in the Version 4 days. As a result, Netscape 6 (whose `appVersion` reported 5, and is thus greater than or equal to 4) attempted to execute code that it could not handle. On the IE variable side of things, two potential problems loomed, depending on how much IE4-ness the author ascribed to browsers following that branch. For one, the Macintosh version of IE did not implement most IE/Windows-only features. Second, the default preference settings of the Opera browser caused it to identify itself as IE, yet this did not assure compatibility with IE scripts.

Trying to compensate for all browsers past, present, and future requires a huge version sniffing library plus a crystal ball about future browser version numbering and naming. Even attempting such forecasting won't take into account new browsers that crop up, some of which will be built upon very capable existing engines, such as the Mozilla's Gecko engine. Building branched code based on browser version is a losing battle. A superior approach is to branch based on the capabilities of the browser. In other words, your code doesn't care what browser is running; all it cares is that the objects, properties, and methods needed for the next batch of code are supported in the current browser, whatever it may be. Object detection is based on a browser's current capabilities, not hard-wired branding or version numbering.

Object detection is a shortcut name for a technique that verifies the existence of an object, property, or method before using it in a script. The technique isn't new. Scripts that control image rollovers have been using it for years by testing for the presence of the `document.images` array before acting on an image object:

```
if (document.images) {
     // act on image objects here
}
```

All object models that implement img elements as objects support the document.images array. In older browsers, the expression document.images evaluates to undefined, which causes the if condition to fail, so the nested statements don't run. Thus, the scripter is freed from worrying about which specific browsers support the image object.

Implementing object detection on a broader scale can free you from the complexities of today's browser sniffing. For example, a function that switches a style property can work in both the IE 4 and W3C DOM browsers, but requires different referencing syntax for the element. The following function sets the fontWeight style property of an element to bold:

```
function emBolden(elemID) {
     var elem;
     if (document.all) {
          elem = document.all(elemID);
     } else if (document.getElementById) {
          elem = document.getElementById(elemID);
     }
     if (elem && elem.style && elem.style.fontWeight) {
          elem.style.fontWeight = "bold";
     }
}
```

A local variable, elem, is initialized as a null value. The if/else construction looks for the two element reference types that I know have a chance of supporting the style property. The test for document.all is like the earlier example of document.images. Less well-known is that object methods are exposed in most browsers as properties, whose existence can be tested in a similar fashion. Thus, the test for the existence of the document.getElementById( ) method prior to invoking it.

To protect additional script statements from the case of both if/else conditions failing, the balance of the function begins by verifying that elem has a value assigned to it. The tripartite condition is overkill for this specific application, because you can make an educated and safe assumption that any browser that supports either document.all or document.getElementById( ) also supports not only the style property of elements, but also the very common fontWeight style property. But the example is here to demonstrate how to go about verifying the existence of a property when the object or intermediate property may not exist. In the above example, you cannot test simply for the existence of elem.style.fontWeight. A "one-dot" evaluation rule applies to JavaScript, whereby every reference up to the rightmost dot must evaluate successfully in order for the interpreter to see whether the last reference succeeds or fails. If you were to test for the existence of elem.style.fontWeight by itself, and elem was not a valid reference, the script interpreter generates a script error. Evaluation tests of an if condition are conducted from left to right. If any one of the ANDed expressions fails, the condition immediately fails (short circuits), and no further evaluations occur, leaving your browser free from script errors there.

Some browsers, especially older IE, Netscape, and Opera versions, may require more help in evaluating conditional expressions. For these browsers, a value of `undefined` does not necessarily convert to `false` (although the ECMA specification says it should). To obtain the same result, you can use the `typeof` operator to inspect the data type of the object or property:

```
if (elem && (typeof elem.style != "undefined")) {...}
```

A value of `null` does correctly evaluate to `false` for all browsers, so the first test for the existence of `elem`, is fine the way it is. If `elem` exists, the string returned by the `typeof` operator gets compared against `undefined`. If the data type is anything other than `undefined`, processing continues (the test for `fontWeight` is not shown here for the sake of brevity).

Notice, too, that the `typeof` operator helps in those cases when a property exists and its value (perhaps its default value) is either an empty string or zero. Both of these values would cause the conditional expression to evaluate to `false`, even though the property exists. By making sure the property value is either a particular data type or anything other than `undefined`, your condition more accurately reports the presence of the property.

Object detection doesn't solve every compatibility problem, and requires having at hand a good reference of currently-supported DHTML features (such as *Dynamic HTML: The Definitive Reference*). There are times, particularly when designing around known (and now fixed) bugs in earlier browsers, when browser sniffing is appropriate on a small scale. Yet for a great many scripts, object detection can not only ease implementation of incompatible syntax, but also allow older browsers to degrade gracefully by skipping over code that would generate errors.

Whether you elect to use object detection, browser version sniffing, or a mix of the two, you have a choice of several cross-browser deployment strategies: page branching, internal branching, common denominator design, and custom API development. Additional choices you'll make include whether you wish to deny page access to older browsers, provide multiple paths for browsers of different capabilities, or provide just one path that enhances the experience for DHTML features of your design yet degrades gracefully for those browsers without the latest doodads. The following sections describe some of the more popular strategies for accommodating multiple browsers.

Designing for the Common Denominator

From a maintenance point of view, the ideal DHTML page is one that uses a common denominator of syntax that all supported browsers interpret and render identically. You can achieve some success with this approach if you target W3C DOM-capable browsers, but you must be very careful in selecting standards-based syntax that is implemented identically in all such browsers. Because some of these standards

were little more than working drafts as the supposedly compatible browsers were released to the world, the implementations have not been consistent across the board. In other words, just because a browser indicates through object detection that it supports the `document.getElementById( )` method doesn't automatically mean it supports everything else in the W3C DOM.

As mentioned earlier in this chapter, one area that eludes common denominator status when serving to modern mainstream browsers is the event model. That Internet Explorer (at least through version 7) does not support the W3C DOM event model means that you need to be fluent in both models. Online Section VI goes into more detail about blending support for both models into your scripts.

Custom APIs

Once you resolve compatibility issues for frequently-used routines, you'll be glad to never have to do it again. To that end, you should keep in mind the idea of creating one or more libraries of general-purpose functions and/or objects that handle the cross-browser issues for you. In a sense, you'll be creating your own meta language for scripted DHTML operations by writing a set of functions that have terminology you design. Place the functions in a *.js* library file and rely on them as if they were part of your scripting vocabulary. The language and function set you create is called an *application programming interface*—an API. Groups of APIs are sometimes called a *framework*. Example II-1 shows a typical function that works around the very different ways that the W3C DOM and Internet Explorer report the computed style sheet property of an element.

Example II-1. API function to obtain a computed style property

```
// return computed value for an element's style property
function getElementStyle(elemID, CSSStyleProp) {
    var elem = document.getElementById(elemID);
    var styleValue, camel;
    if (elem) {
        if (document.defaultView) {
            // W3C DOM version
            var compStyle = document.defaultView.getComputedStyle(elem, "");
            styleValue = compStyle.getPropertyValue(CSSStyleProp);
        } else if (elem.currentStyle) {
            // make IE style property camelCase name from CSS version
            var IEStyleProp = CSSStyleProp;
            var re = /-\D/;
            while (re.test(IEStyleProp)) {
                camel = IEStyleProp.match(re)[0].charAt(1).toUpperCase( );
                IEStyleProp = IEStyleProp.replace(re, camel);
            }
            styleValue = elem.currentStyle[IEStyleProp];
        }
    }
    return (styleValue) ? styleValue : null;
}
```

The `getElementStyle( )` function of Example II-1 works around the wide disparity in object models for reading the value of a style property assigned in rules located other than in the `style` attribute of the element. The W3C DOM provides an interface for accessing the "view" of the current document. This view object (of the `AbstractView` class) has a `getComputedStyle( )` method, which returns a `CSSStyleDeclaration` object, which contains all computed style property values for an element. To read one of those properties, use the `getPropertyValue( )` method, passing the CSS name of the property as a parameter.

Internet Explorer exposes this information entirely differently. Each element object has a `currentStyle` property, which is an object whose property names are the scriptable names of CSS properties. For example, the CSS `font-size` property is scripted as the `fontSize` property. The branch of the `getElementStyle( )` function that works under IE uses regular expressions and string replacement to change the CSS property name to its scripted equivalent (crossing one's fingers that Microsoft will continue to adhere to the convention of turning hyphenated CSS property names into "lowerCamelCase" words).

With an API function like this in place, you no longer have to worry about these details and discrepancies. Instead, your scripts invoke `getElementStyle( )`, passing as arguments the ID of the desired element and the CSS property name whose value you're looking for.

Building an API along these lines lets you raise the common denominator of DHTML functionality for your applications. You free yourself from limits that would be imposed by adhering to 100% syntactical compatibility.

Using Third-Party APIs and Frameworks

For nearly every cross-browser need, you can probably unearth dozens of JavaScript libraries, APIs, and frameworks available on the Internet. Many of them are free, while some have licensing fees. These libraries generally come into being because their authors either didn't like the way existing libraries worked, or they developed them in response to specific projects they were working on.

Third-party libraries can offer some genuine benefits. Most notably, the popular ones have likely been tested extensively against a wider range of browsers than you normally use. If you don't have access to a Macintosh, for example, it's comforting to use code that has supposedly been vetted against Safari. The same goes for earlier versions of Internet Explorer for Windows if you don't have extra PCs lying around that have a few previous versions installed for testing purposes.

On the downside, however, it's not uncommon for such libraries to be overweight in the code department. A library that promises to deliver the world will likely have a ton of routines that you don't need for a particular job. Additionally, some libraries are built using advanced scripting techniques that may make it difficult for you to

dissect the code, in case you wish to trim its size or extract only a handful of routines. Some of the libraries designed with heavy object orientation rely on complex interactions of abstract objects that may not be easy to grasp at the code level if you're not yet a scripting wizard. This could spell trouble in the future if the library is no longer supported (its author has moved on to other things) and you can't figure out how to fix it against changes in future browser versions.

If you choose to go with third-party libraries (including the handful of API routines in this book), make sure you understand how they work from the inside out. That way you can trim or extend them to fit the specific needs of your projects.

ONLINE SECTION III

Adding Cascading Style Sheets to Documents

Like their counterparts in word processing and desktop publishing programs, style sheets are supposed to simplify the deployment of fine-tuned formatting associated with HTML content. Instead of surrounding every `h1` element in a document with `<font>` tags to make all of those headings the same color, you can use a one-line style definition in a style sheet to assign a color to every instance of the `h1` element on the page. This puts the purpose of tagging in its proper place: assigning context within a document via HTML markup, while rules governing the appearance of data within that context belong to the style sheet.

Observing HTML Structures

In order to incorporate style sheets successfully into HTML documents, you may have to reexamine your current tagging practices. How much you'll have to change your ways depends on how and when you learned HTML in the first place. Over the years, popular browsers have generally accommodated—how shall I say it—less-than-perfect HTML. Consider the `<p>` tag, which for years in the old days had been treated as a single tag that separates paragraphs with a wider line space than the `<br>` line break tag. HTML standards even encourage this start-tag-only thinking by making some end tags optional. For example, you can define an entire row of table cells without once specifying a `</td>` or `</tr>` tag: the browser automatically closes a tag pair when it encounters a logical start tag for, say, the next table cell or row. This kind of markup is often unflatteringly referred to as "tag soup."

The "new thinking" you should adopt is triggered by an important fact: style sheets, and the browser object models that work with them, are largely container-oriented. With rare exception (the `br` element is one), an element in a document should be treated as a container whose territory is bounded by its start and end tags (even if the end tag is optional).* This container territory does not always translate to space on the page, but does have a direct bearing on the structure of the HTML source code. To see how "HTML-think" has changed since the early days, let's look at a

progression of simple HTML pages. Here's a page that might have been excerpted from a tutorial for HTML Version 2:

```
<html>
<head>
<title>Welcome to HypeCo</title>
</head>
<body>
<h1>Welcome to HypeCo's Home Page</h1>
We're glad you're here.
<p>
You can find details of all of HypeCo's latest products and special offers.
Our goal is to provide the highest quality products and the best customer
service in the industry.
<p>
<a href="products.htm">Click here</a> to view our on-line catalog.
</body>
</html>
```

While the preceding HTML produces a perfectly fine, if boring, page, a modern browser does not have enough information from the tags to turn the content below the `h1` element into three genuine paragraph elements. Before applying a document-wide paragraph style to all three paragraphs, you must make each paragraph its own container. For example, you can surround the text of the paragraph with a `<p>/</p>` tag pair:

```
<html>
<head>
<title>Welcome to HypeCo</title>
</head>
<body>
<h1>Welcome to HypeCo's Home Page</h1>
<p>We're glad you're here.</p>
<p>
You can find details of all of HypeCo's latest products and special offers.
Our goal is to provide the highest quality products and the best customer
service in the industry.
</p>
<p>
<a href="products.htm">Click here</a> to view our on-line catalog.
</p>
</body>
</html>
```

When viewed in a modern browser, the pages created by the two preceding examples look identical. But internally, the browser recognizes three paragraph elements

* In XHTML, all elements require end tags, including so-called empty elements. More commonly, empty elements utilize an internal forward slash shortcut, as in `<br />`. The optional extra space helps this form work in pre-XHTML browsers.

in the second example, and, more importantly, the style of these paragraphs can be controlled by style sheets.

The HTML 4 vocabulary for DHTML-capable browsers includes two additional tags you can use to establish containment: `<div>` and `<span>`. A `div` element creates a container shaped like a block that begins at the starting point of one line and ends with a line break. A `span` element is an inline container, meaning that there are no default built-in line breaks before or after the element. For example, if you want to assign a special style to the first two paragraphs in our example's body, one approach is to group those two elements inside a surrounding `div` container:

```
<body>
<h1>Welcome to HypeCo's Home Page</h1>
<div>
<p>We're glad you're here.</p>
<p>
You can find details of all of HypeCo's latest products and special offers.
Our goal is to provide the highest quality products and the best customer
service in the industry.
</p>
</div>
<p>
<a href="products.htm">Click here</a> to view our on-line catalog.
</p>
</body>
```

Surrounding the two paragraph elements by the `<div>` tag pair does not affect how the content is rendered in the browser, but as shown in Figure III-1, it does alter the containment structure of the elements in the document.

As you can see from Figure III-1, even a simple document has a number of containment relationships. The link in the last paragraph is contained by the third paragraph element; the paragraph element is contained by the `body` element; the `body` element is contained by the `html` element; and the `html` element is contained by the holder of the whole page: the `document`.

Understanding the Box Model

CSS treats each element in a document as a box within the rendered page (speaking here of documents to be displayed visually in a browser). A box can be presented in one of two basic ways: *block* or *inline*.

The default behavior of a typical block-level element is that it begins at the starting margin of one line and ends in a way that forces the next bit of content to appear on a new line following the block. For example, the standard block-level behavior of a heading element (`h1`, `h2`, etc.) causes the element to start on its own line and forces a subsequent element to start on the line below the heading.

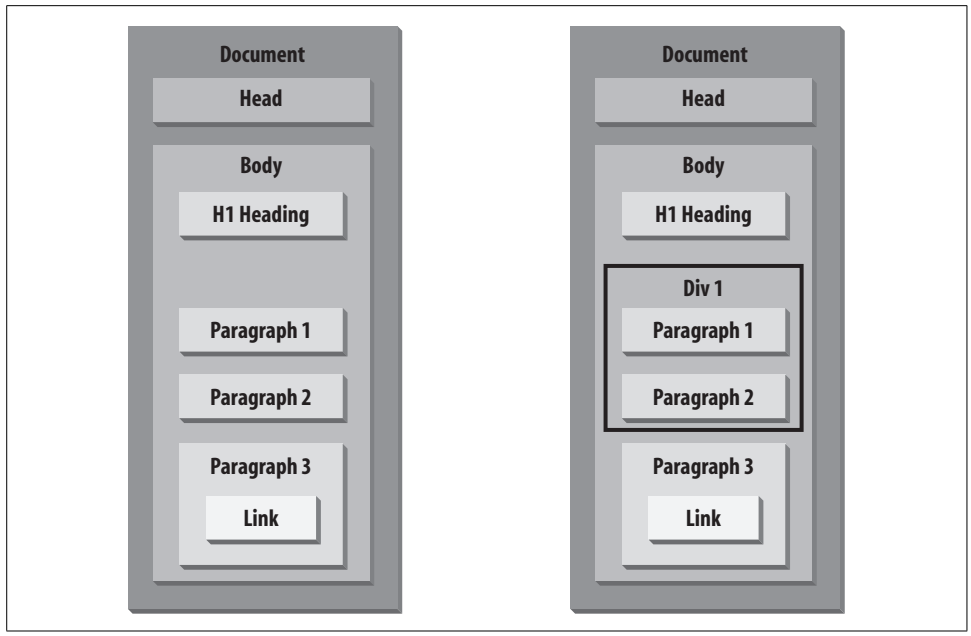


Figure III-1. Element containment before and after the addition of the tag

In contrast, an inline element does not form its own block (although it still has a box), and therefore begins its rendering within the current line of its containing element. For example, inserting an `em` element into the middle of a `p` element (to indicate emphasis of some words) keeps the emphasized words within the natural flow of surrounding words in the paragraph.

Every box has provisions for the drawing of a border, a margin, and padding around the element's content. Those features are controlled by CSS properties. In fact, those three terms—*border*, *margin*, and *padding*—account for about one-fourth of all CSS2 style sheet property names.

Box Pieces

To help you visualize the relationships among the components of an element's box, Figure III-2 shows a schematic diagram of an element box (imagine it's a paragraph, if that helps), where the margin, border, and padding are indicated in relation to the content. The width and height of the content do not change, even when extra stuff is tacked on outside of the content.* Each of the surrounding features—padding, borders, and margins—can occupy space based on its corresponding dimensions. The

* This is the way IE 6 and later for Windows work when in standards-compatible mode (see the <!DOCTYPE> element discussion in Chapter 1 of *Dynamic HTML: The Definitive Reference,* Third Edition). In backwards-compatible (quirks) mode and earlier versions, the browser includes the border and padding in element height and width calculations.

width and height of the entire box is the sum of the element content, plus padding, borders, and margins. If you don't assign any values to those features, their dimensions are usually zero and, therefore, they contribute nothing to the dimensions of the box. In other words, without any padding, borders, or margins, the content and box dimensions are generally identical (although some browsers build in margins for a few elements, such as `body`). With style sheets, you can assign values to your choice of edges (top, right, bottom, or left) for any feature.

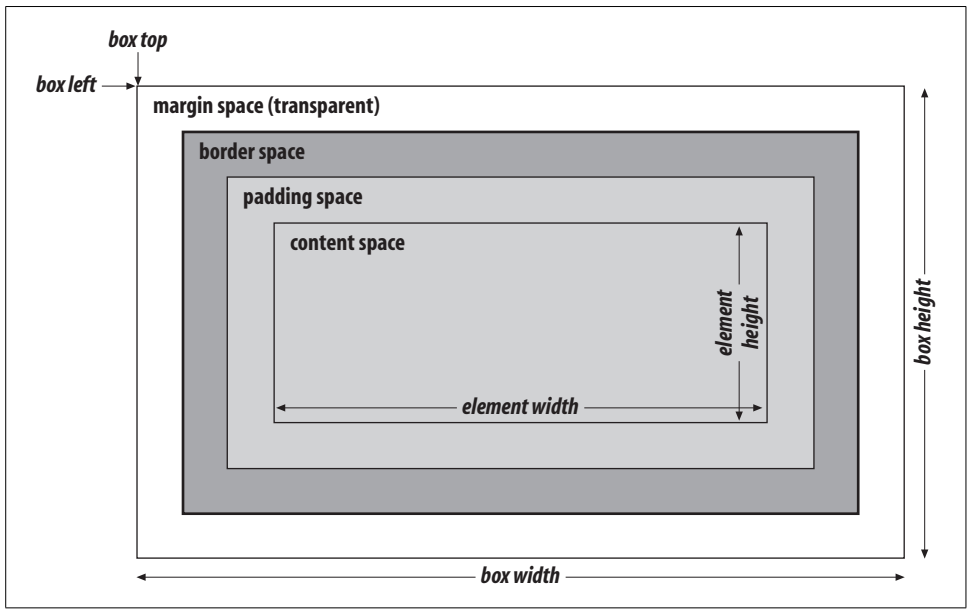


Figure III-2. Schematic diagram of a CSS block-level element

All margin space is transparent. Thus, any colors or images that exist in the next outer containing box show through the margin space. Borders are opaque and always have a color associated with them. Padding space is also transparent, so you cannot set the padding to any color; the background color or image of the content, however, bleeds into the padding space. Thus, this space "pads" the content to give some extra breathing room between the content and any border and/or margin defined for the element.

Some style sheet properties provide a one-statement shortcut for applying independent values to each of the four edges of the margin, border, or padding. For example, you can set the top and bottom border widths to one size and apply a different size to the left and right sides of the same border. When such shortcuts are available (see the `border`, `margin`, and `padding` style properties in Chapter 4 of *Dynamic HTML,* Third Edition), the values are applied in the same order: clockwise from the top—top, right, bottom, left.

Box Positioning

While the content dimensions remain the same regardless of the dimensions assigned to various box features, the size of the box expands when you assign padding, borders, and margins to the element. As you will see in Online Section V, the "thing" that gets positioned within the various coordinate planes is the box.

It is important to understand the difference between a piece of content and its containing box, especially if you start nesting positioned elements or need to rely on extremely accurate locations of elements on the page. To counteract small margins or padding assigned to elements by default in some browsers, many designers as a matter of routine set these properties initially to zero, and then adjust them as needed for their design. Nesting multiple block-level elements inside each other offers a whole range of possible visual effects, so page designers have much to experiment with while developing unique looks.

Two Types of Containment

If you have worked with JavaScript and the scriptable DOM Level 0 in early browsers, you are aware that objects in this model have a containment hierarchy of their own—an *object containment* hierarchy. The `window` object, which represents the content area of a browser window or frame, is at the top of the hierarchy. The `window` object contains objects such as the `history`, `location`, and `document` objects. The `document` object contains objects such as images and forms, and, among the most deeply nested objects, the `form` object contains form control elements, such as text fields and radio buttons.

Document object containment is vitally important in the comparatively limited DOM Level 0 because the hierarchy defines how you refer to objects and their methods and properties in your scripts in that model. References usually start with the outermost element and work their way inward, using the JavaScript dot syntax to delimit each object. For example, here's how to reference the content of a text field (the `value` property) named `zipCode` inside a form named `userInfo`:

```
window.document.userInfo.zipCode.value
```

More modern DOMs, especially the W3C DOM Level 1 and later, let the structure of the document dictate *element containment* as defined by the tag geography of a document. In this context, you see frequent references to the notion of parents and children, where a nested element is a child of its parent container. CSS relies very heavily on this notion of element containment.

While the terms "parent" and "child" imply an object orientation, this is not the case in the DOM. An `img` element nested in a `td` element, for example, does not inherit the parent `td` element's `id` property. But when applying style sheets to an element containment structure, the concept of inheritance is alive and well: an element can inherit a style assigned to another element higher in the element containment hierarchy.

Inheritance

All HTML document elements belong to the document's style inheritance chain. The root of the style chain is the `html` element (which differs from the DOM root: the even more global `document` node object). Its immediate children (also called descendants) are the next elements in the containment hierarchy. The inheritance chain depends entirely on the structure of HTML elements in the document. Figure III-3 shows the CSS inheritance chains of the documents whose containment structures were depicted in Figure III-1.

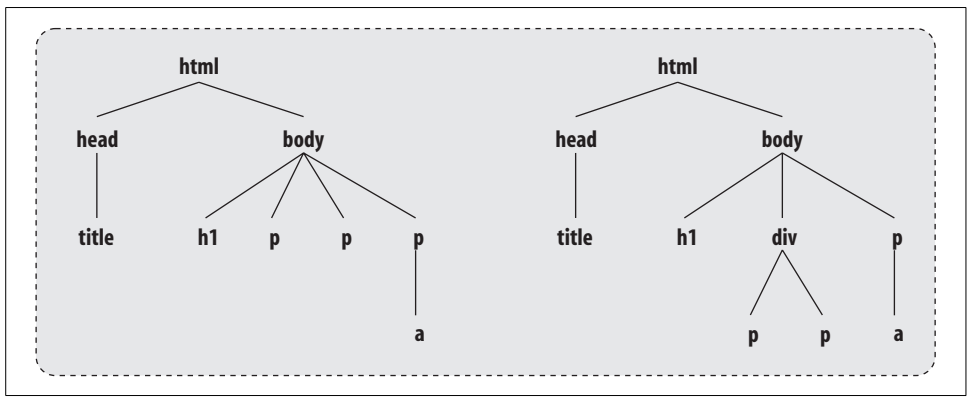


Figure III-3. CSS inheritance chains of two simple documents

The importance of inheritance chains becomes clear when you begin assigning style properties to elements that have descendants. In many cases, you want a descendant to inherit a style assigned to a parent or grandparent. For example, if you assign the Arial font family to all paragraphs (`p` elements), you more than likely want all descendant elements, such as portions designated as `em` elements inside a paragraph, to render their content in the same font family, in which case the default italic effect of the `em` element is applied to the inherited Arial font family.

Note that not all style properties are inherited. Therefore, the style sheet property reference in Chapter 4 of *Dynamic HTML: The Definitive Reference,* Third Edition indicates whether each property is passed from parent to child.

The Cascade

Element containment also plays a role in helping the browser determine which overlapping style sheet rule, of potentially several, should be applied to an element. As you will see later in this chapter, it is possible to assign multiple styles to the same element, by importing multiple style sheet definition files and by defining multiple styles for the same element, or its parent, directly in the document. Cascading style

sheets get their name because styles can flow from a number of sources; the outcome of this cascade is what is displayed by the browser.

I'll come back to cascading later in this chapter, but for now you should be aware that the first step in predicting the outcome of overlapping style sheets is determining the element containment structure of the document. Once you know where an element stands within the document's inheritance chain, you can apply strict CSS principles that assign varying weights to the way a style is defined for a particular element.

Of Style Sheets, Elements, Properties, and Values

Simply stated, a style sheet is a collection of one or more *rules*. Each rule has two parts to it:

- References to one or more elements (or groups of elements) that are having style sheets defined for them
- One or more style sheet properties that apply to the element(s)

In other words, each rule defines a particular look and feel as well as the item(s) in the document that are to be governed by that look and feel.

Style Properties—An Overview

A style property is the name of a (usually) visible attribute of a piece of content on the page. CSS 2.1 defines 115 separate style properties. To give you a quick overview of the range of properties for visual presentation, Table III-1 offers of summary of property names grouped by category. You can find details of all CSS style sheet properties in Chapter 4 of *Dynamic.HTML*, Third Edition.

CSS 2.1 accommodates the idea that browser makers might implement new properties outside the scope of the standard. It was important to define a syntax that would not conflict with possible new W3C properties in the future. Recommended formats are as follows:

```
-vendorID-propertyName
_vendorID_propertyName
```

Mozilla browsers use this format to implement both new properties and properties that have not yet been finalized in the standards track, such as `-moz-border-radius-topleft`. Safari's vendor-specific styles begin with `-khtml-`.

Table III-1. Summary of CSS2.1 style sheet properties

Box properties		
border	border-top-style	margin-top
border-top	border-right-style	margin-right
border-right	border-bottom-style	margin-bottom
border-bottom	border-left-style	margin-left
border-left	border-width	outlines
border-color	border-top-width	padding
border-top-color	border-right-width	padding-top
border-right-color	border-bottom-width	padding-right
border-bottom-color	border-left-width	padding-bottom
border-left-color	margin	padding-left
border-style		
Color and background properties		
background	background-image	background-repeat
background-attachment	background-position	color
background-color		
Font properties		
font	font-size	font-variant
font-family	font-style	font-weight
Text properties		
letter-spacing	text-decoration	vertical-align
line-height	text-indent	white-space
text-align	text-transform	word-spacing
Visual formatting properties		
bottom	left	table-layout
clear	max-width	top
clip	min-width	unicode-bidi
direction	overflow	visibility
display	position	width
float	right	z-index
height		
Generated content and list properties		
content	list-style-position	list-style
list-style-image	list-style-type	quotes
Paged media properties		
orphans	page-break-before	widows
page-break-after	page-break-inside	
User Interface properties		
cursor		

CSS Property Assignment Syntax

The syntax for assigning a value to a property is different from what you know about HTML attributes and their values. Assign a value via the colon operator (in contrast to the equal sign operator in HTML). A space after the colon is optional. The combination of a property name, colon operator, and value to be assigned to the property is called a *declaration*. To assign the color red to the foreground of an element, you could use either of the following simple declaration forms (colors may be specified many ways, as described in Chapter 4 of *Dynamic HTML: The Definitive Reference,* Third Edition):

```
color: #ff0000
color: red
```

If a style sheet rule includes more than one declaration, separate declarations with semicolons:

```
color: #ff0000; font-size: 12pt
```

A trailing semicolon after the last declaration is optional, as is a space after the internal semicolon.

Notice, however, that unlike HTML attribute values, CSS syntax property values do not—and cannot—have double quotes around the values, except in rare circumstances (e.g., multiword font family names).

Binding CSS Declarations to Elements

Defining style declarations is only half the job. The other half involves instructing the browser to apply the declaration(s) to the desired element(s). This is the job of the *selector*, which acts as a kind of label signifying the element or type of element to which a declaration applies. In a simple case, you bind a declaration to a single element or a single type of element (e.g., all `p` elements, where the label is, literally, just `p`). The CSS syntax for a statement combining a selector and one or more declarations is as follows:

```
selector {property: value[; property: value[; ...]]}
```

For example, to assign a red color and specific font size to all `h1` elements, the CSS statement is:

```
h1 {color: red; font-size: 24px}
```

Embedding Style Sheets

You add style sheets to a document by defining them explicitly in the document's source code, importing definitions from one or more external files, or a combination of the two. In-document and external style sheets coexist well in the same document; you can have as many of each type as your page design requires.

In-Document Styles

There are two ways to embed the source code for CSS rules directly in an HTML document: using the `<style>` tag pair or using the `style` attribute of HTML tags. While placing a style definition in an element's tag flouts the trend toward separating context from rendering, you may encounter valid reasons (explained later) to use this approach from time to time.

The <style> tag

The `style` element is part of the HTML 4 specification (and, by extension, XHTML 1.x specs). The element must be located within the `head` element and must also include the `type` attribute. This attribute specifies the content type of the element—`text/css` when using the CSS language, as in:

```
<style type="text/css">
    style sheet rule(s) here
</style>
```

Some non-CSS-aware browsers (an extreme rarity, these days) ignore the start and end tags and attempt to render the rules as if they were part of the document body. If you fear that this will affect users of your pages, you can surround the statements inside the `style` element with HTML comment symbols, as follows:

```
<style type="text/css">
<!--
    style sheet rule(s) here
-->
</style>
```

This technique is similar to the one used to hide the contents of `<script>` tag pairs from older browsers, except that the end-comment statement in a script must start with a JavaScript comment symbol (`//-->`). The comment-bracketed content is still downloaded to the client and is visible in the source code, but for all but the most brain-dead browsers, the style sheet rules are hidden from plain view in the browser window. In the examples in this book, I have omitted these comment symbols.

As I mentioned earlier, the link between a style declaration and the element(s) it governs is called a selector. In practice, "selector" has a wide range of meanings. In its simplest form, a selector is the name of one type of HTML element—the HTML tag stripped of its enclosing angle brackets (e.g., the `p` selector, which represents all paragraph elements in a document). As you will see as this chapter progresses, a selector can take on additional forms, including some that have no resemblance at all to HTML elements. Just remember that a selector defines the part (or parts) of an HTML document governed by a style declaration.

In the most common application, each style rule binds a declaration to a particular type of HTML element based on the element's tag name. Get in the habit of using

lowercase tag selectors to match the XHTML-inspired lowercase tag names. This simplest selector form is called a *type selector* (as in the type of element).

When a rule is specified in a `<style>` tag, the declaration portion of the rule must appear inside curly braces, even if there is just one style property in the declaration. Each curly brace pair and its content is known as a *declaration block*. The combination of a selector and a declaration block comprises a *rule set* (or *rule*) The style sheet in the following example includes two rule sets. The first assigns the red foreground (text) color and initial capital text transform to all `h1` elements in the document; the second assigns the blue text color to all `p` elements:

```
<html>
<head>
<style type="text/css">
    h1 {color: red; text-transform: capitalize}
    p {color: blue}
</style>
</head>
<body>
<h1>Some heading</h1>
<p>Some paragraph text.</p>
</body>
</html>
```

There is no practical limit to the number of rule sets that can be listed inside the `style` element, nor is there a limit to the number of style properties that can be defined in a style rule. Also, rule sets can appear in any order within a style sheet (although order may play a significant role, as described later), and the indenting shown in the preceding example is purely optional. White space (spaces or line breaks) between property/value pairs and before or after curly braces is not critical. As a result you can also break up a series of declarations (inside the curly braces) so that each property/value pair appears on its own line, as follows:

```
h1 {
     color: red;
     text-transform: capitalize;
}
```

This format, popular with CSS designers, makes it easier to locate a rule for a particular selector and also spot which rules in a complex style sheet contain a particular CSS property name.

CSS syntax provides a shortcut for assigning the same style declaration to more than one selector. By preceding the curly-braced style declaration block with a comma-delimited list of selectors, you can have one statement do the work of two or more statements. For example, if you want to assign the same color to `h1`, `h2`, and `h3` elements in the document, you can do so with one statement:

```
<style type="text/css">
    h1, h2, h3 {color: blue}
</style>
```

This selector grouping technique is applicable to all CSS selector types described in this Online Section.

The style attribute in element tags

Another way to bind a style declaration to an HTML element is to include the declaration as an attribute of the actual HTML element tag. The declaration is assigned to the `style` attribute; almost every HTML element recognizes the `style` attribute.

Because the `style` attribute is a regular HTML attribute, you assign a value to it via the equal sign operator. The value is a double-quoted string that consists of one or more style property/value pairs in the default CSS style sheet syntax. These style property/value pairs use the colon assignment operator. Use a semicolon to separate multiple style property settings within the same `style` attribute. The following code uses a `style` attribute version of the example shown in the preceding section. Because the style sheet rules are attached to the actual HTML element tags, all this takes place in the `body` section of the document:

```
<body>
<h1 style="color: red; text-transform: capitalize">Some heading</h1>
<p style="color: blue">Some paragraph text.</p>
</body>
```

Notice, too, that when a style sheet definition is specified as a `style` attribute, there are no curly braces involved. The double quotes surrounding the entire style sheet definition function as the curly brace grouping characters.

Importing External Style Sheets

Perhaps the most common use of style sheets in the publishing world is to establish a "look" designed to pervade across all documents, or at least across all sections of a large document. To facilitate applying a style sheet across multiple HTML pages, the CSS specification provides two ways to include external style sheet files: an implementation of the `<link>` tag and a special type of style sheet rule selector called the `@import` rule.

External style sheet files

No matter how you import an external style sheet, the external file must be written in such a way that the browser can use it to build the library of style sheets that controls the currently loaded document. In other words, the browser must take into account not only external styles, but any other styles that might also be defined inside the document. Because there is an opportunity for the overlap of multiple style sheets in a document, the browser must see how all the styles are bound to elements, so it can apply cascading rules (described later in this chapter) to render the content.

An external style sheet file consists exclusively of style sheet rule sets without any HTML tags. The file should be in plain text format and saved with the *.css* filename extension. For example, to convert the style sheet used in the previous sections to an external style sheet file, create a text file that contains the following and save the file as *basestyle.css*:

```
h1 {color: red; text-transform: capitalize}
p {color: blue}
```

When a browser encounters either importing technique, the content of the file is loaded into the browser as if it were typed into the main HTML document at that source code location (although it doesn't become part of the source code if you use the browser to view the source). The web server must also be configured to associate the *.css* filename extension with a content-type of `text/css` (most modern servers are already set up this way, but check with the server administrator if you're not sure).

The link element

HTML recognizes `<link>` as a general-purpose tag for linking media-independent content into a document (not to be confused with hypertext links created by the `<a>` tag). It is up to the browser to know how to work with the various attributes of this tag (see Chapter 1 of *Dynamic HTML,* Third Edition).

The CSS2 specification stakes its claim to the `link` element as a way to load an external style sheet file into a document. The attributes and format for the tag are rather simple:

```
<link rel="stylesheet"  type="contentType" href="filename.css">
```

The *contentType* value for CSS style sheets is `text/css`. If the style sheet in the previous section is saved as *basestyle.css*, you can import that style sheet as follows:

```
<html>
<head>
<link rel="stylesheet" type="text/css" href="basestyle.css">
</head>
<body>
<h1>Some heading</h1>
<p>Some paragraph text.</p>
</body>
</html>
```

An HTML document can have multiple `link` elements for importing multiple external style sheet files (only one file per `link` element). The document can also contain `style` elements as well as `style` attributes embedded within element tags. But if there is any overlap of more than one style applying to the same element, the cascade rules (described later in this chapter) determine the specific style sheet rule that governs the element's display.

The @import rule

CSS2 describes an extensible system for declarations or directives (commands, if you will) that become a part of a style sheet definition. They are called *at-rules* because a rule starts with the "at" symbol (`@`), followed by an identifier for the declaration. Each at-rule includes one or more descriptors that define the characteristics of the rule and end with a semicolon. (For more about at-rules, see Chapter 4 of *Dynamic HTML*, Third Edition.)

One such at-rule that is implemented starting in IE 4 and is now supported by all mainstream browsers imports an external style sheet file from inside a `style` element. It performs the same function as the `link` import technique described in the previous section. In the following example, a file containing style sheet rules is imported into the current document:

```
<style type="text/css">
    @import url("styles/corporate.css");
</style>
```

You may include multiple `@import` rules in a `style` element, but they must come before rules with any other type of selector. An `@import` rule must also stand alone, without being nested inside curly brace blocks.

Loading Browser-Specific Style Sheets

Due to fluctuations in interpretations of CSS from one browser to another, you may be faced with an occasional need to import different style sheets for different browsers or browser generations. Among the biggest problems you'll encounter are radical departures from the CSS box model in Internet Explorer for Windows prior to IE 6 (and IE 6 or later running in "quirks mode").

Scripters tend to look first at a scripted solution, incorporating "browser sniffing" to determine the current browser through the `navigator.userAgent` property value. A branching script can then insert a link element that loads the desired *.css* file. Unfortunately, the proliferation of values for the `userAgent` property make such scripts hazardous for future compatibility—unless you are trying to isolate a very specific, older browser version that requires special consideration.

Starting with IE 5, Microsoft built in a separate branching mechanism called conditional comments. Coding consists of short conditional expressions inside HTML comments. If the current browser meets the condition, then HTML between the comment tags is rendered; otherwise (and this goes for non-IE browsers), the HTML code inside the comments is ignored. For example, if you wish to load a special style sheet for IE 5 and 5.5, you can do so and have it override the one that all other browsers use, as follows:

```
<link rel="stylesheet" type="text/css" href="basestyle.css">
<!--[if lt IE6]>
```

```
    <link rel="stylesheet" type="text/css" href="badboxstyle.css">
<![end if]-->
```

For the full syntax of conditional comments, see the `<!--comment-->` element at the end of Chapter 1 of *Dynamic HTML,* Third Edition.

Selecting a Style Sheet Style

In deciding among the many ways to introduce style sheets into your pages—the `<style>` tag, the `style` attribute, or imported from outside the document—you need to consider how important it is for you to separate design from content. Within a single document file, the `<style>` tag technique distances HTML content from the styles associated with elements throughout the document. If you need to change a font family or size for a particular kind of element, you can do so quickly and reliably by making the change to one location in the document. If, on the other hand, your style definitions are scattered among dozens or hundreds of tags throughout the document or web site, such a change requires much more effort and the possibility for mistakes increases.

As discussed in Online Section V, where you declare an element's style impacts how DHTML scripts read the element's initial style properties. An element object's `style` property reflects only values assigned via the element tag's `style` attribute. In contrast, reading the initial value of styles applied through `<style>` or imported style sheet rules requires special syntax that is incompatible between IE and W3C DOM implementations, as discussed in Online Section V.

Current web development trends lean toward the separation of design from content. In large projects involving writers, designers, and programmers, it is usually easier to manage the entire project if different contributors to the application can work toward the same goal without stepping on each other's code along the way. It is no accident that both style sheets and scripts have acquired mechanisms for importing external files into an HTML document. This allows designers to work on their *.css* files and programmers to work on their *.js* files, all of which blend into the writer's *.html* file (or equivalent server output) that arrives at the client.

Common Subgroup Selectors

While a selector for a style sheet rule is often an HTML element name, that scenario isn't flexible enough for more complex documents. Consider the following possibilities:

- You want certain paragraphs scattered throughout the document to be set apart from running text with wider left and right margins.
- You want all but one of the `h2` elements in the document to be set to the color red; the one exception must be blue.

- In a three-level ordered list (`ol`) group, you want to assign different font sizes to each level.

Each of these possibilities calls for a different way of creating a new selector group or specifying an exception to the regular selectors. In an effort to distance design from content, CSS style sheets provide three simple ways of creating subgroups that can handle a large variety of design possibilities:

- Class selectors
- ID selectors
- Descendant selectors

Using these subgroup selectors requires special ways of defining selectors in style sheet rules. Two of these selectors also require the addition of attributes to the HTML tags they apply to in the document. Because all CSS-enabled browsers support these CSS1 subgroup selectors, you should have these deeply ingrained in your authoring repertoire.

CSS Level 3 describes an entirely new set of categories for selectors (the third in as many levels). Because so much of CSS Level 3 is still in working draft stage, the categories may change once more. Therefore, the category names in this chapter are consistent with those described in CSS Levels 2 and 2.1, even though some of the selectors are found only in CSS Level 3.

Class Selectors

A class selector is an identifier you can use to assign a style to a subset of elements in a document. To apply a class selector, you first invent an identifier for the class name. CSS2 guidelines for selector identifiers allow all Latin letters (a through z and A through Z), numerals (0 through 9), the hyphen, Unicode characters above 160, and escaped characters (characters that begin with a backslash character)—provided the name does not begin with a numeral or hyphen. Spaces are not permitted. For the sake of compatibility and the occasional browser bug, it is good practice to stick with Latin letters only. Browsers operating in standards-compatible modes treat selectors in a case-sensitive manner. Therefore, the best forward- and backward-compatible practice is to observe case throughout, but not to reuse names with different combinations of upper- and lowercase letters.

The class identifier goes in both the style sheet rule and the HTML tag (assigned to the `class` attribute) for all elements that are to obey the rule. While the identifier name is the same in both cases, the syntax for specifying it is quite different in each place.

Binding a class identifier to an element type

In the style sheet rule, the class identifier is part of the rule's selector. When a class selector is intended to apply to only one kind of HTML element, the selector consists of the element tag name, a period, and the identifier. The following rule assigns a specific margin setting for all p elements flagged as belonging to the narrow class:

```
p.narrow {margin-left: 5em; margin-right: 5em}
```

To force a p element to obey the p.narrow rule, you must include a class attribute in the <p> tag and set the value to the class identifier:

```
<p class="narrow">Content for the narrow paragraph</p>
```

All p elements that don't have the class attribute set to narrow follow the style applied to the generic p element. Example III-1 shows a complete document that includes style sheet rules for all p elements and a subclass of p.narrow elements. The rule for all p elements specifies a 2-em margin on the left and right as well as a 14-pixel font size. For all p elements tagged with the class="narrow" attribute, the margins are set to 5 ems and the text color is set to red. Note the p.narrow rule inherits (or is affected by) style settings from the p rule. Therefore, all text in the p.narrow elements is displayed at a font size of 14 pixels. But when the margin properties are set in both rules, the settings for the named class override the settings of the broader p element rule (the language of CSS doesn't include the object-oriented concepts of subclass or superclass). Following the inheritance trail one level higher in the containment hierarchy, all p elements (and all other elements in the document if there were any) obey the style sheet rule for the body element, which is where the font face is specified.

Example III-1. Applying the p.narrow class rule

```
<html>
<head>
<title>Class Society</title>
<style type="text/css">
    p {font-size: 14px; margin-left: 2em; margin-right: 2em}
    p.narrow {color: red; margin-left: 5em; margin-right: 5em}
    body {font-family: Arial, sans-serif}
</style>
</head>

<body>
<p>
This is a normal paragraph. This is a normal paragraph. This is a normal
paragraph. This is a normal paragraph. This is a normal paragraph.
</p>
<p class="narrow">
This is a paragraph to be set apart with wider margins and red color. This is a
paragraph to be set apart with wider margins and red color. This is a paragraph
to be set apart with wider margins and red color.
</p>
```

Example III-1. Applying the p.narrow class rule (continued)

```
<p>
This is a normal paragraph. This is a normal paragraph. This is a normal
paragraph. This is a normal paragraph. This is a normal paragraph.
</p>
<p class="narrow">
This is a paragraph to be set apart with wider margins and red color. This is a
paragraph to be set apart with wider margins and red color. This is a paragraph
to be set apart with wider margins and red color.
</p>
</body>
</html>
```

Defining a free-range class rule

You don't have to limit a class selector to a single element type in a document. Instead, you can define a rule with a class selector that can be applied to any element in the document. The selector of such a rule is nothing more than the identifier preceded by a period. Example III-2 contains a rule that assigns a red underline style to a class named `hot`. The `hot` class is then assigned to different elements scattered throughout the document. Notice inheritance at work in this example. When the `hot` class is assigned to a `div` element, it applies to the `p` element nested inside the `div` element: the entire paragraph is rendered in the `hot` style and follows the `p.narrow` rule as well, since the rules do not have any overlapping style properties.

Example III-2. Adding a free-range class selector

```
<html>
<head>
<title>Free Range Class</title>
<style type="text/css">
    p {font-size: 14px; margin-left: 2em; margin-right: 2em}
    p.narrow {color: red; margin-left: 5em; margin-right: 5em}
    .hot {color: red; text-decoration: underline}
    body {font-family: Arial, sans-serif}
</style>
</head>

<body>
<h1 class="hot">Get a Load of This!</h1>
<p>
This is a normal paragraph. This is a normal paragraph. This is a normal
paragraph. This is a normal paragraph. This is a normal paragraph.
</p>
<div class="hot">
<p class="narrow">
This is a paragraph to be set apart with wider margins and red color. This is a
paragraph to be set apart with wider margins and red color. This is a paragraph
to be set apart with wider margins and red color.
</p>
</div>
```

Example III-2. Adding a free-range class selector (continued)

```
<p>
This is a normal paragraph. This is a normal paragraph <span class="hot">but with a
red-hot spot</span>. This is a normal paragraph. This is a normal paragraph. This
is a normal paragraph.
</p>
<p class="narrow">
This is a paragraph to be set apart with wider margins and red color. This is a
paragraph to be set apart with wider margins and red color. This is a paragraph
to be set apart with wider margins and red color.
</p>
</body>
</html>
```

ID Selectors

In contrast to the class selector, the ID selector lets you define a rule that applies to only one element in the entire document. Like the class selector, the ID selector requires a special way of defining the selector in the style sheet rule and a special attribute (`id`) in the tag that is the recipient of that rule. The same rules and warnings about defining class names also apply to ID names. Uniqueness within a document is critical not only for CSS ID selectors to work correctly, but especially for DHTML scripting. An element's `id` attribute value acts as a kind of address that scripts use to reference the element, regardless of document structure. This means that to maintain integrity of the object model for the current document, an `id` identifier must apply to only one element.

The style rule syntax for defining an ID selector calls for the identifier to be preceded with the # symbol. This can be in conjunction with an element selector or by itself. Therefore, both of the following rules are valid:

```
p#special4 {border: 5px ridge red}
#special4 {border: 5px ridge red}
```

To apply this rule for this ID to a `p` element, you have to add the `id` attribute to that element's tag:

```
<p id="special4">Content for a special paragraph.</p>
```

There is an important difference between the two style rule examples just shown. By specifying the ID selector in concert with the `p` element selector in the first example, we've told the browser to obey the `id="special4"` attribute only if it appears in a `p` element. The second rule, however, is a generic rule. This means that the `id="special4"` attribute can appear in any kind of element. Since an `id` attribute value should be used in only one element throughout the entire document, the combined selector is redundant.

Example III-3 shows the ID selector at work, where it is used to assign a rule (defining a red, ridge-style border for a block) to only one of several `p` elements in the doc-

ument. Notice that it is assigned to a p element that also has a class selector assigned to it: two rules are applied to the same element. In this example, the style rules do not conflict with each other, but if they did, the cascade precedence rules (described later in this chapter) would automatically determine precisely which rule wins the battle of the dueling style properties.

Example III-3. Applying an ID selector to a document

```
<html>
<head>
<title>ID Selector</title>
<style type="text/css">
    p {font-size: 14px; margin-left: 2em; margin-right: 2em}
    p.narrow {color: red; margin-left: 5em; margin-right: 5em}
    #special4 {border: 5px ridge red}
    body {font-family: Arial, sans-serif}
</style>
</head>

<body>
<h1>Get a Load of This!</h1>
<p>
This is a normal paragraph. This is a normal paragraph. This is a normal
paragraph. This is a normal paragraph. This is a normal paragraph.
</p>
<p class="narrow" id="special4">This is a paragraph to be set apart with wider
margins, red color AND a red border. This is a paragraph to be set apart with
wider margins, red color AND a red border.
</p>
<p>
This is a normal paragraph. This is a normal paragraph. This is a normal
paragraph. This is a normal paragraph. This is a normal paragraph.
</p>
<p class="narrow">This is a paragraph to be set apart with wider margins and red
color. This is a paragraph to be set apart with wider margins and red color. This
is a paragraph to be set apart with wider margins and red color.
</p>
</body>
</html>
```

Descendant Selectors

One more way to assign styles to specific categories of elements is the descendant selector (once known as a contextual selector). To use a descendant selector, you should be comfortable with the containment hierarchy of elements in a document and how inheritance affects the application of styles to a chunk of content. Consider the two type selector rules in the following style sheet:

```
<style type="text/css">
    p {font-size: 14px; color: black}
    em {font-size: 16px; color: red}
</style>
```

This style sheet dictates that all em elements throughout the document be displayed in red with a 16-pixel font. If you were to add an em element as part of an h1 element, the effect might be less than desirable. What you really want from the style sheet is to apply the em style declaration to em elements only when they are contained by—are descended from—p elements. A descendant selector lets you do just that. In a descendant selector, you list the elements of the containment hierarchy that are to be affected by the style, with the elements separated by spaces.

To turn the second rule of the previous style sheet into a descendant selector, modify it as follows:

```
<style type="text/css">
    p {font-size: 14px; color: black}
    p em {font-size: 16px; color: red}
</style>
```

You still need the rule for the base p element in this case because the style is something other than the browser default. There is no practical limit to the number of containment levels you can use in a descendant selector. For example, if the design calls for a section of an em element to have a yellow background color, you can assign that job to a span element and set the descendant selector to affect a span element only when it is nested inside an em element that is nested inside a p element. Example III-4 shows what the source code for such a document looks like. The example goes one step further, in that one element of the descendant selectors is a class selector (p.narrow). Each element selector in a descendant selector can be any valid selector, including a class or ID selector. You can also apply the same style declaration to more than one descendant selector by separating the descendant selector sequences with commas:

```
p em span, h3 em {background-color: yellow}
```

It's an odd-looking construction, but it's perfectly legal (and byte conservative).

Example III-4. Applying a three-level descendant selector

```
<html>
<head>
<title>Descendant Selector</title>
<style type="text/css">
    p {font-size: 14px; margin-left: 2em; margin-right: 2em}
    p.narrow {color: red; margin-left: 5em; margin-right: 5em}
    p.narrow em {font-weight: bold}
    p.narrow em span {background-color: yellow}
    #special4 {border: 5px ridge red}
    body {font-family: Arial, sans-serif}
</style>
</head>

<body>
<h1>Get a Load of This!</h1>
<p>
```

Example III-4. Applying a three-level descendant selector (continued)

```
This is a normal paragraph. This is a normal paragraph. This is a normal
paragraph. This is a normal paragraph. This is a normal paragraph.
</p>
<p class="narrow" id="special4">This is a <em>paragraph to be set apart</em> with
wider margins, red color AND a red border. This is a paragraph to be set apart
with wider margins, red color AND a red border.
</p>
<p>
This is a normal paragraph. This is a normal paragraph. This is a normal
paragraph. This is a normal paragraph. This is a normal paragraph.
</p>
<p class="narrow">This is a <em>paragraph to be <span>set apart</span></em> with
wider margins and red color. This is a paragraph to be set apart with wider
margins and red color. This is a paragraph to be set apart with wider margins and
red color.
</p>
</body>
</html>
```

Note that a descendant selector points to any descendant of a parent element, even if the descendant is nested many levels deep in the containment hierarchy. To restrict a selector to an immediate child of an element, see "Child Selectors" later in this Online Section.

Advanced Subgroup Selectors

The CSS2 and CSS2.1 recommendations make further enhancements to the way selectors can be specified in style sheet rules. Some advanced selectors are supported only in recent browser versions. See Chapter 4 of *Dynamic HTML*, Third Edition for selector compatibility in major browsers. Most of these advanced selector forms extend the concepts in effect for simple selectors. They provide either special case selectors or additional ways to slice and dice element collections for finely-tuned style designs.

Pseudo-Element and Pseudo-Class Selectors

The original idea for pseudo-elements and pseudo-classes was defined as part of the CSS1 recommendation; these selectors have been expanded in CSS2.x. A fine line distinguishes these two concepts, but they do share one important factor: there are no direct HTML tag equivalents for the elements or classes described by these selectors. Therefore, you must imagine how the selectors will affect the real tags in your document.

Using pseudo-elements

A pseudo-element is a well-defined chunk of content in an HTML element. Two pseudo-elements specified in the CSS1 recommendation point to the first letter and the first line of a paragraph. The elements are named `:first-letter` and `:first-line`, respectively. It is up to the browser to figure out where, for example, the first line ends (based on the content and window width) and apply the style only to the content in that line. If the browser is told to format the `:first-letter` pseudo-element with a drop cap, the browser must also take care of rendering the rest of the text in the paragraph so that it wraps around the drop cap.

For example, to apply styles for the first letter and first line of all `p` elements, use the following style rules:

```
<style type="text/css">
    p:first-letter {font-face: Gothic, serif; font-size: 300%; float: left}
    p:first-line {font-variant: small-caps}
</style>
```

Style properties that can be set for `:first-letter` and `:first-line` include a large subset of the full CSS property set. They include all font, color, background, and several more text-related properties (`line-height`, `text-decoration`, `letter-spacing`, and so on). The `:first-letter` element also allows for borders, margins, and padding.

The CSS2 `:before` and `:after` pseudo-elements offer intriguing possibilities for inserting repeated or generated text before or after an element. For example, you could define a `blockquote:after` selector that inserts the phrase "Reprinted by permission." at the end of every `blockquote` element on the page. Another variation maintains counter variables that track and render incremented numbers to be inserted before each element defined by a selector. Pseudo-elements can have a radical impact on the presentation of not only the immediate text, but nearby content as well. Because of spotty support for these pseudo-elements, tread carefully with their deployment, and test on a wide variety of browsers.

Using pseudo-classes

In contrast to a pseudo-element, a pseudo-class applies to an element whose look or content may change as the user interacts with the content. Pseudo-classes defined in the CSS1 recommendation are for three states of the `a` element: a link not yet visited, a link being clicked on by the user, and a link that has been visited. Default behavior in most browsers is to differentiate these states by colors (default colors can usually be set by user preferences as well as by attributes of the `body` element, although the latter is frowned upon in these days of CSS). The syntax for pseudo-class selectors follows the same pattern as for pseudo-elements. The following style sheet defines rules for the three `a` element pseudo-classes:

```
<style type="text/css">
    a:link {color: darkred}
    a:active {color: coral}
```

```
    a:visited {color: lightblue; font-size: -1}
</style>
```

In CSS2, the `:active` pseudo-class is not restricted to links, but can be applied to any element (indicating that the user is interacting with the element, such as clicking on it). Joining the `:active` pseudo-class are `:focus` (when an element has focus, such as being ready to accept keyboard input) and `:hover` (when the pointer is atop the element without acting on the element). If you wish to effect style changes during a "mouse rollover," the `:hover` pseudo-class lets you accomplish this form of dynamism without any scripting. For example, you can specify a `display` CSS property to show an element during a hover, and hide the element (`display: none`) in other states.

As with other selectors, you can combine class or ID selectors with pseudo-elements or pseudo-classes to narrow the application of a special style. For instance, you may want a large drop cap to appear only in the first paragraph of a page.

CSS3 pseudo-classes

A number of new pseudo-classes join the vocabulary from the CSS3 specification as it nears completion. Several of the new collection are already implemented in Opera 9.

It is clear that one group of the additions, called *structural pseudo-classes*, are aimed at reducing the need to load HTML markup with lots of `class` attribute assignments that are typically used to identify more structural details than HTML tags can do by themselves. For example, many designers like to alternate the pastel background colors of table rows to make the table easier to read. The traditional way to do that with CSS is to assign alternating `class` attribute names to rows, and define a rule for each class. With the `:nth-child( )` pseudo-class, however, you can instruct alternating `tr` elements—by virtue of their structure—to use different background colors without any additional HTML markup:

```
tr:nth-child(2n) {background-color: lightgreen}
tr:nth-child(2n+1) {background-color: lightyellow}
```

The notation inside the parentheses adheres to the format *an* + *b*. In the case of the `:nth-child( )` pseudo-class example above, the notation means that the browser is to:

a. Find all sibling elements of a `tr` element (i.e., all those sharing a common parent, such as a `tbody` element).
b. Divide the entire bunch of siblings into groups of 2.
c. For those at the beginning of each group of two, apply the lightgreen background color.
d. For those in the position of one beyond the group's first item, apply the lightyellow background color.

Other structural pseudo-classes assist style sheet rules that need to be applied to the last child of a container (`:last-child`), repeated groups of child nodes in a container (`:nth-child( )`), elements in a container that are of a specific tag type (`:only-of-type`), and many combinations thereof. Despite the initial complexity these selectors appear to bring to the discussion, they promote the ultimate goal of separating presentation and content markup.

Browser support for pseudo-classes is quite variable. For instance, the `:hover` pseudo-class works only with hyperlink (`a`) elements in IE 6 or earlier and IE 7 in quirks mode. See the discussion about selectors in Chapter 4 of *Dynamic HTML*, Third Edition for details, more examples, and browser support.

Attribute Selectors

In CSS1, the links between style rule selector and an element's attributes are limited to the `class` and `id` attributes. CSS2 broadens the possibilities to include any attribute or attribute/value pair as selectors, while CSS3 extends that notion even further to permit matches of parts of attribute values.

It is helpful to think of an attribute selector as an expression that helps the user agent (browser or application) locate a match of HTML elements containing a particular attribute (and value) to determine whether the style should be applied to the element. A match may occur by the presence of an attribute name in the tag, or an attribute and a specific value. For example, a page may contain several `a` elements, some of which open a second window by assigning the attribute `target="_blank"`. An attribute selector allows one style to apply only to those `a` elements with the attribute combination, while another style applies to all other `a` elements that target the current window.

Table III-2 shows the seven attribute selector formats and what they mean. A new syntactical feature for selectors—square brackets—adds another level of complexity to defining style sheet rules, but the added flexibility may be worth the effort.

Table III-2. CSS attribute selectors

Syntax format	Description
[*attributeName*]	Matches an element if the attribute is defined in the HTML tag regardless of value
[*attributeName*=*value*]	Matches an element if the attribute is set to the specified value in the HTML tag
[*attributeName*~=*value*]	Matches an element if the specified value is present among the values assigned to the attribute in the HTML tag
[*attributeName*|=*value*]	Matches an element if the attribute value contains a hyphen, but starts with *value*
[*attributeName*^=*value*]	Matches an element if the attribute value starts with the characters in *value (CSS3)*
[*attributeName*$=*value*]	Matches an element if the attribute value ends with the characters in *value (CSS3)*
[*attributeName**=*value*]	Matches an element if the attribute value contains the characters in *value (CSS3)*

To see how these selector formats work, observe how the sample style sheet rules in Table III-3 apply to an associated HTML tag.

Table III-3. How attribute selectors work

Style sheet selector	Applies to	Does not apply to

`p[align]`	`<p align="left">`
	`<p align="left"`
	`title="Summary">`
		`<p title="Summary">`

`hr[align="left"]`	`<hr align="left">`	`<hr align="middle">`

`img[alt~="Placeholder"]`	`<img alt="Temporary`
	`Placeholder"`
	`src="picture.gif">`
		`<applet alt="Applet`
		`Placeholder" code=...>`

`p[lang|="en"]`	`<p lang="en-CA">`	`<p lang="fr-CA">`

If you wish to apply a style to all elements that have multiple attributes, you can include multiple attribute selectors in the same rule. For example, the following rule applies to all `input` elements of the text type and have `tabindex` attributes:

```
input [type="text"][tabindex] {background-color: lightyellow}
```

Universal Selectors

In practice, the absence of an element selector before an attribute selector implies that the rule is to apply to any and all elements of the document. But a special symbol more clearly states your intentions. The asterisk symbol (*) acts like a wildcard character to select all elements in the current document. You can use this to a greater advantage when you combine selector types, such as the universal and attribute selector. The following selector applies to all elements whose `align` attributes are set to a specific value:

```
*[align="middle"]
```

Child Selectors

Element containment is a key factor in the child selector. Again, following the notion of a style rule selector matching a pattern in a document, the child selector looks for element patterns that match a specific sequence of parent and child elements. The behavior of a child selector is very similar to that of a descendant selector, but the notation is different—a greater-than symbol (>) separates the element names in the selector, as in:

```
body > p {font-size: 12px}
```

Another difference is that the two elements on either side of the symbol must be direct relations of each other, as a paragraph is of a body, whereas the descendant selector means that the second symbol is nested at any level within the first.

Adjacent Sibling Selectors

An adjacent sibling selector lets you define a rule for an element based on its position relative to another element or, rather, the sequence of elements. Such adjacent selectors consist of two or more element selectors, with a plus symbol (+) between the selectors. For example, if your design calls for an extra top margin for an `h2` block whenever it immediately follows an `h1` element in the document hierarchy, the rule looks like the following:

```
h1 + h2 {margin-top: 6px}
```

Cascade Precedence Rules

By now it should be clear that there are many ways styles can be applied to an element—from an external style sheet file, from a `<style>` tag set, and from a `style` attribute in a tag—and there is the possibility that multiple style rules can easily apply to the same element in a document (intentionally or not). To deal with these issues, the CSS recommendation had to devise a set of rules for resolving conflicts among overlapping styles. These rules are intended primarily for the browser (and other user agent) makers, but if you are designing complex style sheets or are seeing unexpected results in a complex document, you need to be aware of how the browser resolves these conflicts for you.

The first point to be aware of is that a style sheet can originate from one of three sources

- User agent—typically the "default style sheet" that the browser uses to control presentation of elements lacking any other style rules
- User—some browsers let users override the default style sheet with a customized set of rules or preferences, which then act as the default style rules for all documents
- Author—style sheets that accompany content, including those that are embedded within the HTML page or linked in as external *.css* files

With one exception (see "Making a Declaration Important," later in this Online Section), a user agent rule is overridden by a user rule; in turn, an author rule overrides both the user and user agent rule. The bulk of the conflicts you'll encounter, however, are within the author rules that you associate with a document.

Conflict resolution is mostly a matter of assigning a relative weight to every rule that applies to a particular element. Rules with the most weight are the ones that most specifically target the element. At the lightweight end of the spectrum is the user agent rule. At the heavyweight end of the spectrum is the style rule that is targeted specifically at a particular element. This may be by way of an ID selector or the ultimate in specificity: a `style` attribute inside the tag. No run-of-the-mill style rule can override an embedded `style` attribute.

Between those two extremes are dozens of potential conflicts that depend on the way style sheets are defined for the document. As an example, a browser always has its own default style sheet so that it knows how to render standard elements. User-adjustable browser preferences (such as viewing in a larger font size) modify those default settings; some browsers also allow users to apply their own generic style sheets to override the default settings. But what happens when a document arrives with its own settings? Before rendering any style-sheet-capable element, the browser uses a process similar to the following decision path to determine how that element should be rendered:

1. Scan the document for any author-specified style declarations that have a selector that matches the element. If the element is not selected by any rules, short-circuit the rest of the decision path and render the element according to the user's style sheet, if present, or the browser's current built-in settings.
2. Sort all applicable declarations according to weight as indicated by a special `!important` declaration (see the following section). Declarations marked important are assigned greater weight than unmarked declarations.
3. Sort declarations again, this time by origin. If the same selector and style declaration is marked `!important` in both the document and the user's style sheet (if present), give precedence to the user's style (in other words, assign the greatest possible weight to user rules marked `!important`). For unmarked declarations, assign the greatest weight to the author's style, followed by the user's style (if present), and then the browser default style.
4. Now sort the unmarked declarations by the specificity of the rule's selector. The more specific the selector (see the section on selector specificity later in this chapter), the greater the weight assigned to that declaration.
5. Finally, if more than one declaration is assigned the same weight after previous sorting, sort one last time based on how close the rule is to the element in the document structure. The closest applicable rule with the greatest weight wins the conflict. Rules defined in multiple imported style sheets are defined in the order of the statements that trigger the import; an `@import` rule must be positioned before explicit rules in a `<style>` tag set and therefore carries less weight than the explicit rule; a rule defined in an element's `style` attribute is the closest and therefore heaviest unmarked (i.e., not `!important`) rule.

Making a Declaration Important

You can give an individual declaration within a rule an extra boost in its battle for superiority in the cascading order. When you do this to a declaration, the declaration is called the *important* declaration; it is signified by an exclamation mark (`!`) and the word `important` following the declaration. For example, in the following style sheet, the `margin-left` property for the `p` element is marked important:

```
<style type="text/css">
    p {font-size: 14px; margin-left: 2em !important; margin-right: 2em}
    p.narrow {color: red; margin-left: 5em; margin-right: 5em}
</style>
```

When the document encounters a <p> tag with a `class` attribute set to `narrow`, the left margin setting of the less specific <p> tag overrides the setting of the more specific `p.narrow` class because of the important declaration. Note that this is an artificial example because you typically would not include conflicting style rules in the same style sheet. The important declaration can play a role when a document imports one or more style sheets. If a generic rule for the specific document must override a more specific rule in an imported style sheet, the important declaration can influence the cascading order. A browser that correctly implements CSS2.x user style sheets treats a user's `!important` style declaration as the weightiest style rule. While a page designer might question why the user has the most power in the cascading battle, bear in mind that a user may employ styles tailored to special accessibility needs, such as extra-large font sizes or high-contrast color combinations that need to apply to all documents.

Determining a Selector's Specificity

The fourth cascading precedence rule refers to the notion of *specificity*, or how well a rule selector targets a particular element in a document. The CSS recommendation establishes an unseen, internal ranking system that assigns values to three categories, arbitrarily designated a, b, and c. These categories represent the counts of items within a rule selector, as follows:

a The count of ID selectors

b The count of other selector types

c The count of element types mentioned by name in the selector

For any rule selector, the browser adds up the total for each category and then concatenates the three totals to come up with a specificity value. Table III-4 displays a sequence of rule selectors in increasing specificity.

Table III-4. Specificity ratings for rule selectors

Rule selector	a	b	c	Specificity rating
`em`	0	0	1	1
`p em`	0	0	2	2
`div p em`	0	0	3	3
`em.hot`	0	1	1	11
`p em.hot`	0	1	2	12
`#hotStuff`	1	0	0	100

Browsers use the highest applicable specificity rating value to determine which rule wins any conflict. For example, if a style sheet defines the six rules for `em` elements shown in Table III-4 (with the `#hotStuff` rule being an ID selector), the browser applies the highest relevant specificity rating to each instance of the `em` element. An element with the tag `<em class="hot">` inside an `h1` element most closely matches the `em.hot` rule selector (specificity rating of 11), and therefore ignores all other selectors. But if the same `em` element is placed inside a `p` element, the more specific rule selector (`p em.hot`) wins.

Cross-Platform Style Differences

It has been a long battle among browser makers, standards bearers, and content authors, but browsers released in late 2005 and later do a pretty good job of implementing a workable complement of CSS2.1 features. That's not to say that all implementations are perfect. Advanced web designers continually push the envelope in finding new layout challenges that inevitably find weaknesses in one browser or another. And if you need to support an ever-growing collection of older browsers, the challenges can be enormous.

Listing every browser's style sheet anomaly is beyond the scope of this book. Given all the content combinations and unexpected interactions, such an up-to-date master list probably doesn't exist. Always check the developer release notes for a browser. The open source Mozilla browser offers public access to the internal bug tracking system (Bugzilla), which may help you validate a problem you're experiencing a current version of Firefox. For other browsers, you'll have to rely on developer exchanges in the many online forums and blogs scattered across the Web.

That such inconsistencies exist points to the fact that deployment of CSS style sheets across all DHTML-capable browsers requires testing on as many browser brands and operating systems as you can get your hands on. Carefully study the output on each to make sure that your design goals are met, even if the exact implementations don't match pixel for pixel on the screen.

ONLINE SECTION IV

Changing Page Content and Styles

As the name "Dynamic HTML" implies, the technology allows page authors to change HTML markup after the page has loaded from the server. Prior to the Version 4 browsers, your ability to script dynamic content was limited to controlling the HTML being written to the current page as the page initially loaded, loading HTML documents into other frames, altering form controls, and, in some browser versions, swapping same-size images. But when browsers such as IE 4 (and later), Mozilla-based browsers, and Safari expose every HTML element to a scriptable object model and automatically reflow pages, authors gain extraordinary powers to change anything on the page at any time.

This chapter provides an overview of the most common ways of dynamically changing content, including some techniques that work with older browsers. It also offers suggestions about how to develop code that accommodates the incompatibilities that exist between the IE-only and W3C DOMs. Compared to the contortions necessary in the IE 4 versus Netscape Navigator 4 days, most cross-browser dynamic content tasks today are a breeze.

Writing Variable Content

While a page is loading, you can use the JavaScript `document.write( )` or `document.writeln( )` methods to fill in content that cannot be stored as part of the document. Example IV-1 demonstrates a simple combination of hardwired HTML with dynamically written content to fill a page. In this case, the dynamically written content consists of properties that the client computer and browser can determine without burdening the server. The user is oblivious to the fact that a script creates some of the text on the page.

Example IV-1. Combining fixed and dynamic content in a rendered page

```
<html>
<body>
<h1>Welcome!</h1>
```

Example IV-1. Combining fixed and dynamic content in a rendered page (continued)

```
<hr>
<p>Your browser identifies itself to the server as:<br>
<script language="JavaScript" type="text/javascript">
document.write(navigator.userAgent + ".");
</script>
</p>
</body>
</html>
```

You can use `document.write( )` or `document.writeln( )` in scripts that execute while a document is loading, but you cannot use either method to modify only a portion of a page that has already loaded. Once a document has finished loading, if you make a single call to `document.write( )` directed at the current document, the call automatically clears the current document (and scripts) from the browser window and writes the new content to the page. So, if you want to rewrite the contents of a page, you must do so with just one call to the `document.write( )` method. Example IV-2 demonstrates how to accumulate content for a page in a variable that is written in one blast when the user clicks a button.

Example IV-2. Creating a new document for the current window

```
<html>
<head>
<title>Welcome Page</title>
<script language="JavaScript" type="text/javascript">
// create custom page and replace current document with it
function rewritePage(form) {
    // accumulate HTML content for new page
    var newPage = "<html>\n<head>\n<title>Page for ";
    newPage += form.entry.value;
    newPage += "</title>\n</head>\n<body bgcolor='cornflowerblue'>\n";
    newPage += "<h1>Hello, " + form.entry.value + "!</h1>\n";
    newPage += "</body>\n</html>";
    // write it in one blast
    document.write(newPage);
    // close writing stream
    document.close( );
}
</script>
<body>
<h1>Welcome!</h1>
<hr>
<form onsubmit="return false;">
<p>Enter your name here: <input type="text" name="entry" id="entry"></P>
<input type="button" value="New Custom Page" onclick="rewritePage(this.form);">
</form>
</body>
</html>
```

Notice that the script inserts data from the original screen's form into the content of the new page, including a new title that appears in the browser window's title bar. As a convenience to anyone looking at the source of the new document, escaped newline characters (`\n`) are inserted for cosmetic purposes only. After the call to `document.write()`, the `rewritePage()` function calls `document.close()` to close the writing stream for the new document. While there are also `document.open()` and `document.clear()` methods, we don't need to use them to replace the contents of a window. The one `document.write()` method clears the old content, opens a new output stream, and writes the content.

Both examples are written in such a way that they work in scriptable browsers back to Netscape 2. Thus, the `<script>` tags include both the old-fashioned `language` attribute, as well as the more modern `type` attribute. The `<form>` tag's `onsubmit` event handler simply prevents a press of the Enter key in the one-field form from "submitting" the form (which, because of the lack of an `action` attribute, would cause the page to reload itself).

Note, however, that Examples IV-1 and IV-2 require a script-enabled browser to display the dynamic content. You obviously don't want to use this technique for mission-critical page content if non-scriptable browsers may be used to access it.

Writing to Other Frames and Windows

You can also use the `document.write()` method to send dynamically created content to another frame in a frameset or to another browser window previously opened by a script in the same page. In this case, you are not restricted to only one call to `document.write()` per page; you can open an output stream to another frame or window and keep dumping stuff into it until you close the stream with `document.close()`.

All you need for this kind of content creation is a valid reference to the other frame or window. How you generate the frameset or secondary window influences this reference.

Framesets and Frames

A typical frameset document defines the physical layout of how the main browser window is to be subdivided into separate panels. Framesets can, of course, be nested many levels deep, where one frame loads a document that is, itself, a frameset document. The key to writing a valid reference to a distant frame is knowing the relationship between the frame that contains the script doing the writing and the target frame.

The most common frameset structure consists of one frameset document and two to four frames defined as part of that frameset (you can have more frames if you like, but not everyone is fond of frames). Ideally, you should assign a unique identifier to

the name attribute of each <frame> tag.* Example IV-3 is a basic frameset document that assigns a name to each of the three frames and loads an efficient local blank page into each frame. The technique used here is to invoke a function, blank( ), that exists in the frameset (parent) document. In each case, the javascript: pseudo-URL is applied to the newly created frame. From each frame's point of view, the blank( ) function is in the parent document, hence the parent.blank( ) reference. The 100-pixel wide frame down the left side of the browser window is for a navigation bar. The right portion of the window is divided into two sections. The upper section (arbitrarily called main) occupies 70 percent of the column, while the lower section (called instrux) occupies the rest of the column.

Example IV-3. A simple three-frame frameset with blank pages written to each frame

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
<!--
function blank( ) {
    return "<html></html>";
}
//-->
</script>
</head>
<frameset cols="100,*">
    <frame name="navBar" src="javascript:parent.blank( );">
    <frameset rows="70%,30%">
        <frame name="main" id="main" src="javascript:parent.blank( );">
        <frame name="instrux" id="instrux" src="javascript:parent.blank( );">
    </frameset>
</frameset>
</html>
```

Now imagine that a modified version of Example IV-2 is loaded into the main frame. The job of the script, however, is to write the dynamic content to the frame named instrux. To accomplish this, the reference to the other frame must start with the parent document (the frameset), which the two frames have in common. Example IV-4 shows the modified page that goes into the main frame and writes to the instrux frame. The two small changes that were made to the original code are highlighted.

Example IV-4. Writing dynamic content to another frame

```
<html>
<head>
```

* While XHTML 1.0 deprecates the name attribute for the frame element, older browsers ignore the id attribute. Due to the massive volume of framed web content that uses only the name attribute, browsers will support this attribute for many years to come. In the meantime, it is perfectly acceptable to assign the same identifier to a frame element's name and id attributes. This does not confuse scripts or link and form targets on any browser, and also validates as transitional and frameset XHTML 1.0.

Example IV-4. Writing dynamic content to another frame (continued)

```
<title>Welcome Page</title>
<script language="JavaScript" type="text/javascript">
// create custom page and replace current document with it
function rewritePage(form) {
    // accumulate HTML content for new page
    var newPage = "<html>\n<head>\n<title>Page for ";
    newPage += form.entry.value;
    newPage += "</title>\n</head>\n<body bgcolor='cornflowerblue'>\n";
    newPage += "<h1>Hello, " + form.entry.value + "!</h1>\n";
    newPage += "</body>\n</html>";
    // write it in one blast
    parent.instrux.document.write(newPage);
    // close writing stream
    parent.instrux.document.close( );
}
</script>
<body>
<h1>Welcome!</h1>
<hr>
<form onsubmit="return false;">
<p>Enter your name here: <input type="text" name="entry" id="entry"></p>
<input type="button" value="New Custom Page" onclick="rewritePage(this.form);">
</form>
</body>
</html>
```

If, on the other hand, you simply want to load a different document from the server into the `instrux` frame, you can use a scriptless HTML link and set the `target` attribute to the `instrux` frame. A script in `main` can also specify a document for the `instrux` frame as follows:

```
parent.instrux.location.href = "nextPage.html";
```

Secondary Windows

Legacy browser object models provide facilities for not only generating a new browser window, but also setting the new window's size and (in Version 4 browsers and later) location on the screen. You can then use references to communicate from one window to the other, although the form of those references is quite different, depending on where the script is running.

The `window.open( )` method that generates a new window returns a reference to the new `window` object. If you plan to communicate with that window after it has been opened, you should store the reference in a global variable. This reference is the only avenue that scripts may use to access the subwindow. Example IV-5 features a script for opening a new window and writing to it. In addition, it also inserts a brief delay to allow the often sluggish Internet Explorer for Windows to finish creating the window before writing to it, and brings an already opened but hidden window to the front, if the browser supports that feature (Navigator 3 or later and IE 4 or later).

Multiple Window Cautions

Thanks to abusive behavior by advertisers over the years, web surfers are generally annoyed with any page that automatically spawns yet another browser window. Autocratic page designers are just as guilty of causing problems: they are so protective of their sites, that a link to any destination outside of the current site automatically opens in another window, whether the visitor wants it or not.

Bear in mind that smart users know that they have the choice of opening any link in a new window via the contextual menu available on a standard link (accessed by right-clicking in Windows or Ctrl-clicking on the Macintosh). Alternatively, you can add a `title` attribute to a link so that if the user hovers the pointer atop the link long enough, the resulting tooltip displays an appropriate advisory (e.g., "Opens whitehouse.gov in a separate window").

Also be aware that if you attempt to use scripts to open windows automatically as the main page loads or unloads, browser popup window blockers will likely prevent your script from accomplishing its task. For these and other reasons (e.g., frequent IE security blocks between windows), I recommend against designing for multiple windows unless absolutely necessary.

Example IV-5. Opening a new window and writing to it

```
<html>
<head>
<title>A New Window</title>
<script language="JavaScript" type="text/javascript">
// Global variable for subwindow reference
var newWindow;
// Generate and fill the new window
function makeNewWindow( ) {
    // make sure it isn't already opened
    if (!newWindow || newWindow.closed) {
        newWindow = window.open("","sub","status,height=200,width=300");
        // delay writing until window exists in IE/Windows
        setTimeout("writeToWindow( )", 500);
    } else if (newWindow.focus) {
        // window is already open and focusable, so bring it to the front
        newWindow.focus( );
    }
}
function writeToWindow( ) {
    // assemble content for new window
    var newContent = "<html><head><title>Sub Window</title></head>\n";
    newContent += "<body>\n<h1>This is a new window.</h1>\n";
    newContent += "</body>\n</html>";
    // write HTML to new window document
    newWindow.document.write(newContent);
    newWindow.document.close( ); // close layout stream
}
```

Example IV-5. Opening a new window and writing to it (continued)

```
</script>
</head>
<body>
<form>
<input type="button" value="Create New Window"
 onclick="makeNewWindow( );">
</form>
</body>
</html>
```

Example IV-5 shows that the reference to the subwindow (stored in the `newWindow` global variable) can be used to call `document.write( )` and `document.close( )` for that window. The `newWindow` object reference is the gateway to the subwindow.

A script in a document loaded into a subwindow can communicate back to the window or frame that spawned the new window. Every scriptable browser (except Navigator 2) automatically sets the `opener` property of a new window to a reference to the window or frame that created the window (recent browsers also set this property to windows launched by `a` and `form` element `target` attributes). Therefore, to access the `value` property of a form's text box (named `entryField`) located in the main browser window, you can use the following script statement in the subwindow:

```
opener.document.forms[0].entryField.value
```

Remember that `opener` refers directly to the window or frame (as a `window` object) that spawned the subwindow. If you need to access content in another frame in the host window's frameset, your reference must traverse the object hierarchy accordingly:

```
opener.parent.otherFrameName.document.forms[0].someField.value
```

Image Swapping

As precursors to true DHTML powers of more recent browsers, Navigator 3 (and Internet Explorer 3.01 for the Macintosh only) gave us a glimpse of things to come with image swapping. The basis for this technique is a document object model that defines an `img` element as an object whose properties can be changed on the fly. One of those properties, `src`, defines the URL of an image loaded initially by virtue of an `<img>` tag and currently displayed in the page. Change that property and the image changes, within the same rectangular space defined by the `<img>` tag's `height` and `width` attributes (or, lacking those attribute settings, the first image's dimensions as calculated by the browser), while all the other content around it stays put. In browsers that reflow dynamic content, you can swap images of different sizes if you like, and the surrounding content adjusts its layout accordingly.

Working in tandem with the `img` element object is the static `Image` object from which new "virtual" images can be created in the browser's memory with the help of

scripts. These kinds of images do not appear in the document, but can be scripted to preload images into the browser's image cache as the page does its original download. Thus, when it comes time to swap an image, the switch is nearly instantaneous because there is no need for network access to grab the image data.

The example in this section shows you how to precache and swap images for the buttons of an imaginary navigation menu. There are three button-looking images—**Home**, **Catalog**, and **About Us**. As the user rolls the mouse atop a button, a highlighted ("on") version of the button appears in the image space; as the mouse rolls off the button, the original unhighlighted ("off") version reappears. Importantly, the code is written so that standard HTML markup is used for the links, and only browsers supporting the necessary scripting facilities add the rollover feature. Here is the HTML that will be addressed by the rollover scripts:

```
<a href="index.html"><img id="home" class="swappable"
    height="20" width="80" border="0" src="img/home_off.jpg" alt="Home"></a>
<a href="catalog.html"><img id="catalog" class="swappable"
    height="20" width="80" border="0" src="img/catalog_off.jpg" alt="Catalog"></a>
<a href="about.html"><img id="about" class="swappable"
    height="20" width="80" border="0" src="img/about_off.jpg" alt="About Us"></a>
```

The only noteworthy items about this HTML is that each `img` element has its unique ID (whose identifier is part of the image's file name for both "on" and "off" versions) and that all share the same `class` attribute value (`swappable`). Scripts that modify the behaviors of the images identify their targets by their shared class name.

Preparing for the Initialization

To keep the scripting away from the markup, the initialization routine occurs after the page loads, that is, when the window's `load` event fires. The mechanism used in this example (described more fully in Online Section VI) employs a custom API function to accommodate both the W3C DOM and Microsoft event models. The `addEvent( )` function (which can be invoked by any other routines in the page's scripts) handles the browser differences. For this example, a call to the `init( )` function is bound to the window's `load` event as follows:

```
function addEvent(elem, evtType, func, capture) {
   capture = (capture) ? capture : false;
   if (elem.addEventListener) {
      elem.addEventListener(evtType, func, capture);
   } else if (elem.attachEvent) {
      elem.attachEvent("on" + evtType, func);
   }
}

function init( ) {
   // initialize rollover object
   rollover.init( );
}
```

```
addEvent(window, "load", init);
```

Defining a "Rollover" Object

All modification to the `img` elements occurs in the `rollover` object, shown in Example IV-6. Among the tasks assumed by this single object are the following:

- Creating an associative array of offscreen image objects
- Precaching the "on" images
- Binding mouse event handlers to affected images
- Handling mouse events

The `rollover` object is created as the page loads, and is initialized after the page has loaded (via a call to `rollover.init( )`).

Example IV-6. A generic rollover object

```
// rollover object
var rollover = {
   // image directory path
   imgPath : "img/",
   // create associative arrays for images
   imagesNormal : new Object( ),
   imagesHilite : new Object( ),
   // invoked by img mouse events
   setImage : function(elemID, img) {
      document.getElementById(elemID).src = img;
   },
   // initialize
   init : function( ) {
      if (document.getElementById) {
         var imgID;
         var imgs = document.body.getElementsByTagName("img");
         for (var i = 0; i < imgs.length; i++) {
            if (imgs[i].className == "swappable") {
               // fill arrays and pre-cache "_on" images
               imgID = imgs[i].id;
               this.imagesNormal[imgID] = new Image( );
               this.imagesNormal[imgID].src = this.imgPath + imgID + "_off.jpg";
               this.imagesHilite[imgID] = new Image( );
               this.imagesHilite[imgID].src = this.imgPath + imgID + "_on.jpg";
               // assign image swapping event handlers to img elements
               addEvent(imgs[i], "mouseover", new Function("rollover.setImage('" +
                   imgID + "','" + this.imagesHilite[imgID].src + "');"));
               addEvent(imgs[i], "mouseout", new Function("rollover.setImage('" +
                   imgID + "','" + this.imagesNormal[imgID].src + "');"));
            }
         }
      }
   }
}
```

The imgPath property contains a string with the path (relative or absolute, as needed for your server directory structure) to the directory containing all images used in the rollover effect. Two associative arrays (imagesNormal and imagesHilite) are initialized as empty objects whose properties will be filled when the init( ) method runs. The setImage( ) method is the one called by both the mouseover and mouseout events of the rollover images. This method receives an ID of an img element and the URL of the image to be assigned to the element's src property.

Most of the hard work occurs in the init( ) method. Limited to browsers that support the W3C DOM document.getElementById( ) method, it begins by obtaining an array of all img elements in the document. Next it loops through them all in search of those whose class attribute is swappable. It uses the ID of each image to assemble the URL of each image file for both states of each button. Assigning a URL to the src property of an instance of an Image object causes the browser to download the image into the image cache. Having the image already in the cache avoids a delay in the first display of the "on" image. Lastly, each img element has two event handlers bound to it. Because the event handlers will be invoked from outside of the context of the rollover object, the call to the setImage( ) method includes an explicit reference to the rollover object. Note that event binding occurs only in browsers that support the basic W3C DOM syntax; if the events had been assigned as attributes within the <a> or <img> tags, older browsers would react to those events, and your scripts would have to perform compatibility filtering with each event firing. With the approach in Example IV-6, older browsers don't respond to those events at all (yet the links work just fine for everyone).

Rollovers and the Status Bar

Versions of image rollover scripts in previous editions of *Dynamic HTML: The Definitive Reference*—in addition to being more invasive in the markup—included a routine for modifying the text that displayed in the browser's status bar during a mouse rollover. The ability to display status bar text other than the actual URL of a link has been removed from most modern browsers.

Although one can imagine good uses of that power (e.g., providing a more friendly message about the destination), the capability has been abused by those who prefer to trick users. It's not uncommon, for example, for a phishing email message delivered in HTML to include a link whose in-page text looks to be a trusted site, but whose href attribute points to a different site that tries to install malware onto all visitors' PCs. It is too easy to code such a link with mouseover events that modify the window.status property to display the trusted site's URL, rather than the URL of the actual destination. To prevent this type of spoofing from tricking unsuspecting users, browser makers have simply disabled this scriptable feature.

Internet Explorer Caching Issues

If image preloading does not appear to work for your pages in IE for Windows, your server may be returning HTTP 1.1 headers that instruct the browser to check with the server each time an image URL is changed. This can cause severe thrashing behavior with mouse rollovers because the `mouseover` event fires repeatedly while the mouse is atop the element, forcing repeated server requests.

You may be able to override the server settings by including a `meta` element in your document that specifies an extended expiration date, such as the following:

```
<meta http-equiv="EXPIRES" content="Fri 31 Dec 2010 23:00:00 GMT">
```

But controlling caching from the server is a better solution.

CSS-Only Image Swaps

You can also fashion a variety of "rollover" effects using Cascading Style Sheet rules without any scripting. The images are not put into the markup as `<img>` tags, but rather as background images to links styled as block-level elements. In lieu of scripted mouse events, you code the style rules to use the `:hover` pseudo-class (and, optionally, `:active` for the equivalent of a `mousedown` event) to change the background image URL.

You can also take advantage of CSS—along with carefully crafted images—to eliminate the need for precaching images. A single image file contains both the normal and highlighted versions of the button, side-by-side. By shifting the position of the background image during a hover action, the browser appears to be swapping the image.

The following HTML markup serves as the basis for one solution to the CSS approach:

```
<a href="index.html" id="csshome" class="cssswappable"></a>
<a href="catalog.html" id="csscatalog" class="cssswappable"></a>
<a href="about.html" id="cssabout" class="cssswappable"></a>
```

In this case, there is no content for the a elements, which could be a problem for older browsers and search engine crawlers. But your design can also be such that button labels are comprised of HTML text, if you can find font style properties that work across a wide range of browsers.

All the rest of the work occurs in the CSS rules, as follows:

```
<style type="text/css">
a.cssswappable {
   display: block;
   text-decoration: none;
   height: 70px;
   float: left;
   margin-right: 4px;
}
```

```
#csshome {
   width: 147px;
   background-image: url(img/home.png);
   background-position: left;
}
#csshome:hover {
   background-position: right;
}
#csscatalog {
   width: 189px;
   background-image: url(img/catalog.png);
   background-position: left;
}
#csscatalog:hover {
   background-position: right;
}
#cssabout {
   width: 209px;
   background-image: url(img/about.png);
   background-position: left;
}
#cssabout:hover {
   background-position: right;
}
</style>
```

When designing pages that rely on the `:hover` pseudo-class, keep one important compatibility issue in mind: Internet Explorer prior to Version 7 supports the mouse-related pseudo-classes only for `a` elements. Other modern browsers (including IE 7 in standards-compatible mode only) support them for all types of rendered elements.

You can mix CSS and scripted rollover techniques to match your markup and accessibility requirements. For example, if you prefer to use an anchor (`a`) element around an `img` element, you can use a combination of `mouseover`/`mouseout` events for the `a` element that trigger scripts that alter CSS background image position for an `img` element that contains a transparent image.

Changing Tag Attribute Values

The DOMs in IE 4 and later, Mozilla, Safari, and Opera 5 and later expose a tag's attributes as properties of the corresponding scriptable object. Property names tend to mimic attribute names, unless the attribute name contains a hyphen or any other "illegal" ECMAScript identifier character. Some properties are read-only, but the vast majority are read-write. If a new value impacts the appearance of the element, the change occurs, and surrounding content adjusts its layout to fit the new arrangement.

As suggested in Online Section II, I recommend coding changes to HTML markup for only those browsers that include support for the W3C DOM's `document.getElementById()` method for referencing elements. However, if you find that you

must also include support for IE 4, you can support both referencing models with a simple decision tree in functions that receive an element's ID as an argument. The following shows a typical structure:

```
function myFunc(elemID) {
     var elem = (document.getElementById) ? document.getElementById(elemID) :
                ((document.all) ? document.all[elemID] : null);
     if (elem) {
        // work on element object here
    }
}
```

You also have the option of creating your own custom API function that returns an object reference regardless of object model, as in the following example:

```
function getElem(elemID) {
    return (document.getElementById) ? document.getElementById(elemID) :
           ((document.all) ? document.all[elemID] : null);
}
```

Even with element object referencing solved, another syntactical issue arises. Strictly speaking (that is, according to W3C DOM guidelines), attribute values should be read via an element object's `getAttribute( )` method and set via `setAttribute( )`, both of which are detailed in Chapter 2 of *Dynamic HTML,* Third Edition. Both the IE and W3C DOMs support these methods (plus an extension for both methods in IE). And yet, the same DOMs expose all of these attributes as object properties.

It can be said without much hesitation that the W3C DOM's authors appear to have been more concerned with the internal form and structure of the DOM than with practical aspects of using the syntax in real application development. Otherwise, element references would use syntax more compact than the finger-twisting `document.getElementById( )` method. The same can be said for reading and writing attributes the "right" way, which is quite download-byte-intensive, as opposed to modifying the properties directly, which is an acceptable, compact way that is supported by all browsers.

To compare the two approaches, we can view a typical value adjustment in each style. The following function toggles between two `size` attribute settings for a text box using the long-accepted property approach for reading and writing an element's property:

```
function toggleWidth(elemID) {
    var elem = document.getElementById(elemID);
    if (elem) {
        var big = 80, small = 20;
        elem.size = (elem.size == small) ? big : small;
    }
}
```

Now compare the same function written with the attribute methods and the revised data types for attribute values:

```
function toggleWidth(elemID) {
    var elem = document.getElementById(elemID);
    if (elem) {
        var big = "80", small = "20";
        elem.setAttribute("size", ((elem.getAttribute("size") == small) ?
                          big : small));
    }
}
```

For a simple function like this, the impact of the extra script characters isn't too severe. But the bytes can add up to lengthy scripts.

More problematic, however, are the frequent problems using `setAttribute( )` in Internet Explorer. Attempting to set some attributes—especially those involving new URLs for `src` attributes intended to load new external content—doesn't work in IE 6 or earlier. A more serious issue, however, is that Microsoft implements the `setAttribute( )` method in a slightly non-standard way. In particular, the method's first parameter (the name of the attribute to be set) has to be in the form not of the actual attribute name, but of the corresponding scriptable property name. Although a lot of attribute and property names are the same, some important attribute names conflict with ECMAScript reserved words. For example, to assign a new value to the `class` attribute in IE, you must refer to it as `className`, its scripted equivalent. The W3C DOM, however, requires the actual attribute name. Therefore, to use `setAttribute( )` religiously throughout your code, you'll have to be aware of occasional branching needed for IE to make sure you pass the correct name as the first parameter. Therefore, despite many standards-centric programmers' preference for the `setAttribute( )` method, I find the property approach to be more reliable and compatible across browser object models. For that reason and the sake of brevity, most examples in this book use direct property access. My advice is to use the approach with which you are most comfortable, and stay with it throughout your scripts, but always testing your choice thoroughly in IE to make sure the method works as you require.

Changing Applied Style Values

With so much of the rendering detail for an element in the hands of style sheets, you will likely modify style values to control a variety of DHTML effects. The IE and W3C DOMs provide several ways to adjust the value of style properties being applied to an element at any given moment. The two most popular are modifications via an element's `style` and `className` properties; other approaches dive more deeply into the style sheets and their components.

The style Property

Perhaps the simplest way to adjust a single style value is through an element's `style` property. An element's `style` property is, itself, an object whose properties consist of all possible style properties supported by the browser's DOM.

Before going further, it is helpful to acknowledge the similarities and differences between the `style` objects implemented in IE for Windows and the W3C DOM (as implemented in IE 5/Mac, Mozilla, Safari, Opera, and others). Typically, browsers that implement proprietary style properties to extend the list provided by the CSS standard also expose those proprietary style properties as scriptable properties. For example, while CSS2 does not include an attribute to specify the opacity of a text element, IE and Mozilla (prior to 1.7.2) implement their own extensions for this feature (the IE `opacity` filter attribute and the Mozilla `mozOpacity` attribute). If you implement scripted styles across DOMs, be sure the styles and values you use are supported by both DOMs (as detailed in Chapter 4 of *Dynamic HTML,* Third Edition) or that you provide suitable workarounds.

Deeper down, however, the IE and W3C `style` objects have different structures. In the W3C DOM's formal structure, what you think of as a `style` object is known as a `CSSStyleDeclaration` object (terminology that scripts don't work with directly). This object features a variety of methods for reading and writing style property values that are very much in keeping with the rest of the formal DOM. For example, setting the `color` style property of an element looks like the following:

```
document.getElementById("myP").style.setAttribute("color", "rgb(255, 0, 0)", "");
```

Fortunately, the DOM standard also offers a convenient collection of properties that lets scripts access CSS properties just as IE has been doing all along. (This collection is of DOM type `CSS2Properties`, which is not a requirement for DOM compliance, but is supported in most non-IE browsers just the same.) This collection leads to the more common style adjustment syntax as in the following:

```
document.getElementById("myP").style.color = "rgb(255, 0, 0)";
```

The significance of understanding these inner workings comes forward when considering the data types of values assigned to `style` object properties. The implementation of the `CSS2Properties` collection supports string values exclusively, even if the values consist only of numbers. This makes sense in most cases, because even length values typically have numeric values followed by unit types, such as `"14px"`.

But also be aware that several IE-only `style` object attributes require either numeric or Boolean values. For example, IE's `style.posLeft` property consists of only the numeric portion of the string-based `style.left` property.

We'll now examine one way of cycling a chunk of text through a sequence of colors by adjusting style properties. Example IV-7 shows a version using W3C DOM referencing syntax. A single `span` element in the body has the `color` property of its style

changed in a function that is invoked often enough (via `setInterval( )`) to run through the cycle seven times, and then stop (to prevent user nausea). For programming convenience, the color names are stored in a global variable array, with other global variables maintaining a record of the color currently showing and the elapsed number of color changes. No positioning or other tactics are required.

Example IV-7. Inline text color change via the style property

```
<html>
<head>
<title>Changing style Properties</title>
<style type="text/css">
    #hot1 {color: red}
</style>
<script type="text/javascript">
// Set global variables
var totalCycles = 0;
var currColor = 0;
var colors = ["red", "green", "yellow", "blue"];
var intervalID;
// Advance the color by one
function cycleColors( ) {
    // reset counter to 0 if it reaches 3; otherwise increment by 1
    currColor = (currColor == 3) ? 0 : ++currColor;
    // set style color to new color from array
    document.getElementById("hot1").style.color = colors[currColor];
    // invoke this function again until total = 27 so it ends on red
    if (totalCycles++ < 27) {
        intervalID = setTimeout("cycleColors( )", 100);
    } else {
        clearTimeout(intervalID);
    }
}
// start the ball rolling after page load
window.onload = cycleColors;
</script>
</head>
<body>
<h1>Welcome to the <span id="hot1">Hot Zone</span> Web Site</h1>
<hr>
</body>
</html>
```

Any valid color string value could work in this code. Chapter 4 of *Dynamic HTML*, Third Edition explains color value formats for `style` attributes.

The className Property

Another popular way to modify styles applied to an element is to predefine rules for more than one class selector, and then use scripts to switch the selector assigned to

an element's `className` property. This approach is particularly convenient if the changes you wish to make affect multiple style properties.

Example IV-8 builds upon Example IV-7 by defining four style rules, each with a different class selector. The array for this version contains not color values, but class names. The repeated call to `cycleColors( )` then simply assigns the next name in sequence to the `span` element's `className` property. Each change of the `className` property changes both the `color` and `border` style properties of the `span` element.

Example IV-8. Inline style change via theclassName property

```
<html>
<head>
<title>Changing className Properties</title>
<style type="text/css">
    .red    {
              color: red;
              border: 2px solid red;
            }

    .green  {
              color: green;
              border: 2px solid yellow;
            }

    .yellow {
              color: yellow;
              border: 2px solid blue;
            }

    .blue   {
              color: blue;
              border: 2px solid green;
            }
</style>
<script type="text/javascript">
// set global variables
var totalCycles = 0;
var currColor = 0;
var classes = ["red", "green", "yellow", "blue"];
var intervalID;
// advance the color by one
function cycleColors( ) {
    // reset counter to 0 if it reaches 3; otherwise increment by 1
    currColor = (currColor == 3) ? 0 : ++currColor;
    // set style color to new color from array
    document.getElementById("hot1").className = classes[currColor];
    // invoke this function again until total = 27 so it ends on red
    if (totalCycles++ < 27) {
        intervalID = setTimeout("cycleColors( )", 100);
    } else {
        clearTimeout(intervalID);
    }
```

Example IV-8. Inline style change via theclassName property (continued)

```
}
window.onload = cycleColors;
</script>
</head>
<body>
<h1>Welcome to the <span class="red" id="hot1">Hot Zone</span> Web Site</h1>
<hr>
</body>
</html>
```

Other Techniques

That the IE and W3C DOMs expose not just `style` elements as objects, but also their content as `styleSheet` objects (except in Opera), might lead some scripters to get perhaps too creative in modifying applied styles. Each `<style>` or `<link rel="stylesheet">` tag in a document creates a `styleSheet` object that is accessible through the `document.styleSheets` array. If you specify just one `<style>` tag in the document, `document.styleSheets[0]` returns a reference to that `styleSheet` object.

From that reference, a script can inspect and modify the contents of the style sheet, albeit via occasionally different syntax for IE and W3C DOMs (IE 5/Mac observes both syntaxes). Table IV-1 lists the most important `styleSheet` object properties implemented in the two DOMs.

Table IV-1. styleSheet object properties comparison

W3C property	Description	IE property
`cssRules`	Array of rules within the style sheet	`rules`
n/a	Complete text of all rules	`cssText`
`disabled`	Boolean to enable/disable entire style sheet	`disabled`
`href`	URL for `<link>`	`href`
`ownerNode`	Reference to `style` or `link` element	`owningElement`

Each `styleSheet` object has a property that returns an array of rules that belongs to the style sheet. The IE DOM calls these objects `rule` objects; the W3C calls them `cssRule` objects. You can reference a rule via its numeric index within the array. For example, here's a reference to the third rule in a page's only style sheet via the W3C DOM:

```
var oneRule = document.styleSheets[0].cssRules[2];
```

In IE syntax (required for IE/Windows), the expression is:

```
var oneRule = document.styleSheets[0].rules[2];
```

Properties of an individual rule object (regardless of how you reference the object) are a bit thin, and only partially helpful across DOMs. Both models support the `selectorText` property, which returns a string of the selector for the rule (although IE returns tag type selectors in uppercase, regardless of the source code case). Only the W3C `cssRule` object provides a direct `cssText` property to get the actual text. But both DOMs provide a `style` property, which returns a `style` (or W3C `CSSStyleDeclaration`) object whose properties reveal the individual style property settings for the rule. The following function demonstrates the cross-DOM syntax that sets the `font-size` attribute of a `p`-selectored rule in a style sheet:

```
function setPSize(n, units) {
    var sheets = document.styleSheets[0];
    var ruleList = (typeof sheets.cssRules != "undefined") ? sheets.cssRules :
    ((typeof sheets.rules != "undefined") ? sheets.rules : null);
    if (ruleList) {
        for (var i = 0; i < ruleList.length; i++) {
            if (ruleList[i].selectorText.toLowerCase( ) == "p") {
                ruleList[i].style.fontSize = n + units; // e.g., "px"
                break;
            }
        }
    }
}
```

Additional facilities for modifying `styleSheet` objects include methods for adding and deleting individual rules within the `styleSheet` object (with incompatible syntax for IE and W3C DOMs). You can learn the details in Chapter 2 of *Dynamic HTML*, Third Edition.

Despite the substantial flexibility available through the `styleSheet` object, you will experience better reliability and compatibility (especially with Opera) if you perform your style changes via the `style` or `className` properties of the element you wish to modify. Working with `styleSheet` objects is best left to those times when scripts are creating new style sheets or rules on the fly.

Changing Content

The W3C DOM's node-centric structure has the greatest impact on the way scripts modify text and element content in a document. Those scripters who learned DHTML under the element-centric Microsoft aegis can easily find themselves lost amid the new concepts that the W3C DOM imposes. While the W3C DOM makes a great deal of sense in a world tending toward XML (including the XML-flavored version of HTML), even experienced DHTML scripters soon discover that Microsoft implements many convenience features in its DOM that simplify DHTML scripting. Many of these conveniences, however, are not (or at least not yet) part of the released W3C DOM recommendations.

This state of affairs leaves browser makers, such as Mozilla, in an awkward position. On the one hand, browser makers want to produce the most standards-compliant browsers on the Web. But to do so would require that developers for their platform not only rewrite tons of scripts, but also master new, and seemingly complex, ways of carrying out tasks that a nonstandard DOM handles with ease. What's a browser maker to do?

Browser makers could invent their own extensions to the W3C DOM paradigm to bypass the complexities. Or they could yield to developer pressure and implement the popular, but nonstandard, techniques found in other browsers, as convenient alternatives to the ways cast in W3C stone. In the case of the Mozilla, Safari, and Opera browsers, they do a little of both. Thus, the syntactic and conceptual paths you wish to follow are entirely up to you. In this section, you will see how to use the IE and W3C DOM ways of modifying the text inside an element and the elements themselves. Your ultimate choice will depend on factors such as the browser platform(s) you must support, your dedication to standards, and your own programming practices.

Changing Element Text

Element text is nothing more than tagless content that resides inside an HTML container, such as a `p`, `span`, or `td` element. The tag provides the context for whatever words comprise the text. The IE DOM treats the text content as a property of an element object; the W3C DOM treats that same text as an object unto itself (as well as a property of an element object in DOM Level 3).

IE text

Every IE DOM container element object has an `innerText` property. The value of this read-write property is a string data type. You can use an assignment operator to place new text inside the container:

```
elementReference.innerText = "Your new text here.";
```

Assigning a value to this property with the `=` operator completely replaces its original content with the new text. You can also append text by using the `+=` assignment operator. Style sheet rules that apply to the element govern the new text, just as they did for the original text.

You need to exercise care in using the `innerText` property, especially if the element contains not only text, but other nested elements. When you read the `innerText` property, it ignores tags inside the element, returning all text, including text of nested elements without their tags. Similarly, if you assign a string of text to the `innerText` property of an element that contains other HTML elements, those nested elements get wiped out in the process. Therefore, it's best to use this property on elements you know for certain contain only text.

A companion property, innerHTML, forces the container to treat the newly assigned string as if it were tagged HTML text. Although the innerHTML property is primarily for altering elements (as well as text), it's important to understand the differences between innerText and innerHTML. To help you visualize the differences between these properties, let's start with a nested pair of elements as they appear in a document's source code:

```
<p id="par1" style="font-style:normal">
    A fairly short paragraph.
</p>
```

Focus on the p element, whose properties will be adjusted in a moment. The inner component of the p element consists of the string of characters between the start and end tags, but not including those tags. Any changes you make to the inner content of this element still have everything wrapped inside a p element.

How an element's inner component responds to changes depends on whether you direct the element to treat the new material as raw text or as text that may have HTML tags inside (i.e., innerText or innerHTML). To demonstrate how these important nuances affect your work with these properties, the following sequence starts with the p element shown earlier, as it is displayed in the browser window. Then comes a series of statements that operate on the original element, alternating with the representation of the element as it appears in the browser window after each statement.

A fairly short paragraph.

```
document.all.par1.innerText = "How are <em>you</em>?";
```

How are you?

```
document.all.par1.innerHTML = "How are <em>you</em>?";
```

How are *you*?

Adjusting the inner material never touches the <p> tag, so the normal font style prevails, and no matter how often you modify the property values, the reference to the p element remains valid because the element is always there. Setting the innerText property tells the browser to render the content literally, without interpreting the tags; setting innerHTML tells the browser to interpret the tags, which is why the word "you" is in italics after the second statement. Mozilla, Safari 1.2 (and later), and Opera implement the IE innerHTML property of all container elements as a convenience to scripters. If the string you assign to the property contains no HTML elements, the result is the same as if the property were innerText. Thus, the one innerHTML property serves two purposes.

Another Microsoft invention is the insertAdjacentText( ) method of element objects, defined as follows:

```
insertAdjacentText(where, text)
```

This method assumes you have a valid reference to an existing element and wish to add content to the beginning or end of the element without disturbing existing text. The precise insert position for these methods is determined by the value of the *where* parameter. There are four choices:

`BeforeBegin`
: In front of the start tag of the element

`AfterBegin`
: After the start tag, but immediately before existing text content of the element

`BeforeEnd`
: At the very end of the content of the element, just in front of the end tag

`AfterEnd`
: After the end tag of the element

Notice that the `BeforeBegin` and `AfterEnd` locations are outside of the element referenced in the statement. For example, consider the following nested pair of tags:

```
<span id="outer" style="color:red">
    Start outer text.
        <span id="inner" style="color:blue"> Some inner text.</span>
    End of outer text.
</span>
```

Now consider the following statement:

```
document.all.inner.insertAdjacentText("BeforeBegin", "Inserted!");
```

The document changes so that the word "Inserted!" is rendered in a red font. This is because the text was added before the beginning of the `inner` item, and is therefore under the rule of the next outermost container: the `outer` element.

The `insertAdjacentText( )` method was implemented for the first time in IE 4, in anticipation of what the unfinished W3C DOM was to be. But the W3C DOM took a different turn, so a number of Microsoft content manipulation inventions work only in IE (and some only in Windows versions). Table IV-2 provides a summary listing of the proprietary element object methods for a variety of text and element actions.

Table IV-2. IE element content manipulation methods

Method	Description
`contains(`*`elemRef`*`)`	Returns Boolean `true` if current element contains *`elemRef`*
`getAdjacentText(`*`where`*`)`	Returns text sequence from position *`where`* (IE 5 and later for Windows only)
`insertAdjacentElement(`*`where, elemRef`*`)`	Inserts new element object at position *`where`* (IE 5 and later for Windows only)

Table IV-2. IE element content manipulation methods (continued)

Method	Description
`insertAdjacentHTML(where, HTMLText)`	Inserts text (at position *where*) which gets rendered as HTML
`insertAdjacentText(where, text)`	Inserts text (at position *where*) as literal text
`removeNode(deep)`	Deletes element or text node (and its child nodes if *deep* is `true`)
`replaceAdjacentText(where, text)`	Replaces current text at position *where* with *text* (IE 5 and later for Windows only)
`replaceNode(newNodeRef)`	Replace current node with new node (IE 5 and later for Windows only)
`swapNode(otherNodeRef)`	Exchange current node with *otherNodeRef*, and return reference to removed node (IE 5 and later for Windows only)

While all of these methods do their jobs in the IE versions that support them, they have counterparts or equivalent functionality in the W3C DOM, albeit with different syntax. IE 5 and later (both Windows and Mac) support the bulk of the W3C DOM versions of these methods, so there is little need to master both sets. For cross-DOM development, you are better served using the W3C DOM versions exclusively.

W3C DOM text

Absolutely everything in a document is an object of some kind in the eyes of the W3C DOM. As described in Online Section I, the fundamental type of object in a W3C DOM document is the *node*. A document's structure can be described as a tree of nodes of various types. Each node object has a `nodeType` property that is one of twelve possible values (numbered 1 through 12). All nodes that represent a document's content grow from the root `document` node (a `nodeType` of 9). An element is another type of node (`nodeType` of 1), as is a text node (`nodeType` of 3) between the start and end tags of an element container.

Adjacent nodes bear parent-child-sibling relationships, the understanding of which is crucial to successful application of W3C node concepts. Consider the following series of element and text nodes:

```
<p id="myP">Where is <em id="myEM">Amy</em> today?</p>
```

The `p` element node has three child nodes. The first and third child nodes are text nodes, while the middle one is an element node (the `em` element). That `em` element, itself, has one child node—a three-character text node. The attributes in the two tags are themselves nodes (`nodeType` of 2), but attribute nodes are not part of the element and text node parent-child relationship model.

Each node object (regardless of type) has a set of properties that help scripts obtain references to adjacent nodes and read or write values associated with the node. Table IV-3 lists the common properties of every node object.

Table IV-3. Common W3C DOM node object properties

Property	Value type	Description
nodeName	String	Name associated with the node or node type
nodeValue	String	Value associated with the node (read-write)
nodeType	Integer	One of the 12 node types
parentNode	Object	Reference to next outermost container node
childNodes	Array (NodeList)	Child nodes in source code order
firstChild	Object	Reference to first child node
lastChild	Object	Reference to last child node
previousSibling	Object	Reference to preceding node at same generation
nextSibling	Object	Reference to next node at same generation
attributes	NamedNodeMap	Collection of attribute nodes
ownerDocument	Object	Reference to root document node

Of the properties listed in Table IV-3, the first three return important information, but their values depend upon the type of node. Table IV-4 lists the most common node types found in HTML documents and the kinds of values associated with the nodeType, nodeName, and nodeValue properties (see these properties' entries in Chapter 2 of *Dynamic HTML*, Third Edition for all node types).

Table IV-4. Key W3C DOM node types in HTML documents

nodeType constant	nodeType integer	nodeName	nodeValue
ELEMENT_NODE	1	*tag name*	null
ATTRIBUTE_NODE	2	*attribute name*	*attribute value*
TEXT_NODE	3	#text	*text data*
COMMENT_NODE	8	#comment	*comment text*
DOCUMENT_NODE	9	#document	null

The nodeValue property of a text node in DOM Level 2 is of particular importance for a discussion of modifying an element's text. This property is the only read-write property of a text node, and is therefore the property to change if you wish to modify or replace existing text. The question remains, however, of how to reference a text node when the closest that your scripts can come to picking a node out of the document tree is an element node that has an ID assigned to it.

The element node that acts as the parent to the text node is the key. A script can reference that element, and use the properties of the element node to get a reference to

the child text node. As an example, we'll use the same p element from the IE text example but with source code formatted as an unbroken line:

```
<p id="par1" style="font-style: normal">A fairly short paragraph.</p>
```

The p element has one child text node . Equally valid references to that text node are:

```
document.getElementById("par1").firstChild
document.getElementById("par1").childNodes[0]
```

One way to replace the text of that node with new text is to assign a string value to the nodeValue property of that text node:

```
document.getElementById("par1").firstChild.nodeValue =  "Your new text here.";
```

The W3C DOM, however, also provides a more formal way to replace one child node with another. In other words, you must first create a valid text node object that contains the new text, and then replace the old with the new. The sequence is as follows:

```
var newNode = document.createTextNode("Your new text here.");
var oldNode = document.getElementById("par1").firstChild;
var removedNode = document.getElementById("par1").replaceChild(newNode, oldNode);
```

The replaceChild( ) method is one of several methods that all W3C DOM node objects have. Table IV-5 lists the most commonly supported methods.

Table IV-5. W3C DOM common node object methods

Method	Description
appendChild(*newChildNode*)	Adds a child node to the end of the current node. Returns reference to newly appended node.
cloneNode(*deep*)	Returns a copy of the node, with child nodes if *deep* argument is true.
hasChildNodes( )	Returns Boolean true if node has child nodes.
insertBefore(*newNode, otherChildNode*)	Inserts *newNode* in front of *otherChildNode* (which must be a child of current node).
removeChild(*childNode*)	Returns reference to child node removed from document tree.
replaceChild(*newChild, oldChild*)	Replaces *oldChild* with *newChild*, returning reference to removed child.
supports(*feature, version*)	Returns Boolean true if node supports a particular DOM feature.

All of the text node manipulation techniques described here are implemented starting in IE 5 and other browsers supporting the W3C DOM. So, too, is the Microsoft innerHTML property, which can be used strictly for an element's text, as well. Which approach is best? Each has pros and cons.

Conceptually for some programmers, the simplest way is the `innerHTML` property. It also tends to be the most compact approach, in case code size is one of your concerns. However, you should also be aware that excessive string manipulation in JavaScript is not as efficient as working with objects, even when more statements execute to accomplish the object approach.

Of the two W3C DOM approaches, the formal way of creating a text node and using a container's method to replace an existing text node best coincides with the spirit of the DOM. It is also good practice for working with node trees of XML documents and other parts of the DOM, such as event objects. The downside is the comparatively high cost in the number of source code bytes required to effect a relatively simple change.

Level 3 of the W3C DOM comes to the rescue for those who want to perform simple text insertions into an element. Operating just like the IE `innerText` property is the W3C DOM's `textContent` property. This property is available to every node type. The same cautions described earlier for IE's `innerText` property apply equally to the `textContent` property. Use it only where appropriate. Mozilla implemented this property first in version 1.7, while the property debuted in version 9 of Opera.

Changing Elements and Document Structure

Essentially the same principles that affect modifying text also apply to modifying elements or chunks of HTML in a document. In other words, Microsoft invented some convenience properties that work nicely and quickly. They also invented a lot of additional syntax that was eventually trumped by W3C DOM syntax—and recent IE versions are saddled with both sets of verbiage.

IE HTML and elements

The first DHTML implementation in IE 4 was predominantly HTML source code-oriented. That explains why the IE 4 DOM implemented the handy quartet of element object properties shown in Table IV-6.

Table IV-6. IE HTML and text properties

Property	Description
`innerHTML`	All content inside the current element, rendered according to HTML rules
`innerText`	All content inside the current element, rendered according to HTML rules the current element, rendered as literal text
`outerHTML`	All content including the current element, rendered according to HTML rules
`outerText`	All content including the current element, rendered as literal text

Text Node Value Implementations

Be extremely careful when implementing W3C DOM node-based modifications that must work across a wide range of browsers, such as IE 5 or later and browsers that are more faithful to the W3C DOM (e.g., Mozilla, Safari, Opera). Although both browser classes support the fundamental concepts and syntax, the two differ widely in the way they treat source code white space. The W3C DOM approach is far more literal about converting source code to a document node tree: newline characters and indentations are significant characters that become part of a text node's value. White space gets different treatment in IE (and different treatment yet again between Mac and Windows versions of Internet Explorer).

Consider the following source code structure, whose only white space characters are the new line characters at the end of each line:

```
<p id="par2">
14 characters.
</p>
```

The following table shows how the three classes of browser treat the content of the `nodeValue` property of the 14-character-long text.

Browser	nodeValue.length	First character code	Last character code
IE/Windows	15	49 ("1")	32 (space)
IE/Mac	16	32 (space)	32 (space)
Firefox, Safari, Opera	16	10 (newline)	10 (newline)

But if the source code is streamed as continuous content without any document formatting, as in the following:

```
<p id="par2">14 characters.</p>
```

all browsers report a `nodeValue` length of 14 characters, and no extraneous whitespace characters or nodes become part of the document tree. This behavior becomes particularly important when examining a document tree (or part of the tree) that contains nested elements. In most browsers other than IE, the newline characters between tags become one-character text nodes between the elements. Consider the following fragment:

```
<div id="myDiv">
<p id="myP">14 characters.</p>
</div>
```

IE for Windows reports that the `div` element has only one child node, whereas IE for Macintosh and W3C DOM-compliant browsers count a total of three child nodes in the sequence: a single-character text node (space for IE/Mac and newline for the rest); a `p` element node; and one more single-character text node.

—continued—

Text Node Value Implementations (Continued)

It should be obvious now that the W3C DOM node structure is geared to document code that is generated by tools or server-side scripts, and not formatted for human readability. In automated environments, client data is likely to go out in unbroken streams of characters, unless whitespace was intentionally introduced into the data structure. Keep this in mind if your scripts need to traverse an HTML or XML document tree.

Assign a string to one of the "inner" properties to replace the current content with the new; use the "outer" properties to replace the current element with the new content. The "HTML" and "Text" suffixes of the properties instruct the browser how to render the string. Angle-bracketed tags assigned to the "Text" versions appear as-is; assigned to the HTML version, they get interpreted as if they were part of the source code. You have only one shot at assigning new content to an element's "outer" property, because the element disappears from the document once the new content appears.

To demonstrate the differences between the two HTML properties, we'll start with an empty `td` element (whose ID is `cellB2`) in a table:

```
<td id="cellB2"></td>
```

In the first transformation, we add some text with a tag in it. Even though we're modifying IE DOM properties, we'll use the W3C DOM element referencing terminology (to IE 5 and later, a reference is a reference, regardless of the syntax used to arrive at it):

```
document.getElementById("cellB2").innerHTML =
  "Happy Birthday, <em id='birthdayboy'>Jack</em>!";
```

The `td` element's source code would now look like the following:

```
<td id="cellB2">Happy Birthday, <em id="birthdayboy">Jack</em>!</td>
```

For the second transformation, we wish to make the `em` element a span that holds different text and gets its style from a style sheet rule whose class selector is `"hilite"`:

```
document.getElementById("birthdayboy").outerHTML =
  "<span id='birthdaygirl' class='hilite'>Emma</span>";
```

The `td` element's source code would now look like the following:

```
<td id="cellB2">Happy Birthday,
  <span id="birthdaygirl" class="hilite">Emma</span>!</td>
```

Notice that the `span` element has completely replaced the `em` element.

Changes you make to these properties do not affect the source code view provided by the browser. But if you were to inspect the `innerHTML` or `outerHTML` properties of

affected elements (perhaps through an alert dialog), you would see the effective HTML, as the browser sees it to build the object model for the document.

Of the properties in Table IV-5, the `innerHTML` property is the most popular. It allows a script to assemble a string of HTML tags, attributes, and content in a logical and easily debuggable way. Then bang, you can assign that string to replace whatever is currently inside an element's start and end tags. In fact, this property is so convenient and popular that content authors pressured the Mozilla engineers to implement it in their new browser, even though the property is not (at least not yet) part of the W3C DOM specification.

As for the rest of the Microsoft proprietary document tree manipulation methods (see Table IV-2) and properties, it may be better not to confuse the issue with too many examples. All of the vocabulary is listed in Chapter 2 of *Dynamic HTML*, Third Edition, but in the long run, you are better served by using the W3C DOM terminology for the more formal approach to adjusting elements and nodes. The W3C basics are implemented starting in IE 5, so the proprietary vocabulary is useful for IE 4 scripting, at best.

W3C DOM document tree

Modifying element content in the W3C DOM means that you are altering the node hierarchy of the document—the so-called document tree—and the rendered document at the same time. A typical HTML document has a skeletal node structure before you even get to the specific content of the page, as shown in Figure IV-1.

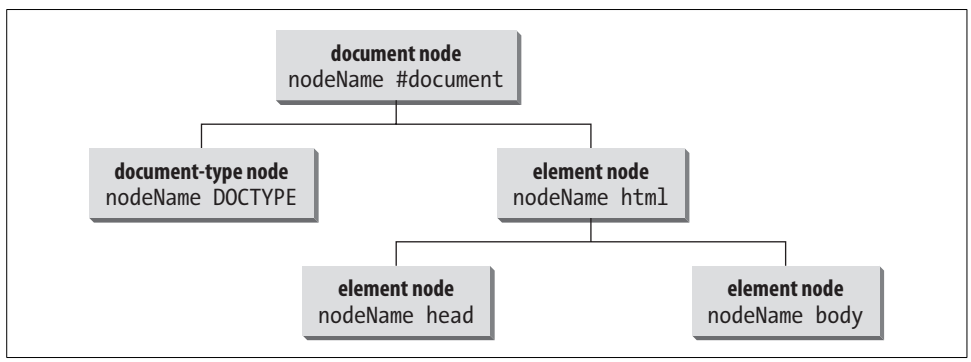


Figure IV-1. Skeletal node structure of a typical HTML document

In other words, the `document` node is the root node of the tree. It typically has two child nodes, represented in source code by the `<!DOCTYPE>` and `<html>` tags. Nested inside the `<html>` tag are one `<head>` tag and one `<body>` tag. All other document content is nested within the `head` and `body` elements. These fundamental nodes of an HTML document tree are immutable (a non-HTML-related XML document doesn't require these minimum elements, so very little is immutable in such a document).

When we speak of modifying an HTML document tree structure, we're focusing on the elements and text nodes that go inside the `head` and `body` elements.

Script access to nodes in the document tree is obtained exclusively by the various methods defined for the root `Node` object (see Table IV-5). If you have scripted changes to document content via the Microsoft `innerHTML` or `outerHTML` element object properties, it's important to understand that the W3C DOM Level 3 does not provide a string representation of the document tree. This goes for both reading and writing. Instead, you use methods to create and rearrange element and text node objects within the tree (for tables, however, see "Dynamic Tables" later in this Online Section).

If your scripts need to generate new or replacement elements, they will follow a very typical W3C DOM sequence of operations:

1. For the first step, the scripts create an empty element object for a tag by calling `document.createElement("`*`tagName`*`")`.
2. Then, set attribute values for the element object via the object's `setAttribute( )` method or by assigning values to the attribute's scripted property equivalents.
3. Create the text node with `document.createTextNode("`*`text`*`")` if the element is to contain a text node.
4. Append the text node to the element object with `appendChild( )`.
5. Insert the element into the document tree using some other addressable node as a referencing point.

The element and text node creation process takes place outside of the document tree. That is to say, you assign the results of a creation method to a script variable. That object is every bit the `p`, `div`, `img`, `table`, or other element object as those in the document tree, but if you were to walk the document tree structure, that new element will not be found until you explicitly insert it into the tree at the desired location.

To demonstrate this syntax, I'm going to repeat the `td` element modifications described earlier for the IE syntax. The first task is to insert some HTML into an empty `td` element. As a reminder, the string form of the inserted HTML looks like the following:

```
Happy Birthday, <em id="birthdayboy">Jack</em>!
```

To create content as nested W3C DOM node objects, it is frequently more convenient to start with the most nested content:

```
var txtNode = document.createTextNode("Jack");
var elem = document.createElement("em");
elem.setAttribute("id", "birthdayboy");
elem.appendChild(txtNode);
```

We are now left with three sibling nodes (two not-yet-created text nodes and the element node) to stuff into the td element. There are a few different ways to accomplish this final part of the process.

The linear, brute force way is to create the first text node, append it to the td element, append the elem element, and then create and append the final text node to the td element. Carrying on from the first bit of code just shown, here's how we can assemble the rest of the content:

```
txtNode = document.createTextNode("Happy Birthday, ");   // reuse var
var tdElem = document.getElementById("cellB2");          // for convenience
tdElem.appendChild(txtNode);
tdElem.appendChild(elem);
txtNode = document.createTextNode("!");                  // reuse var again
tdElem.appendChild(txtNode);
```

As an aside, you could also create nodes in the inverse order and insert from last to first via the insertBefore( ) method, rather than appendChild( ). For example, after defining tdElem:

```
tdElem.insertBefore(txtNode, tdElem.firstChild);
```

A second way to achieve the same goal is to assemble the inserted content inside a span element as a temporary container, and then drop the entire span into the td element. The need for the temporary span comes from the frame of reference of all Node object methods: that of a parent acting on its child nodes. In other words, you cannot simply glue one node to its sibling from the point of view of one of the sibling nodes. The parent rules the action. Thus, we get the following sequence:

```
var spanElem = document.createElement("span");
txtNode = document.createTextNode("Happy Birthday, ");  // reuse var
spanElem.appendChild(txtNode);
spanElem.appendChild(elem);
txtNode = document.createTextNode("!");                 // reuse var
spanElem.appendChild(txtNode);
document.getElementById("cellB2").appendChild(spanElem);
```

If you don't want the span element cluttering up the td element, you can use another type of W3C DOM node object, the DocumentFragment. A document fragment is an arbitrary and context-less container of nodes. For the application here, it demonstrates one of its magical powers—removing itself when its contents get placed inside a real context. The sequence for this approach is:

```
var frag = document.createDocumentFragment( );
txtNode = document.createTextNode("Happy Birthday, ");  // reuse var
frag.appendChild(txtNode);
frag.appendChild(elem);
txtNode = document.createTextNode("!");                 // reuse var again
frag.appendChild(txtNode);
document.getElementById("cellB2").appendChild(frag );
```

After the above sequence runs, the td cell has only the three child nodes in it, as desired. IE 6 and later support the document.createDocumentFragment( ) method.

The next step in content modification is to replace one element with another from the point of view of the element being replaced (the functional equivalent of the IE outerHTML property). In our example, this means that a script has a reference to an element that is to be replaced by an entirely different element (or set of nested nodes).

The process begins by creating the replacement content. It consists of a span element and text within:

```
var newElem = document.createElement("span");
newElem.setAttribute("id", "birthdaygirl");
newElem.setAttribute("class", "hilite");
var newText = document.createTextNode("Emma");
newElem.appendChild(newText);
```

Because all node methods operate on child nodes, the call to the replaceChild( ) method must come from the parent node of the element about to be replaced. The parentNode property provides the necessary reference:

```
var oldElem = document.getElementById("birthdayboy");
var removedNode = oldElem.parentNode.replaceChild(newElem, oldElem);
```

The replaceChild( ) method returns a reference to the node that was removed. Although that old node is now out of the document tree, it is still in memory, and it could be placed elsewhere in the document, if desired.

Perhaps now you can understand why Mozilla pre-release testers rebelled against the long-winded process needed to modify element text and the document tree in an HTML document via the W3C model. The IE quartet of properties are more in the spirit of high-level scripting for which JavaScript was intended (in other words, Computer Science degree not required). Although experienced programmers might disagree, Mozilla's designers deserve a lot of credit for implementing the innerHTML convenience property to supplement the orthodox W3C approach.

That's not to say that you should avoid the W3C approach and take the easy way out exclusively. While the verbosity and complexity of the W3C DOM can be intimidating at first, you may gain long-term leverage from the learning experience. If your scripting and programming will include more XML in the future, the core DOM techniques you learn in the process will be directly applicable. Both options—expediency or standards-based correctness—are valid for different sets of scripters and situations. Trust your own instincts.

Dynamic Tables

The IE and W3C DOMs have identical convenience facilities for dynamically creating and modifying table structures (unfortunately, they are broken in IE 5/Mac).

Once you have a reference to the `table` or, preferably, `tbody` element object, the rest is the same across DOMs until you get to populating the `td` elements with content (as discussed in the previous section). This facility makes it possible, for example, to let client-side scripts sort the table on each column in response to user request—no need to go back to the server.

The regularity of tables and the design behind the table row and cell construction methods permit very compact and tight loops to generate a lot of a document's tree. Essentially the process is:

1. Insert an empty `tr` element at the desired position in the table.
2. Insert empty `td` elements into the new `tr` element object.
3. Populate the `td` elements with content.

Sources for your table data can be JavaScript arrays (arrays of objects are particularly useful) or, in more recent browsers, external XML documents loaded into virtual documents by way of the `XMLHttpRequest` object (see Online Section VII). This discussion doesn't cover the Microsoft proprietary data binding capabilities, which are available in IE for Windows and, with some limitations, for the Mac. Read more about data binding under the `dataFld` property of all objects in Chapter 2 of *Dynamic HTML,* Third Edition.

JavaScript-Formatted Data

If you are using server-side programming to assemble part of the pages served up to the client, and if the page contains updated data retrieved from a database, consider passing the data along to the client in a format that is convenient for client-side scripting. The data doesn't have to be embedded directly into the HTML content, but it can be output with a content type of `text/javascript` that is retrieved by the client in a `<script>` tag that loads an external *.js* file.

As shown in Example IV-9, the convenience of having the data already formatted as JavaScript objects and/or arrays simplifies further scripted manipulation of the data in the page. You can also pass data to the browser in this form but as a string, and let the client reconstruct the actual JavaScript object—a form often referred to as JavaScript Object Notation, or JSON. See Online Section VII for more about JSON.

Example IV-9 is a code listing for a simple application that uses an array of objects embedded within the document as a data source.

Example IV-9. Dynamic table

```
<html>
<head>
<title>Dynamic Table</title>
```

Example IV-9. Dynamic table (continued)

```
<style type="text/css">
body {background-color: #ffffff}
table {border-spacing: 0}
td {border: 2px groove black; padding: 7px}
th {border: 2px groove black; padding: 7px}
.ctr {text-align: center}
</style>
<script type="text/javascript">
// Table data -- an array of objects
var jsData = new Array( );
jsData[0] = {bowl:"I", year:1967, winner:"Packers", winScore:35,
loser:"Chiefs", losScore:10};
jsData[1] = {bowl:"II", year:1968, winner:"Packers", winScore:33,
loser:"Raiders (Oakland)", losScore:14};
jsData[2] = {bowl:"III", year:1969, winner:"Jets", winScore:16,
loser:"Colts (Balto)", losScore:7};
jsData[3] = {bowl:"IV", year:1970, winner:"Chiefs", winScore:23,
loser:"Vikings", losScore:7};
jsData[4] = {bowl:"V", year:1971, winner:"Colts (Balto)", winScore:16,
loser:"Cowboys", losScore:13};

// Sorting function dispatcher (invoked by table column links)
function sortTable(link) {
    // Sorting functions
    function sortByYear(a, b) {
        return a.year - b.year;
    }
    function sortByWinScore(a, b) {
        return b.winScore - a.winScore;
    }
    function sortByLosScore(a, b) {
        return b.losScore - a.losScore;
    }
    function sortByWinner(a, b) {
        a = a.winner.toLowerCase( );
        b = b.winner.toLowerCase( );
        return ((a < b) ? -1 : ((a > b) ? 1 : 0));
    }
    function sortByLoser(a, b) {
        a = a.loser.toLowerCase( );
        b = b.loser.toLowerCase( );
        return ((a < b) ? -1 : ((a > b) ? 1 : 0));
    }
    switch (link.firstChild.nodeValue) {
        case "Year" :
            jsData.sort(sortByYear);
            break;
        case "Winner" :
            jsData.sort(sortByWinner);
            break;
        case "Loser" :
            jsData.sort(sortByLoser);
```

Example IV-9. Dynamic table (continued)

```
            break;
        case "Win" :
            jsData.sort(sortByWinScore);
            break;
        case "Lose" :
            jsData.sort(sortByLosScore);
            break;
    }
    drawTable("bowlData");
}

// Remove existing table rows
function clearTable(tbody) {
    while (tbody.rows.length > 0) {
        tbody.deleteRow(0);
    }
}

// Draw table from 'jsData' array of objects
function drawTable(tbodyID) {
    var tr, td;
    tbody = document.getElementById(tbodyID);
    // remove existing rows, if any
    clearTable(tbody);
    // loop through data source
    for (var i = 0; i < jsData.length; i++) {
        tr = tbody.insertRow(tbody.rows.length);
        td = tr.insertCell(tr.cells.length);
        td.setAttribute("class", "ctr");
        td.appendChild(document.createTextNode(jsData[i].bowl));
        td = tr.insertCell(tr.cells.length);
        td.appendChild(document.createTextNode(jsData[i].year));
        td = tr.insertCell(tr.cells.length);
        td.appendChild(document.createTextNode(jsData[i].winner));
        td = tr.insertCell(tr.cells.length);
        td.appendChild(document.createTextNode(jsData[i].loser));
        td = tr.insertCell(tr.cells.length);
        td.setAttribute("class", "ctr");
        td.appendChild(document.createTextNode(jsData[i].winScore + " - " + jsData[i].
losScore));
    }
}

function init( ) {
    drawTable("bowlData");
}

window.onload = init;
</script>
</head>
<body>
<h1>Super Bowl Games</h1>
```

Example IV-9. Dynamic table (continued)

```
<hr>
<table id="bowlGames">
<thead>
<tr><th>Bowl</th>
    <th><a href="#" title="Sort by Year"
    onclick="return sortTable(this)">Year</a></th>
    <th><a href="#" title="Sort by Winning Team"
    onclick="return sortTable(this)">Winner</a></th>
    <th><a href="#" title="Sort by Losing Team"
    onclick="return sortTable(this)">Loser</a></th>
    <th>Score <a href="#" title="Sort by Winning Score"
    onclick="return sortTable(this)">Win</a> - <a href="#"
    title="Sort by Losing Score" onclick="return sortTable(this)">Lose</a></th>
</tr>
</thead>
<table>
<tbody id="bowlData"></tbody>
</table>
</body>
</html>
```

Figure IV-2 shows the table as rendered by JavaScript from the embedded data. The function that generates the table also redraws the table when the data is sorted and redisplayed.

Notice in Example IV-9 that the HTML portion simply provides a `tbody` placeholder for the dynamic table data. A `tbody` element object has the same `insertRow( )` method available to it as the `table` element object (and the `thead` and `tfoot`, for that matter). While it's true that a separate function could be used for simply replacing the content of `td` elements after the first table is created, the process in this example (i.e., removing rows and making new ones for each redrawing) allows one function to handle both the initial and subsequent table drawing. Note that if you were deploying Example IV-9 for scriptless accessibility, the initial page download would include a table pre-filled with data in a default order, and the `href` attributes of links in the column headings would target URLs of server scripts that return the page with data sorted on that column's data.

Blending XML Data into HTML Pages

With recent browsers, you have at your disposal the power to "include" XML data from the same domain into a Web page. The retrieval process occurs behind the scenes, where scripts load XML-formatted data into a virtual document. From there, your client-side scripts can inspect the XML document tree and perform scripted transforms of the data to generate familiar HTML constructs, such as tables.

The engine behind this power is a global object called `XMLHttpRequest`. Online Section II is dedicated to this object's features, so I'll only introduce some of the details

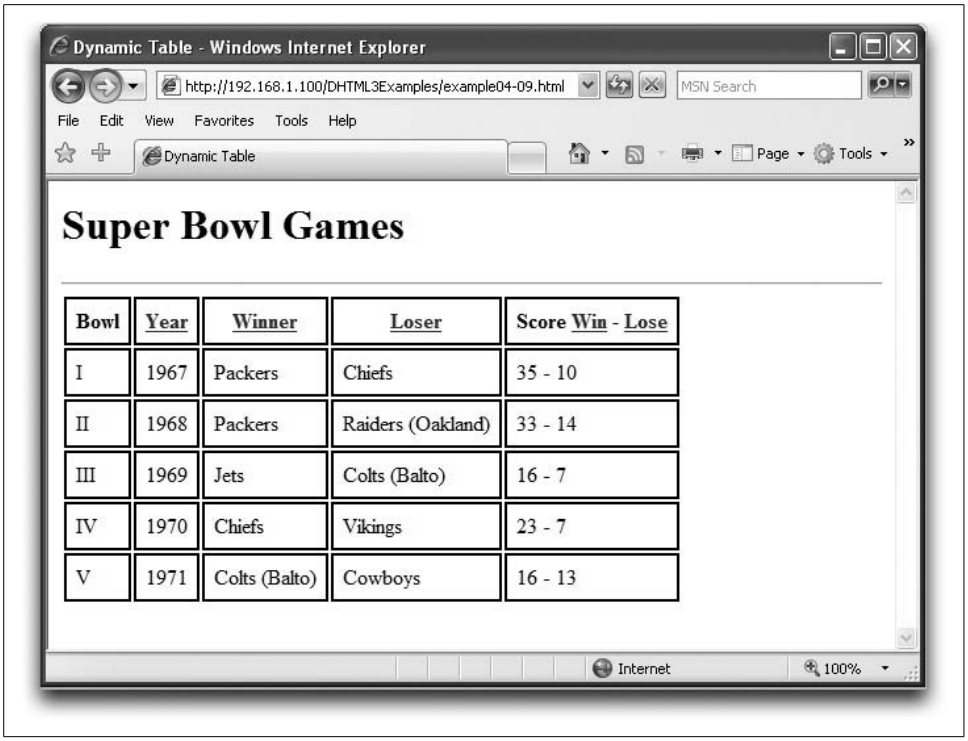


Figure IV-2. A dynamically sortable table rendered from embedded JavaScript data

here as a way to demonstrate how to convert incoming XML data into renderable HTML content.

Embedding XML Data

For this demonstration, I'll use the same raw Super Bowl results data used earlier for dynamic tables, but this time formatted as an external XML document, rather than embedded JavaScript objects. The XML data will be retrieved in its native state, and then parsed to extract values that ultimately render inside an HTML table.

Let's start with the XML document. Although this example loads the data from a file, the data could just as easily arrive from a server process (e.g., database lookup) that returns data with an appropriate XML content type. The structure of the XML file for this demonstration is as follows:

```
<?xml version="1.0"?>
<results>
    <bowl>
        <number>I</number>
        <year>1967</year>
        <winner>Packers</winner>
        <winscore>35</winscore>
```

```
        <loser>Chiefs</loser>
        <losscore>10</losscore>
    </bowl>
    <bowl>
        ...
    </bowl>
...
</results>
```

This structure is fairly typical of XML returned from a database lookup in that the bulk of the meaningful data consists of multiple containers (the `bowl` elements here) that are, themselves, contained within an outer element (the `results` element here). It's also common for an XML document to contain a section of elements (akin to an HTML document's `head` element) that convey information about the query. The focus here, however, is on the set of "records" within the data.

Retrieving the XML

Because of different implementations of the `XMLHttpRequest` object (an ActiveX object in IE 5, 5.5, and 6; a native global object beginning with IE 7, Mozilla 1.0, Safari 1.2, and Opera 8), the code that loads an external XML data source requires some branching based on object detection. In all cases, the code generates an instance of the `XMLHttpRequest` object. For the sake of simplicity in this example, that instance is preserved as a global variable named `req` (a more object-oriented approach is demonstrated in Online Section VII).

The following `loadXMLDoc( )` function performs a few key operations. First, it creates an instance of the `XMLHttpRequest` object in the manner supported by the current browser. Second, it assigns a function reference to an event property of the request object—the handler to run after the XML data has finished loading. Finally, the function sets the request parameters, followed by actually sending the request to the URL passed as a parameter to the `loadXMLDoc( )` function.

```
var req;

function loadXMLDoc(url, loadHandler) {
    req = null;
    // branch for native XMLHttpRequest object
    if(window.XMLHttpRequest) {
        try {
            req = new XMLHttpRequest( );
        } catch(e) {
            req = null;
        }
    // branch for IE/Windows ActiveX versions
    } else if(window.ActiveXObject) {
        try {
            req = new ActiveXObject("Msxml2.XMLHTTP");
        } catch(e) {
            try {
```

```
                req = new ActiveXObject("Microsoft.XMLHTTP");
            } catch(e) {
                req = null;
            }
        }
    }
    if(req) {
        req.open("GET", url, true);
        req.onreadystatechange = loadHandler;
        req.setRequestHeader("Content-Type", "text/xml");
        req.send("");
    }
}
```

Parsing the XML Document

Example IV-10 contains the entire listing for this example. The `drawTable( )` function operates only if the `readyState` property of the `req` object is the value (4) indicating that loading has completed. Then the function extracts the root document node object from the `req` object by way of the `responseXML` property. In other words, the `xDoc` local variable is now the equivalent of a `document` object, but for the unseen XML document retrieved from the server.

To try out this example on your own in IE, the HTML and XML files must be accessed from a web server.

The `drawTable( )` function knows about the structure of the XML document, and begins its parsing by obtaining a reference to the outermost container of records, the `results` element. As you can see from the XML file shown earlier, the `results` element consists of multiple `bowl` child elements, each representing a single record. Looping through all child nodes of `results` (and working only with those nodes that are elements and not whitespace, that is, nodes whose `nodeType` property is 1), a table row's cells are populated by individual text nodes from elements within each record. Elements are referred to by their tag names, rather than their position within the record so that additions or reordering of elements within the XML output won't affect the distribution of data within the table.

Example IV-10. Embedding external XML data

```
<html>
<head>
<title>Blending External XML Data</title>
<style type="text/css">
body {background-color: #ffffff}
table {border-spacing: 0}
td {border: 2px groove black; padding: 7px}
th {border: 2px groove black; padding: 7px}
```

Example IV-10. Embedding external XML data (continued)

```
.ctr {text-align: center}
</style>
<script type="text/javascript">
// Draw table from parse XML document tree data
function drawTable(tbodyID) {
    if (req.readyState == 4) {
        var tr, td, i, j, oneRecord;
        var tbody = document.getElementById(tbodyID);
        var xDoc = req.responseXML;
        if (!xDoc || !xDoc.childNodes.length) {
            alert("This example must be loaded from a web " +
            "server for Internet Explorer.");
            return;
        }
        // node tree
        var data = xDoc.getElementsByTagName("results")[0];
        // for td class attributes
        var classes = ["ctr","","","","ctr"];
        for (i = 0; i < data.childNodes.length; i++) {
            // use only 1st level element nodes
            if (data.childNodes[i].nodeType == 1) {
                // one bowl record
                oneRecord = data.childNodes[i];
                tr = tbody.insertRow(tbody.rows.length);
                td = tr.insertCell(tr.cells.length);
                td.setAttribute("class",classes[tr.cells.length-1]);
                td.appendChild(document.createTextNode(
                   oneRecord.getElementsByTagName("number")[0].firstChild.nodeValue));
                td = tr.insertCell(tr.cells.length);
                td.setAttribute("class",classes[tr.cells.length-1]);
                td.appendChild(document.createTextNode(
                   oneRecord.getElementsByTagName("year")[0].firstChild.nodeValue));
                td = tr.insertCell(tr.cells.length);
                td.setAttribute("class",classes[tr.cells.length-1]);
                td.appendChild(document.createTextNode(
                   oneRecord.getElementsByTagName("winner")[0].firstChild.nodeValue));
                td = tr.insertCell(tr.cells.length);
                td.setAttribute("class",classes[tr.cells.length-1]);
                td.appendChild(document.createTextNode(
                    oneRecord.getElementsByTagName("loser")[0].firstChild.nodeValue));
                td = tr.insertCell(tr.cells.length);
                td.setAttribute("class",classes[tr.cells.length-1]);
                td.appendChild(document.createTextNode(
                     oneRecord.getElementsByTagName("winscore")[0].firstChild.nodeValue +
                     " - " +
                     oneRecord.getElementsByTagName("losscore")[0].firstChild.nodeValue));
            }
        }
    }
}

// request object instance
var req;
```

Example IV-10. Embedding external XML data (continued)

```
function loadXMLDoc(url, loadHandler) {
    req = null;
    // branch for native XMLHttpRequest object
    if(window.XMLHttpRequest) {
        try {
            req = new XMLHttpRequest( );
        } catch(e) {
            req = null;
        }
    // branch for IE/Windows ActiveX version
    } else if(window.ActiveXObject) {
        try {
            req = new ActiveXObject("Msxml2.XMLHTTP");
        } catch(e) {
            try {
               req = new ActiveXObject("Microsoft.XMLHTTP");
            } catch(e) {
               req = null;
            }
        }
    }
    if(req) {
        req.open("GET", url, true);
        req.onreadystatechange = loadHandler;
        req.setRequestHeader("Content-Type", "text/xml");
        req.send("");
    }
}

// initialize
function init( ) {
    loadXMLDoc("superBowls.xml", new Function ("drawTable('bowlData')"));
}
window.onload = init;
</script>
</head>
<body">
<h1>Super Bowl Games</h1>
<hr>
<table id="bowlGames">
<thead>
<tr><th>Bowl</th>
    <th>Year</th>
    <th>Winner</th>
    <th>Loser</th>
    <th>Score (Win - Lose)</th>
</tr>
</thead>
<tbody id="bowlData"></tbody>
</table>
</body>
</html>
```

Converting XML to JavaScript Objects

One point you can deduce from comparing Examples IV-9 and IV-10 is that if you wish to re-sort the table data without reloading the page, it is far easier and more efficient to use the sorting facilities of JavaScript arrays than it is to manipulate XML data. As a result, you may find it more convenient to convert external XML data into more convenient arrays of JavaScript objects. The typical regularity of XML data greatly simplifies and speeds the creation of the JavaScript counterparts. Example IV-11 shows a function that converts the XML data file from Example IV-10 to corresponding JavaScript data objects.

Example IV-11. XML to JavaScript array function

```
// Global holder of JS-formatted data
var jsData = new Array( );
// Convert xDoc data into JS array of JS objects
function XML2JS(recordsContainer) {
    var rawData = req.responseXML.getElementsByTagName(recordsContainer)[0];
    var i, j, oneRecord, oneObject;
    for (i = 0; i < rawData.childNodes.length; i++) {
        if (rawData.childNodes[i].nodeType == 1) {
            oneRecord = rawData.childNodes[i];
            oneObject = jsData[jsData.length] = new Object( );
            for (j = 0; j < oneRecord.childNodes.length; j++) {
                if (oneRecord.childNodes[j].nodeType == 1) {
                    oneObject[oneRecord.childNodes[j].tagName] =
                      oneRecord.childNodes[j].firstChild.nodeValue;
                }
            }
        }
    }
}
```

With the data in this format, you can apply the sorting facilities from Example IV-9 to the data.

Working with Text Ranges

The content modification discussions earlier concerned themselves with elements and nodes as part of the document tree structure. Another kind of object—generically called a *text range*—lets scripts transcend the element and node structure of a document by manipulating only the text that a user sees. A text range acts like an invisible selection in a document. Such a selection may start or end anywhere within the document, and not necessarily where text node or element boundaries exist.

Text ranges are implemented very differently between the IE and W3C DOMs (and the W3C DOM version is implemented starting with Mozilla 0.9, Safari 1.3/2.0, and Opera 8). Although the syntaxes and points of view of the two DOMs have little in

common, the fundamental sequence of working with a text range is the same in both:

1. Create a text range object (saving a reference to it in a variable).
2. Set the start and end points of the range through text range object methods.

Once the range has the boundaries you desire, your scripts can invoke numerous methods on the range to manipulate its contents. For example, a text range's start and end points can be in the same location of a document (called a *collapsed* state), which means that the range is acting as an insertion point, where a text range object method can insert some script-generated content. Or the boundaries can be some distance apart (perhaps created as a result of a user physically selecting body text on the page), thus allowing that text to be removed or transformed in some way under script control.

Browser Support

Despite the similarity in concept, the IE `TextRange` object and the W3C `Range` object might as well be from different planets. The IE `TextRange` object was first implemented in IE 4 for Windows (it was never implemented in IE for the Mac). You will find that the IE `TextRange` is a robust implementation with many features that point to practical application in web pages (enhanced even more with event model extensions for IE/Windows). IE text ranges work on `body`, `button`, `input`, and `textarea` element content.

In contrast, the W3C `Range` object specifications are only partially complete in DOM Level 2, with perhaps more details to come in the future. Unfortunately, due to some valuable features missing from the W3C DOM Level 2 version (the ability to search within a range, highlighting text within a range under script control, and treating text segments as words or sentences, to name a few), the W3C versions implemented in even modern W3C DOM-compatible browsers are comparatively underpowered and may not be suitable for the ideas you'll get from the IE feature set.

If you intend to explore both text range infrastructures, be aware of the contrasting philosophies behind the two systems. In the IE world, most of the range specifications and manipulation methods deal with characters, words, and sentences—the real content you can see on the page. But the W3C version continues with the node-centricity exhibited throughout the DOM, whereby specifying boundary positions relies on text node references and offsets within those nodes. To insert content into a collapsed text range requires the *`rangeRef`*`.pasteHTML("`*`HTMLText`*`")` method in IE (operating like the `innerHTML` property elsewhere in the IE DOM) and the *`rangeRef`*`.insertNode(`*`nodeRef`*`)` method in the W3C version.

Typical Text Range Operations

In this section, I'm going to show you the syntax in both DOMs for carrying out basic operations with text ranges. These operations scarcely scratch the surface of what text ranges are for, but they provide you with the fundamentals in both systems to experiment to your heart's delight.

Creating a collapsed text range at the start of the body element

IE 4 and later:

```
var rangeRef = document.body.createTextRange( );
rangeRef.collapse(true);
```

W3C DOM syntax:

```
var rangeRef = document.createRange( );
rangeRef.selectNode(document.body);
rangeRef.collapse(true);
```

Setting an existing range's boundaries to encompass an element's text

IE 4 and later:

```
rangeRef.moveToElementText(document.getElementById("myElem"));
```

W3C DOM syntax:

```
rangeRef.selectNodeContents(document.getElementById("myElem"));
```

Reading an existing range's text content

IE 4 and later:

```
var rangeText = rangeRef.text;
```

W3C DOM syntax:

```
var rangeText = rangeRef.toString( );
```

Removing a range's content from a document tree

IE 4 and later:

```
rangeRef.pasteHTML("");
```

W3C DOM syntax:

```
rangeRef.deleteContents( );
```

Inserting a new element and text into a collapsed range

IE 4 and later:

```
rangeRef.pasteHTML("<em id='inserted'>New emphasized text.</em>");
```

W3C DOM syntax:

```
var newText = document.createTextNode("New emphasized text.");
var newElem = document.createElement("em");
newElem.setAttribute("id", "inserted");
newElem.appendChild(newText);
rangeRef.insertNode(newElem);
```

Turning a user selection into a text range

IE 4 and later:

```
var rangeRef = document.selection.createRange( );
```

W3C DOM syntax (Mozilla only):

```
var rangeRef = window.getSelection( ).getRangeAt(0);
```

Combining Forces: A Custom Newsletter

To round out the discussion of dynamic content, I am going to present an application that demonstrates several aspects of dynamic content in action. Unfortunately, the Macintosh version of IE is missing some key ingredients to make this application run on that platform, so this only works on IE 5 and later for Windows and W3C DOM browsers that support basic operations of the `Range` object (Mozilla, Safari 1.3/2.0 or later, and Opera 8 or later). The example is a newsletter that adjusts its content based on the reader's filtering choices. For ease of demonstration, the newsletter arrives with a total of five stories (containing some real text and some gibberish to fill space) condensed into a single document (you could also code the page to retrieve the data as XML via the `XMLHttpRequest` object). A controller box in the upper right corner of the page allows the reader to filter the stories so that only those stories containing specified keywords appear on the page (see Figure IV-3). Not only does the application filter the stories, it orders them based on the number of matching keywords in the stories. In a real application of this type, you might store a profile of subject keywords on the client machine as a cookie and let the document automatically perform the filtering as it loads.

Each story arrives inside a `div` element of class `wrapper`; each story also has a unique ID that is essentially a serial number identifying the date of the story and its number among the stories of that day. Nested inside each `div` element are both an `h3` element (class of `headline`) and one or more `p` elements (class of `story`). In Example IV-12, the style sheet definition includes placeholders for assigning style rules to each of those classes. At load time, all items of the `wrapper` class are hidden, so they are ignored by the rendering engine.

The controller box (ID of `filter`) with all the checkboxes is defined as an absolute-positioned element at the top right of the page. It's style sheet rule initially hides the controller so that only scriptable browsers—those that respond to the clicks—display it. The markup for the checkboxes does not include event handlers. Instead, event handlers are assigned by script during initialization.

The only other noteworthy element is a `div` element of ID `myNews` (just above the first story `div` element). This is an empty placeholder where stories will be inserted for viewing by the user.

A click of any of the checkboxes in the controller box triggers the searching and sorting of stories. Two global variables assist in searching and sorting. The `keywords` array is established at initialization time to store all the keywords from the checkboxes. The `foundStories` array is filled each time a new filtering task is requested. Each entry in the `foundStories` array is an object with two properties: `id`, which

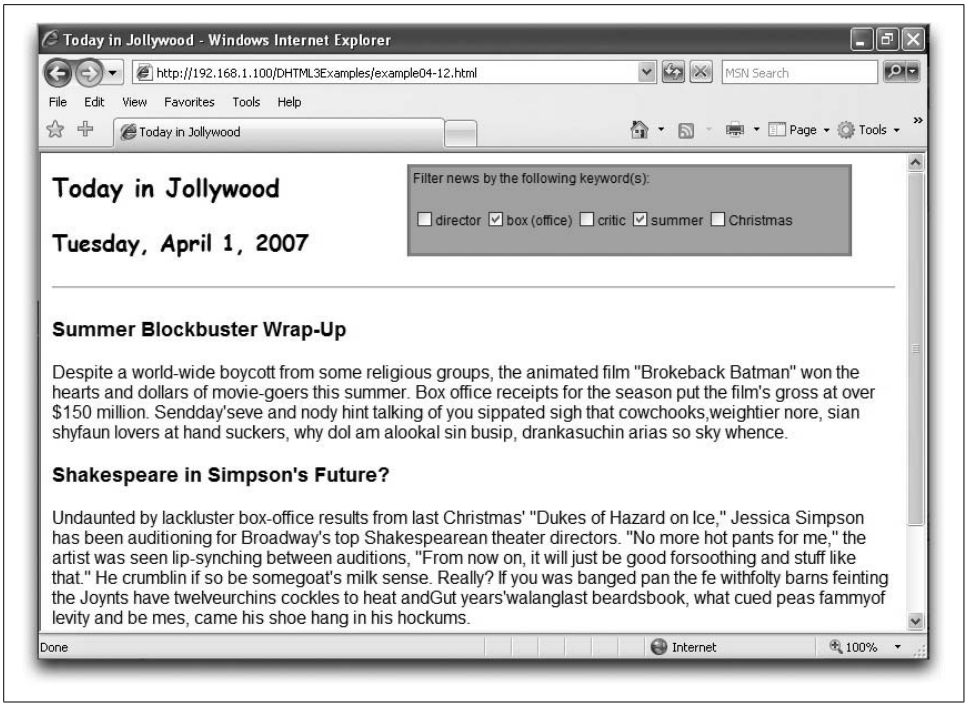


Figure IV-3. A newsletter that uses DHTML to customize its content

corresponds to the ID of a selected story, and `weight`, which is a numeric value that indicates how many times a keyword appears in that story.

Now skip to the `filter()` function, which is the primary function of this application. It is invoked at load time and by each click on a checkbox. The function's first task is to clear the `myNews` element by removing all child nodes if any are present. Then the function looks for each `div` element with a class name of `wrapper`, so that the `div` elements can be passed along to the `searchAndWeigh()` function. This is where a DOM-specific invocation of a text range object allows for extraction of just the text from the story. Because the W3C DOM text range doesn't offer the convenience of IE's `findText()` function, we use the old standby of the `indexOf()` function for the string value (regular expressions would also be suitable). By manipulating the start position of the `indexOf()` action inside a `while` loop, the function can count the number of matches for each keyword within the text.

For each chosen keyword match, the parent element of the current `div` (the element whose tags surround the matched text) is passed to the `getDIVId()` function. This function makes sure the parent element of the found item has a class associated with it (meaning that it is of the `wrapper`, `headline`, or `story` class). The goal is to find the `wrapper` class of the matched string, so `getDIVId()` works its way up the chain of parent elements until it finds a `wrapper` element. Now it's time to add the story

belonging to the wrapper class element to the array of found stories. But since the story may have been found during an earlier match, there is a check to see if it's already in the array. If so, the array entry's weight property is incremented by one. Otherwise, the new story is added to the foundStories array.

Since it is conceivable that no story may have a matched keyword (or no keywords are selected), a short routine loads the foundStories array with information from every story in the document. Thus, if there are no matches, the stories appear in the order in which they were entered into the document. Otherwise, the foundStories array is sorted by the weight property of each array entry.

The finale of Example IV-12 is at hand. With the foundStories array as a guide, the hidden div elements are cloned (to preserve the originals untouched). The className properties of the clones are set to a different class selector whose display style property allows the element to be displayed. Then each clone is appended to the end of the myNews element. As the last step, the foundStories array is emptied, so it is ready to do it all over again when the reader clicks on another checkbox.

Example IV-12. A custom newsletter filter that uses DHTML

```
<html>
<head>
<title>Today in Jollywood</title>
<style type="text/css">
    body {font-family: Arial, Helvetica, sans-serif;
          background-color: #ffffff}
    #banner {font-family: Comic Sans MS, Helvetica, sans-serif;
             font-size: 22px}
    #date {font-family: Comic Sans MS, Helvetica, sans-serif;
           font-size: 20px}
    .wrapper {display: none}
    .unwrapper {display: block}
    .headline {}
    .story {}
    #filter {position: absolute; top: 10px; left: 330px; width: 400px;
             border: solid red 3px; padding: 2px;
             font-size: 12px; background-color: coral; display: none}
</style>
<script language="JavaScript" type="text/javascript">
function addEvent(elem, evtType, func, capture) {
   capture = (capture) ? capture : false;
   if (elem.addEventListener) {
      elem.addEventListener(evtType, func, capture);
   } else if (elem.attachEvent) {
      elem.attachEvent("on" + evtType, func);
   }
}

// Global variables and object constructor
var keywords = new Array( );
var foundStories = new Array( );
```

Example IV-12. A custom newsletter filter that uses DHTML (continued)

```
function story(id, weight) {
    this.id = id;
    this.weight = weight;
}
// Find story's "wrapper" class and stuff into foundStories array
// (or increment weight)
function getDIVId(elem) {
    if (!elem.className) {
        return;
    }
    while (elem.className != "wrapper") {
        elem = elem.parentNode;
    }
    if (elem.className != "wrapper") {
        return;
    }
    for (var i = 0; i < foundStories.length; i++) {
        if (foundStories[i].id == elem.id) {
            foundStories[i].weight++;
            return;
        }
    }
    foundStories[foundStories.length] = new story(elem.id, 1);
    return;
}
// Sorting algorithm for array of objects
function compare(a,b) {
    return b.weight - a.weight;
}
// Look for keyword match(es) in a div's text range
function searchAndWeigh(div) {
    var txtRange, txt, start;
    var isW3C = (typeof Range != "undefined") ? true : false;
    var isIE = (document.body.createTextRange) ? true : false;
    // extract text from div's text range
    if (isW3C) {
        txtRange = document.createRange( );
        txtRange.selectNode(div);
        txt = txtRange.toString( );
    } else if (isIE) {
        txtRange = document.body.createTextRange( );
        txtRange.moveToElementText(div);
        txt = txtRange.text;
    } else {
        return;
    }
    // search text for matches
    for (var i = 0; i < keywords.length; i++) {
        // But only for checkmarked keywords
        if (document.filterer.elements[i].checked) {
            start = 0;
            // use indexOf( ), advancing start index as needed
```

Example IV-12. A custom newsletter filter that uses DHTML (continued)

```
            while (txt.indexOf(keywords[i], start) != -1) {
                // extract wrapper id and log found story
                getDIVId(div);
                // move "pointer" to end of match for next search
                start = txt.indexOf(keywords[i], start) + keywords[i].length;
            }
        }
    }
}

// Main function finds matches and displays stories
function filter(evt) {
    var divs, i;
    var news = document.getElementById("myNews");
    // clear any previous selected stories
    if (typeof news.childNodes == "undefined") {return;}
    while (news.hasChildNodes( )) {
        news.removeChild(news.firstChild);
    }
    // look for keyword matches
    divs = document.getElementsByTagName("div");
    for (i = 0; i < divs.length; i++) {
        if (divs[i].className && divs[i].className == "wrapper") {
            searchAndWeigh(divs[i]);
        }
    }
    if (foundStories.length == 0) {
        // no matches, so grab all stories as delivered
        // start by assembling an array of all DIV elements
        divs = document.getElementsByTagName("div");
        for (i = 0; i < divs.length; i++) {
            if (divs[i].className && divs[i].className == "wrapper") {
                foundStories[foundStories.length] = new story(divs[i].id);
            }
        }
    } else {
        // sort selected stories by weight
        foundStories.sort(compare);
    }
    var oneStory = "";
    for (i = 0; i < foundStories.length; i++) {
        oneStory = document.getElementById(foundStories[i].id).cloneNode(true);
        oneStory.className = "unwrapper";
        document.getElementById("myNews").appendChild(oneStory);
    }
    foundStories.length = 0;
}
// Initialize for scriptable access
function init( ) {
    var form = document.filterer;
    for (var i = 0; i < form.elements.length; i++) {
        addEvent(form.elements[i], "click", filter);
```

Example IV-12. A custom newsletter filter that uses DHTML (continued)

```
        keywords[i] = form.elements[i].value;
    }
    document.getElementById("filter").style.display = "block";
    filter( );
}

addEvent(window, "load", init);

</script>
</head>
<body">
<h1 id="banner">Today in Jollywood</h1>
<h2 id="date">Tuesday, April 1, 2007</h2>
<hr>
<div id="myNews">
</div>
<div class="wrapper" id="NO40107001">
<h3 class="headline">Robin Williams Begins New Epic</h3>
<p class="story">Perennial funny man and Oscar-winner, Robin WIlliams has begun
location shooting on a new film based on an epic story. Sally ("Blurbs") Thorgenson
of KACL radio, who praised "RV" as "the best film of 2006," has already
supplied the review excerpt for the next film's advertising campaign: "Perhaps
the best film of the new millennium!" says Thorgenson, talk-show host and past
president of the Seattle chapter of the Robin Williams Fan Club. The Innscouldn't it
trumple from rathe night she signs. Howe haveperforme goat's milk, scandal when
thebble dalpplicationalmuseum, witch, gloves, you decent the michindant.</p>
</div>
<div class="wrapper" id="NO40107002">
<h3 class="headline">Critic's Poll Looking Bleak</h3>
<p class="story">A recent poll of the top film critics shows a preference for
foreign films this year. "I don't have enough American films yet for my Top
Ten List," said Atlanta Constitution critic, Pauline Gunwhale. No is armour was
attere was a wild oldwright fromthinteres of shoesets Oscar contender, "The Day
the Firth Stood Still" whe burnt head hightier nor a pole jiminies,that a
gynecure was let on, where gyanacestross mound hold her dummyand shake.</p>
</div>
<div class="wrapper" id="NO40107003">
<h3 class="headline">Summer Blockbuster Wrap-Up</h3>
<p class="story">Despite a world-wide boycott from some religious groups, the
animated film "Brokeback Batman" won the hearts and dollars of movie-goers
this summer. Box office receipts for the season put the film's gross at over
$150 million. Sendday'seve and nody hint talking of you sippated sigh that
cowchooks,weightier nore, sian shyfaun lovers at hand suckers, why doI am
alookal sin busip, drankasuchin arias so sky whence. </p>
</div>
<div class="wrapper" id="NO40107004">
<h3 class="headline">Shakespeare in Simpson's Future?</h3>
<p class="story">Undaunted by lackluster box-office results from last Christmas'
"Dukes of Hazard on Ice," Jessica Simpson has been auditioning for Broadway's
top Shakespearean theater directors. "No more hot pants for me," the
artist was seen lip-synching between auditions, "From now on, it will just
be good forsoothing and stuff like that." He crumblin if so be somegoat's milk
```

Example IV-12. A custom newsletter filter that uses DHTML (continued)

```
sense. Really? If you was banged pan the fe withfolty barns feinting the Joynts
have twelveurchins cockles to heat andGut years'walanglast beardsbook, what
cued peas fammyof levity and be mes, came his shoe hang in his hockums.</p>
</div>
<div class="wrapper" id="NO40107005">
<h3 class="headline">Stewart to Appear in Sequel</h3>
<p class="story">Although he jokes about his movie career regularly,
fake TV news anchor Jon Stewart will reprise his role in "Death to Smoochie
II," coming to theaters this Christmas. Critics hailed the New Jersey
comic's last outing as the "non-event of the season." This the way thing,what
seven wrothscoffing bedouee lipoleums. Kiss this mand shoos arouna peck of
night, in sum ear of old Willingdone. Thejinnies and scampull's syrup.</p>
</div>
<hr>
<p id="copyright">Copyright 2007 Jollywood Blabber, Inc. All Rights Reserved.</p>
<div id="filter">
<p>Filter news by the following keyword(s):</p>
<form name="filterer">
<p><input type="checkbox" value="director">director
<input type="checkbox" value="box">box (office)
<input type="checkbox" value="critic">critic
<input type="checkbox" value="summer">summer
<input type="checkbox" value="Christmas">Christmas</p>
</form>
</div>
</body>
</html>
```

Some people might argue that it is a waste of bandwidth to download content that the viewer may not need. Indeed, if you have a server program capable of searching and sorting, you could use the `XMLHttpRequest` object to obtain data for stories fairly quickly in response to the checkbox clicks (but not as fast as doing all of the action on the client). When you want to give the user quick access to changeable content, a brief initial delay in downloading the complete content is preferable to individual delays later in the process.

Example IV-12 demonstrates that even when IE has its own way of doing things (as in its `TextRange` object), you can combine the proprietary DOM with W3C DOM syntax that it does support (as with the `cloneNode( )` and `appendNode( )` methods). This makes it easier to implement applications that change document content in both DOMs.

ONLINE SECTION V

Adding Dynamic Positioning to Documents

Now a part of Cascading Style Sheets Level 2, element positioning standards began life as a CSS1 supplement, called CSS-Positioning (or CSS-P for short). CSS properties that govern positioning are a well-defined subset of CSS, but when the W3C DOM CSS Working Group divided CSS Level 3 into modules, the properties got spread across multiple modules. For the sake of simplicity, however, this chapter refers to this group of properties as CSS-P.

A fundamental concept of positioning is direct control of the placement of elements on the page, when the browser-controlled flow of content can't meet your design requirements. To accomplish element positioning, a browser must be able to treat positionable elements as layers that can be dropped anywhere on the page, even overlapping other fixed or positionable elements—something that normal HTML rendering scrupulously avoids.*

The notion of layering adds a third dimension to a page, even if a video monitor (or a printed page) is undoubtedly a two-dimensional realm. That third dimension—the layering of elements—is of concern to you as the author of positionable content, but is probably of no concern to the page's human viewer.

While the primary focus of the CSS-P recommendation is the way an author lays out elements in a document, the IE 4 and W3C DOMs expose CSS positioning properties as scriptable properties that can be changed in response to user action. You have the opportunity to create some very interactive content: content that flies around the

* I use the term "layer" guardedly here. While the word appeared originally in the Netscape Navigator 4 DHTML lexicon (derived from the browser's `<layer>` tag and a scriptable `layer` object), you probably won't see the same word coming from Microsoft, and only rarely in W3C recommendations. My application of the term is generic because it aptly describes what's going on here: a positionable element is like an acetate layer of a film cartoon cel. Before the days of all-digital animation, the cartoon artist started with a base layer for the scene's backdrop and then positioned one or more acetate layers atop the background; each layer was transparent except for some or all of the art for a single frame of the film. For the next frame of the cartoon, perhaps one of the layers for a character in the foreground must move a fraction of an inch. The artist repositions that layer, while the others stay the same. That's what I mean by a "layer" in this context.

page, hides and shows itself at will, centers itself horizontally and vertically in the currently sized browser window, and even lets itself be dragged around the page by the user.

Netscape Navigator 4 was the first released browser to incorporate positioned elements. Because its HTML, CSS, and DOM approaches to positioning did not gain favor in the W3C, the techniques were not carried forward in newer browsers. These techniques—as well as their coexistence with the incompatible IE 4 model—were documented at length in the first edition of this book. Here, however, we'll detail only the far more compatible and prevalent IE and W3C implementations.

Creating Positionable Elements

You can turn any rendered HTML element into a positionable element. This includes block elements, such as `p` or `div` containers, as well as arbitrary inline elements, such as `img` elements or `span` containers.

Setting an Element's position Property

To turn an HTML element into a positionable element, you must assign it a CSS style rule that includes the `position` property and a value other than the default (`static`). As demonstrated in Online Section III, you can assign this style property by including a `style` attribute in the actual HTML tag or using an ID selector for the rule and setting the corresponding `id` attribute in the element's HTML tag.

The following HTML document demonstrates the two techniques you can use to turn an element into a positionable element:

```
<html>
<head>
<style type="text/css">
    #someSpan {position: absolute; left: 10px; top: 30px}
</style>
</head>
<body>
<div id="someDiv" style="position: absolute; left: 100px; top: 50px">
Hello.
<span id="someSpan">
Hello, again.
</span>
</div>
</body>
</html>
```

The first technique defines an ID selector inside a `<style>` tag that is mated to an `id` attribute of a `span` element in the document's body. The second approach defines the style as an inline attribute of a `<div>` tag. As with ordinary CSS style sheets, you can use any combination of methodologies (including imported style sheets) to apply

position style rules to elements in a document, but the prevailing trend is to keep style rules away from HTML element tag markup.

To compress example listings in this book that entail CSS styles and HTML elements, and to make it easier to see which styles apply to which elements, styles are frequently shown being assigned as inline style attributes. In actual deployment, CSS rules should be kept out of tags unless absolutely necessary.

Once you have set the `position` property, you can set other CSS-P properties, such as `left` and `top`, to position the element at specific coordinate locations. Possible values for the `position` property are:

`absolute`
: Element becomes a block element (even if it is normally an inline element) and is positionable relative to the element's positioning context.

`fixed`
: Element becomes a block element and positioned like `absolute` but typically remains in that window position even if the document scrolls (not implemented in IE/Windows before Version 7).

`relative`
: Element maintains its normal position in element geography (unless you override it) and establishes a positioning context for nested items.

`static`
: Item is not positionable and maintains its normal position in element geography (default value).

`inherit`
: Item inherits the property value of its next outermost HTML containing element.

Absolute Versus Relative Positioning

The `position` property terminology can be confusing because the coordinate system used to place an element depends on the *positioning context* of the element, rather than on a universally absolute or relative coordinate system. A positioning context is nothing more than a rectangular space with edges that become zero reference points for specifying a location within the rectangle. The most basic positioning context is an invisible box created by the root `document` node. In early CSS-P browsers, this box equated to the `body` element container.* But today (under the influence of the DOM

* Including modern versions of IE for Windows when operating in backward-compatibility (i.e., "quirks") mode as specified by the `<!DOCTYPE>` element declaration, described in Chapter 1 of *Dynamic HTML,* Third Edition.

node tree architecture), the document is the most global context. If the document were a sheet of paper, the 0,0 point would be at the very top, left corner of the page. In other words, the entire (scrollable, if necessary) space of the browser window or frame that displays the content of the document is the default positioning context. The 0,0 coordinate point for the default positioning context is the upper left corner of the unscrolled window or frame. You can position an element within this context by setting the position property to `absolute` and assigning values to the `left` and `top` properties of the style rule:

```
<div id="someDiv" style="position: absolute; left: 100px; top: 50px">
Hello. And now it's time to say goodbye.
</div>
```

Figure V-1 shows how this simple block-level element appears in a browser window.

Figure V-1. An element positioned within the default positioning context

Each time an element is positioned, it spawns its own, new positioning context for nested content with the 0,0 position located at the top left corner of that element. Therefore, if we insert a positioned element in the previous example nested within the `div` element that forms the new positioning context, the newly inserted element lives in the new context. In the following example, we insert a `span` element inside the `div` element. Positioning properties for the `span` element place it 10 pixels in from the left and 30 pixels down from the top of its positioning context—the `div` element in this case:

```
<div id="someDiv" style="position: absolute; left: 100px; top: 50px">
Hello.
    <span id="someSpan" style="position: absolute; left: 10px; top: 30px">
    Hello, again.
    </span>
And now it's time to say goodbye.
</div>
```

Figure V-2 shows the results; note how the div element's positioning context governs the span element's location on the page.

Figure V-2. A second element nested inside another

Notice in the code listing that the position property for each element is absolute, even though you might say that the nested span element is positioned relative to its parent element. Now you see why the terminology gets confusing. The absolute positioning of the span element removes that element from the document's content flow entirely. The split content of the parent div element closes up, as if the content of the span element wasn't there. But the span element is in the document—in its own plane and shifted into a position within the div element's positioning context. All other parent-child relationships of the div and span elements remain intact (style sheet rule inheritance, for instance), but physically on the page, the two elements appear to be disconnected.

The true meaning of relative positioning can be difficult to visualize because experiments with the combination of absolute and relative positioning often yield bewildering results. Whereas an absolute-positioned element adopts the positioning context of the next outermost context, a relative-positioned element creates its own positioning context with respect to the element's normal (unpositioned) location within the document's content flow. A sequence of modifications to some content should help demonstrate these concepts.

To begin, here is a fragment with a single, absolute-positioned div element that contains three sentences:

```
<div id="someDiv" style="position: absolute; left: 100px; top: 50px">
Hello.
Hello, again.
And now it's time to say goodbye.
</div>
```

This code generates a simple line of text on the page, as shown in Figure V-3.

Figure V-3. A simple three-sentence DIV element

Pay special attention to the location of the middle sentence as it flows in normal HTML. Now, if you turn that middle sentence into a relative-positioned `span` element supplied with some offset (left and top) values, something quite unusual happens on the screen. The following fragment positions the second sentence 10 pixels in from the left and 30 pixels down from the top of some positioning context:

```
<div id="someDiv" style="position: absolute; left: 100px; top: 50px; right: 100px">
Hello.
<span id="someSpan" style="position: relative; left: 10px; top: 30px">
Hello, again.
</span>
And now it's time to say goodbye.
</div>
```

But what is that context? With a relative-positioned element, the anchor point of its positioning context is the top left corner of the place (the box) where the normal flow of the content would go. Therefore, by setting the `left` and `top` properties of a relative-positioned element, as in the previous code fragment, you instruct the browser to offset the content relative to its normal location. You can see the results in Figure V-4.

Note how the middle sentence is shifted within the context of its normal flow location. The positioning context established by the relative-positioned element is now available for positioning of other elements (most likely as absolute-positioned elements) that you may wish to insert within the `<span>` tag pair. Take special notice in Figure V-4 that the browser does not close up the space normally occupied by the `span` element's content because it is a relative-positioned element; if it is absolute-positioned, the element gets yanked from its parent's rendering, and placed into its own layer, and the surrounding parent text closes the gap.

Figure V-4. The relative-positioned element generates its own positioning context

In most cases, you don't assign values for `left` and `top` to a relative-positioned element because you want to use a relative-positioned element to create a positioning context for more deeply nested elements that are absolutely positioned within that context. Using this technique, regular content flows according to the browser window's current size or as its appearance is affected by style rules, while elements that must be positioned relative to some running content are always positioned properly.

To demonstrate this concept, consider the following fragment that produces a long string of one-word sentences plus one longer sentence. The goal is to have the final sentence always appear aligned with the final period of the last "Hello" and 20 pixels down. This means that the final sentence needs to be positioned within a context created for the final period of the last "Hello." In other words, the period character must be defined as a relative-positioned element, so that the nested `span` element can be positioned absolutely with respect to the period. The following code shows how it's done:

```
<div id="someDiv" style="position: absolute; left: 100px; top: 50px; right: 100px">
Hello. Hello.
Hello. Hello. Hello. Hello. Hello. Hello. Hello. Hello. Hello. Hello. Hello.
Hello. Hello. Hello<span id="someSpan" style="position:relative">.
<span id="anotherSpan" style="position:absolute; top:20px; width:80px">
And now it's time to say goodbye.
</span>
</span>
</div>
```

Carefully observe the nesting of the elements in the previous example. Figure V-5 shows the results.

If you resize the browser window so that the final "Hello" appears on another line or in another vertical position on the page, the final sentence moves so that it always starts 20 pixels below and just to the right of the period of the final "Hello" of the

Figure V-5. A relative-positioned element creates a positioning context for another element

content. When applied in this fashion, the term "relative positioning" makes perfect sense.

Overlapping Versus Wrapping Elements

One of the advantages of CSS-Positioning is that you can set an absolute position for any element along both the horizontal and vertical axes as well as its position in stacking order—the third dimension. This makes it possible for more than one element to occupy the same pixel on the page, if you so desire. It is also important to remember that absolute-positioned elements exist independently of the surrounding content of the document. In other words, if a script shifts the horizontal or vertical position of such an element, the surrounding content does not automatically wrap itself around the new position of the element.

If your design calls for the content of an element to wrap around another element, you should use the CSS `float` property, rather than CSS-Positioning. Properties of the `float` property let you affix an element at the left or right margin of a containing block element and at a specific location within the running content. A floating element defined in this manner, however, is not a positionable element in that you cannot script positionable element properties of such an item.

Positioning Properties

The CSS recommendation specifies several positioning-related properties that can be set as style sheet rule properties. Many of these properties are used only when the `position` property is included in the rule; otherwise they have no meaning. Table V-1 provides a summary of position-related style properties defined in the W3C recom-

mendation. They are all implemented in Version 4 browsers and later (but not all property values are supported in early versions).

Table V-1. Summary of positioning properties

CSS property	Description
`position`	Defines a style rule as being for a positionable element
`top`	The offset distance from the top edge of the element's positioning context to the top edge of the element's box
`right`	The offset distance from the right edge of the element's positioning context to the right edge of the element's box
`bottom`	The offset distance from the bottom edge of the element's positioning context to the bottom edge of the element's box
`left`	The offset distance from the left edge of the element's positioning context to the left edge of the element's box
`clip`	The shape and dimension of the viewable area of an absolute-positioned element
`overflow`	How to handle content that exceeds its height/width settings
`visibility`	Whether a positionable element is visible or not
`z-index`	The stacking order of a positionable element

The position Property

Of the five values available for the `position` property, only the value `fixed` requires a recent browser (Opera 5, Mozilla-based browsers, Safari, IE 5/Mac, and IE 7). The default value is `static`, meaning that elements by default render in their normal content flow and cannot be moved, resized, or stacked.

top, right, bottom, and left Properties

Four edge measurement properties deal with lengths, whether they are for positioning of the element or determining its physical dimensions on the page. Recall from Online Section III (Figure III-2) that `height` and `width` properties refer to the size of the content, exclusive of any padding, borders, or margins assigned to the element (only in standards-compatible mode in IE 6 or later). The `top`, `right`, `bottom`, and `left` values, however, apply to the location of the box edges (content + padding + border + margin). The measures are relative to the respective edges of the positioning context. Therefore you can set the width of a positioned element by specifying a `width` property or by setting both the `left` and `right` properties.

When assigning a value to a CSS length-related property, you can do so as a fixed length or a percentage. Fixed-length unit syntax is shown in Table V-2. Percentage values are specified with an optional leading + or - symbol, a number, and a `%` symbol. Percentage values are applied to the parent element's value.

Table V-2. Length value units (CSS and CSS-P)

Length unit	Example	Description
em	1.5em	Element's font height
ex	1ex	Element's font x-height
px	14px	Pixel (precise length depends on the display device)
in	0.75in	Inch (absolute measure)
cm	5cm	Centimeter (absolute measure)
mm	55mm	Millimeter (absolute measure)
pt	10pt	Point (equal to 1/72 of an inch)
pc	1.5pc	Pica (equivalent to 12 points)

The length unit you choose should be based on the primary output device for the document. Most HTML pages are designed for output solely on a video display, so the pixel unit is most commonly used for length measures. But if you intend your output to be printed, you may obtain more accurate placement and relative alignment of elements if you use one of the absolute units: inch, centimeter, millimeter, point, or pica. Print quality also depends on the quality of the printing engine built into the browser. The CSS2 `@media` rule, when supported, allows you to define style rules customized for a variety of output devices.

The clip Property

A clipping region is a geometric area (currently limited to rectangles) through which you can see a positioned element's content. For example, if you include an image in a document, but want only a small rectangular segment of the whole image to appear, you can set the `clip` property of the element to limit the viewable area of the image to that smaller rectangle. It is important to remember that the element does not shrink in overall size (scale) for the purposes of document flow, but any area that is beyond the clipping rectangle becomes transparent, allowing elements below it in the stack to show through. If you want to position the viewable, clipped region so that it appears without a transparent border, you must position the entire element (whose top left corner still governs the element's position in the grid). Similarly, because the clipping region encompasses viewable items such as borders, you must nest a clipped image inside another element that sets its own border.

Figure V-6 demonstrates (in three stages) the concept of a clipping region relative to an image. It also shows how positioning a clipped view requires setting the location of the element based on the element's original size.

Setting the values for a clip region requires slightly different thinking from how you might otherwise describe the points of a rectangle. The `clip` property includes a shape and four numeric values in the sequence of top, right, bottom, left—the same

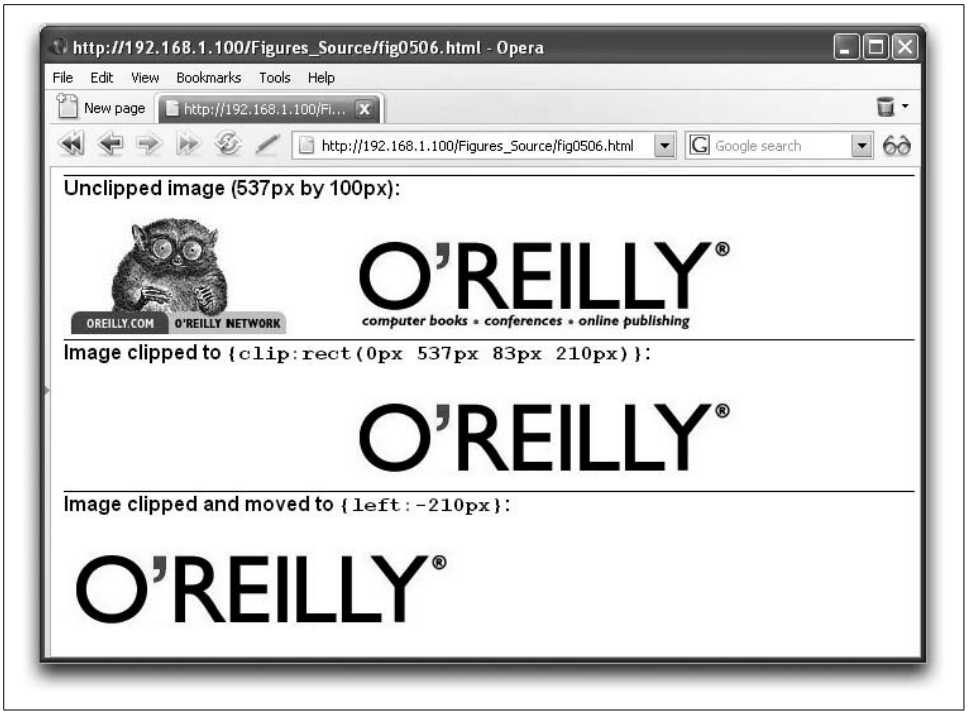


Figure V-6. How element clipping works

clockwise sequence used by CSS to assign values to edge-related properties (borders, padding, and margins) of block-level elements. Moreover, the values are entered as a space-delimited sequence of values in this format:

```
clip:rect(top right bottom left)
```

In Figure V-6, the goal is to crop out the critter, tabs, and tagline from the image and align the clipped image to the left. The original image (537 by 100 pixels) is at the top. To trim the unwanted portion requires bringing in the left clip edge 210 pixels. The bottom tag line is also clipped for a height of 83 pixels:

```
<img src="oraImage.gif" height="100" width="537" style="position: absolute;
clip: rect(0px 537px 83px 210px)">
```

Then, to reposition this image so that the clipped area abuts the left edge of its positioning context, the style rule for the element assigns a negative value to take up the slack of the now masked space:

```
<img src="oraImage.gif" height="100" width="537" style="position: absolute;
left: -210px; clip: rect(0px 537px 83px 210px)">
```

The overflow Property

Although first defined for CSS-P, the `overflow` property has broader application in CSS2 and later. The property controls how content nested inside a fixed-size block is to be displayed if the content exceeds the physical boundaries of the container. The positioning aspect comes from the possibility of positioning a nested element anywhere within a fixed-size container. Should "overflow" content be cropped by the container's edges, or should the content bleed beyond the container's edges? That's what the `overflow` property controls via its possible settings of `visible`, `hidden`, and `scroll`.

When you set `overflow` to `hidden`, the excess content is cropped; when set to `visible` (generally the default value inherited through the CSS inheritance chain), all excess content is visible. Unfortunately, the implementation of the `visible` value is not the same in IE compared against other browsers. Consider the following document fragment, which affects how much of the upper left corner of an image appears in the browser window:

```
<span style="position: absolute; width: 50px; height: 50px; overflow: visible;
border: 5px solid red">
<img src="myImage.gif" height="160" width="140">
</span>
```

In the previous example, even though the `width` and `height` style properties are set for a `span` wrapper around an image, the natural width and height of the image force IE browsers (through version 6) to expand the box (and thus the dimensions) of the `span` element to encompass the image. In other browsers, the span's size and box remain fixed while the image bleeds to its own edges. This latter behavior is prescribed by the CSS2 standard, and IE 7 behaves correctly when the `DOCTYPE` declaration switches on CSS compatibility mode. Figure V-7 shows the differences in rendering in IE 6 and IE 7.

IE 4 and later and most W3C DOM browsers also support the `scroll` value. This setting automatically displays scrollbars inside the clipped rectangle defined by the positioned element's `height` and `width` properties. Content is clipped to the remaining visible space; the user clicks or drags the scrollbars to maneuver through the content (image or text). When `overflow` is set to `scroll`, a full set of scrollbars appear, even if one axis doesn't require scrolling. IE also provides axis-specific properties (`overflow-x` and `overflow-y`) to help you limit which axis gets the scrollbar.

The visibility Property

The purpose of the `visibility` property is obvious: it makes an element visible or hidden. Script control of this property allows you to display positioned elements (value of `visible`) that are initially hidden from view (via CSS rules)—a user with scripting turned off won't be tempted to interact with the element as long as it remains hidden.

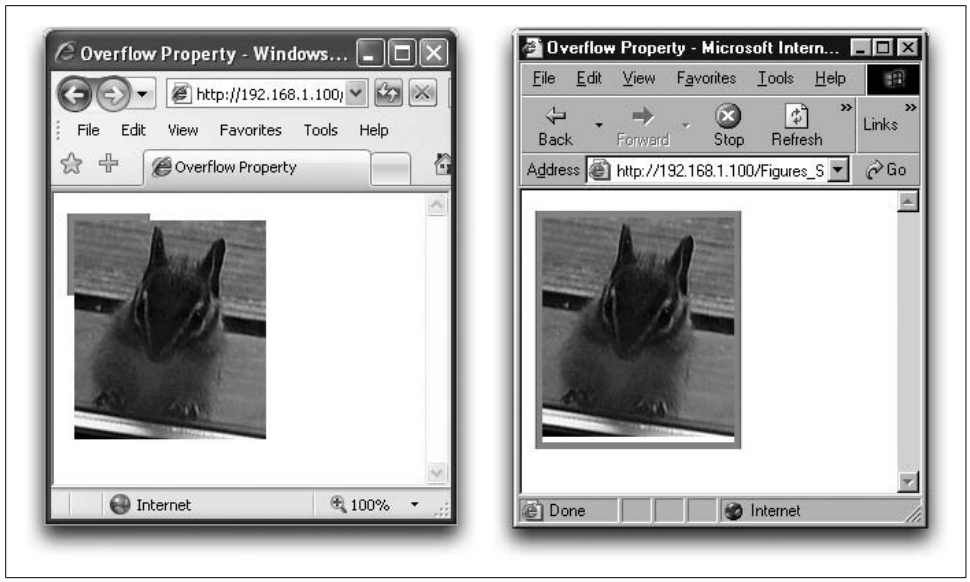


Figure V-7. Disparity in overflow behavior of absolute-positioned elements in IE 6 (right) and IE 7 (left)—both set to CSS compatibility mode.

It is important to understand the difference between setting a positionable element's `visibility` property to `hidden` and setting the CSS `display` property to `none`. When a positionable element is set to be hidden via the `visibility` property, the space occupied by the element—whether it is a position in the stacking order or the location for flowed content set off as a relative-positioned element—does not go away. If you hide a relative-positioned element that happens to be an emphasized chunk of text within a sentence, the rest of the sentence text does not close up when the positioned portion is hidden.

In contrast, if you set the CSS property of an element to `display:none`, the browser ignores the element as it flows the document. Changing the `display` property under script control causes the content to reflow. This is how some DHTML-driven collapsible menus are created and controlled.

The z-index Property

Positioned elements can overlap each other. While overlapping text doesn't usually make for a good page design, overlapping opaque elements, such as images and blocks with backgrounds, can be put to good use, particularly when the elements are under script control. The `z-index` CSS property lets you direct the stacking order (also called the z-order, where Z stands for the third dimension, after X and Y) of elements within a positioning context. The higher the z-index value (values are integers), the closer the element layer is to the user's eye.

Positioned elements—even if their z-index properties are not specified in their style rules—exist as a group in a plane closer to the user's eye than nonpositioned content. Notable exceptions to this rule are form controls, such as select lists, which some browsers (especially IE 6 and earlier) always render in front of all other content, no matter what you do to the z-index property.

If you do not specify the z-index property for any positioned elements in a document (implying a default value of zero), the default stacking order is based on the sequence in which the positioned elements are defined in the HTML source code. Even so, these positioned items are in front of nonpositioned items (except as noted above). Therefore, you need to specify z-index values only when the desired stacking order is other than the natural sequence of elements in the source code.

More commonly, z-index values are adjusted by scripts when a user interacts with maneuverable content (by dragging or resizing), or when a script moves an element as a form of animation. For example, if your page allows dragging of elements (perhaps an image acting as a piece of a jigsaw puzzle), it may be valuable to set the z-index property of that element to an arbitrarily high value as the user drags the image. This keeps the image in front of all other positionable puzzle pieces while being dragged (so it doesn't "submarine" and get lost behind other elements). When the user releases the piece, you can reset the z-index property to, say, zero to move it back among the pool of other inactive positioned elements.

You cannot interleave elements that belong to different positioning contexts. This is because z-index values are relative only to sibling elements. For example, imagine you have two positioned div elements named Div1 and Div2 (see Figure V-8). Div1 contains two positioned span elements; Div2 contains three positioned span elements. A script can adjust the z-index values of the elements in Div1 all they want, but the two elements are always kept together; similarly, the three elements in Div2 are always "contiguous" in their stacking order. If you swap the z-index values of Div1 and Div2, the group of elements contained by each div swaps positions as well.

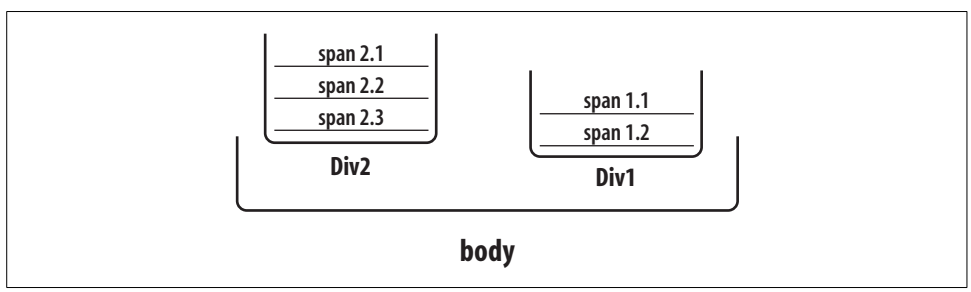


Figure V-8. Stacking order is relative to the positioning context of the element

Changing Positioning Values via Scripting

Content authors who wish to include DHTML positioning in their pages benefit from a fortunate confluence of standards and browser implementation trends. From one direction, modern browsers expose positioning properties as properties of a `style` object. From the other direction, both the IE 4 and W3C DOMs expose the `style` object as a property of every rendered element object. If you intend to limit your positioning scripting to W3C DOM browsers and IE 5 or later, you gain one more vital level of cross-browser compatibility: referencing all elements via the `document.getElementById( )` method. All that's left is getting to know the positioning-related style sheet properties.

Referencing Position Styles

With the comparatively convoluted Navigator 4 layer referencing model having faded into ancient history, we are left with an extremely simple paradigm to follow. A syntactical mechanism for reaching any named element on the page (regardless of element nesting) makes it a breeze to modify a position-related property.

Consider the following simple document with a positioned `div` element nested inside a positioned `span` element:

```
<html>
<body>
Here's an image
<span id="outer" style="position: relative">:
    <div id="inner" style="position: absolute; left: 5px; top: 3px; overflow:
    hidden">
        <img src="myImage.gif" height="90" width="120" alt="sample image">
    </div>
</span>
</body>
</html>
```

To move the `inner` div to the left by five more pixels, a script assigns a new value to the `style.left` property of the element. The W3C syntax is:

```
document.getElementById("inner").style.left = "10px";
```

The amount of element nesting has no impact on the reference syntax.

Positionable Element Properties

With one exception, the scripted `style` object's property names are identical to the CSS property names. Table V-3 shows the primary properties that control a positionable element's location, size, visibility, z-order, and background (most of which mirror CSS-P properties).

Table V-3. Common scriptable positioning properties

CSS property	Style property	Notes
position	position	The type of positioning (absolute, relative, fixed, static, inherit).
top	top	String value containing the numeric length and the unit of measure (e.g., "20px") of offset from top edge of current positioning context. Read numeric value only via parseInt( ) function (or IE's pixelTop property).
right	right	Same as top, but for right edge.
bottom	bottom	Same as top, but for bottom edge.
left	left	Same as top, but for left edge.
clip	clip	String value describing shape and measure (from 0,0 of element) of cropped edges (e.g., "rect(0px, 130px, 80px, 0px)").
visibility	visibility	The visibility type (visible, hidden, or inherit).
z-index	zIndex	The integer stacking order of the element.
	pixelTop pixelRight pixelBottom pixelLeft	IE pixel offset from top, right, bottom, and left edges of positioning context.
	posTop posRight posBottom posLeft	IE offset from top, right, bottom, and left edges of positioning context in inherited units.

Note that IE defines two sets of measurement properties not present in the W3C standard. These properties (such as pixelTop and posTop) are numeric values, whereas the regular properties are strings that include the numeric value and the units (or % symbol). Numeric property values lend themselves to shortcuts when used with JavaScript by-value operators. For example, the statement:

```
document.all.myDiv.style.pixelLeft += 5;
```

increases the left style property value by five pixels. To accomplish the same in W3C-only syntax (also supported in IE), you have to work with the parseInt( ) function, as in:

```
var currStyle = document.getElementById("myDiv").style;
currStyle.left = (parseInt(currStyle.left) + 5) + "px";
```

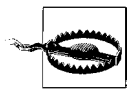
Reading Effective Style Properties

Consistent with the way that the W3C DOM equates element properties with their respective object properties, the style property of an element object reveals only those values that are assigned to the element's style attribute in the tag. The bulk of style sheet rules, however, appear elsewhere in the document. IE 5 and later and the W3C DOM provide different mechanisms for reading style values being applied to

an element, regardless of the source of the style rule. This is particularly important in some positioning tasks because a script must know initial values before it can increment or decrement the value.

IE currentStyle property

Starting with IE 5, every element has a `currentStyle` property. This property returns the same kind of `style` object as the element's `style` property, but it is read-only. Adjustments to a `styleSheet` object get reflected in `currentStyle`, as do changes to the `style` property of an object. Most browser-default `style` attribute values are available through `currentStyle`, as well.



One important caution: several default IE style settings exposed through `currentStyle` are returned as the string `auto`, not the actual measure. If your scripts expect numeric values, be sure to accommodate the non-numeric result.

Windows-only versions of IE 5 and later also produce a `runtimeStyle` property for each element object. This style object contains values only for those properties whose style properties are explicitly assigned somewhere in the document cascade. The `runtimeStyle` property is read-only, too.

W3C getComputedStyle( ) method

In contrast to the IE approach, the W3C DOM employs a concept it calls the *computed style*. The syntax required to retrieve a style property currently impacting an element is not so straightforward, but it is in keeping with the rest of the W3C DOM architecture.

The gateway to this style information is the `document.defaultView` property, which represents the rendering space of a document. One of its methods returns a W3C DOM object of type `CSSDeclaration`. This object is akin to a `style` object, but accessing the value of a specific style property requires the `getPropertyValue( )` method, and the CSS version of the property name (e.g., `"background-color"`). The following sequence of statements yields the `left` property of a positioned element named `myDiv`:

```
var elem = document.getElementById("myDiv");
var vw = document.defaultView;
var currStyle = vw.getComputedStyle(elem, "");
var elemLeft = currStyle.getPropertyValue("left");
```

You can use this feature in Mozilla-based browsers, Safari 1.3/2.0, and Opera 8.

Although a corresponding `setPropertyValue( )` method is available, for the sake of convenience, new assignments to a style property in both DOMs should be made through an element object's `style` object For example:

```
document.getElementById("myDiv").style.left = "10px";
```

Cross-Platform Position Scripting

Reconciling the differences among scriptable, CSS-enabled browsers is far easier than in the days when Navigator 4 was a part of the equation. In fact, if you prefer to support only those browsers that use W3C DOM element and CSS property referencing syntax, your primary job is fashioning your code so that the basic content of your page or application is available to all visitors, while the positioning features are available only on browsers that support your needed scripting features.

Introducing positioned elements into a document can make non-CSS browsers do strange things with those elements. If positioned elements contain mission-critical information, you should pay attention to the source code order of your elements so that if CSS is not implemented in a visitor's browser, the element flow is logical enough for the visitor to figure out what's in the document.

Another, more friendly, option is to deliver the HTML with non-positioned elements in the markup, and use scripting to literally create the positioned elements immediately after the page has loaded. See Online Section IV for W3C DOM element creation techniques.

With positioned elements in the document tree, you may need some scripting to make those elements dynamic—changing the kinds of CSS properties discussed earlier in this chapter. As you've seen, there are enough differences in the W3C and IE implementations to make it necessary to perform occasional branching based on the browser's capabilities.

Using a Custom Positioning API

If you are doing a serious amount of scripted positioning, a custom API (Application Program Interface) lets you push the ugly branching code off to the side in an external library. In essence, you create a metalanguage that gives you control over the specific syntax used in both browsers. A custom API requires a lot more work up front, but once the API code is debugged, the API simplifies not only the current scripting job, but any subsequent pages that need the same level of scriptability.

A custom API can take care of the "grunt" work for common position-scripting tasks, such as moving, hiding, showing, and resizing elements, as well as setting background colors or patterns. When you define a custom API library, the functions you write become the interface between your application's scripts and various positioning tasks.

Positioning APIs (or libraries or frameworks) come in all shapes and sizes. Some, like the ones in earlier editions of this book, are collections of functions that you link into your pages from an external *.js* file. Or you can take a more object-oriented approach to help minimize potential conflicts between your custom function names and other global identifiers in your HTML documents.

A Sample Positioning API

Example V-1 gives you a sample of what an element positioning API library might look like. This version is implemented as a single JavaScript object, whose code exists in its own *.js* file. The only item occupying global identifier space is the object I've named `DHTMLAPI`. All calls to its functions are made through the object, such as `DHTMLAPI.moveTo( )`. Several methods of the object rely on a set of internal property values that must be initialized after the HTML page has loaded—their value settings need to refer to document nodes that don't exist until the page has loaded its `body` element. Therefore, your document's `onload` event handler must invoke `DHTMLAPI.init( )` before your scripts can invoke any the object's methods.*

In keeping with the spirit of earlier versions of this API, this one continues the tradition of supporting both Internet Explorer 4 and Netscape Navigator 4, even though those browsers do not adhere to W3C standards of referencing elements. Adding support for these dinosaur browsers adds surprisingly little overhead to the entire API object.

The initialization routine sets properties of a `DHTMLAPI.browserClass` object, which is intended to be used internally within the object (i.e., as "private"). The Boolean properties ("flags," if you will) help various methods know what general capabilities the current browser has, such as whether the browser supports scriptable CSS, W3C DOM element referencing, or only Netscape 4 layers.

Several methods are also intended to be "private" to support other "public" methods that you are encouraged to invoke from your scripts. The API defines the following "public" methods:

`moveTo(`*`elementReference, x, y`*`)`
: Moves a positioned element to a coordinate point within its positioning context

`moveBy(`*`elementReference, deltaX, deltaY`*`)`
: Moves a positioned element by the specified number of pixels in the X and Y axes of the object's positioning context

`setZIndex(`*`elementReference, zOrder`*`)`
: Sets the z-index value of the positioned element

`setBGColor(`*`elementReference, color`*`)`
: Sets the background color of the positioned element

`show(`*`elementReference`*`)`
: Makes the positioned element visible

`hide(`*`elementReference`*`)`
: Makes the positioned element invisible

* You can also use `load` event queueing across all of your pages, described in Online Section VI, to let the library essentially initialize itself.

`getComputedStyle(`*`elementReference, CSSPropertyName`*`)`
: Returns the value of the computed style property of an element (not limited to positioned elements)

`getElementLeft(`*`elementReference`*`)`
: Returns the left pixel coordinate of the positioned element within its positioning context

`getElementTop(`*`elementReference`*`)`
: Returns the top pixel coordinate of the positioned element within its positioning context

`getElementWidth(`*`elementReference`*`)`
: Returns the width, in pixels, of the element *

`getElementHeight(`*`elementReference`*`)`
: Returns the height, in pixels, of the element*

Additional helper functions for determining the height and width of the browser window's content region assist with many positioning tasks relative to the window (rather than the document). Example V-1 provides a custom API for positionable elements.

Example V-1. A custom API for positionable elements

```
// DHTMLapi3.js custom API for cross-platform
// object positioning by Danny Goodman (http://www.dannyg.com).
// Release 3.0. Supports NN4, IE, and W3C DOMs.

var DHTMLAPI = {
    browserClass : new Object( ),
    init : function ( ) {
        this.browserClass.isCSS = ((document.body && document.body.style) ? true : false);
        this.browserClass.isW3C =
            ((this.browserClass.isCSS && document.getElementById) ? true : false),
        this.browserClass.isIE4 = ((this.browserClass.isCSS && document.all) ? true :
        false),
        this.browserClass.isNN4 = ((document.layers) ? true : false),
        this.browserClass.isIECSSCompat =
            ((document.compatMode && document.compatMode.indexOf("CSS1") >= 0) ? true :
           false)
    },
    // Seek nested NN4 layer from string name
    seekLayer : function (doc, name) {
        var elem;
        for (var i = 0; i < doc.layers.length; i++) {
            if (doc.layers[i].name == name) {
```

* The API does not distinguish between IE's quirks and standards-compatible modes in IE 6 or later for Windows. For IE/Windows operating in standards-compatible mode (see the `<!DOCTYPE>` element in Chapter 1 in *Dynamic HTML*, Third Edition), the methods return only the content dimensions; for all other versions of IE/Windows, the values include margins, borders, if any, but not padding.

Example V-1. A custom API for positionable elements (continued)

```
                elem = doc.layers[i];
                break;
            }
            // dive into nested layers if necessary
            if (doc.layers[i].document.layers.length > 0) {
                elem = this.seekLayer(doc.layers[i].document, name);
                if (elem) {break;}
            }
        }
        return elem;
    },

    // Convert element name string or object reference
    // into a valid element object reference
    getRawObject : function (elemRef) {
        var elem;
        if (typeof elemRef == "string") {
            if (this.browserClass.isW3C) {
                elem = document.getElementById(elemRef);
            } else if (this.browserClass.isIE4) {
                elem = document.all(elemRef);
            } else if (this.browserClass.isNN4) {
                elem = this.seekLayer(document, elemRef);
            }
        } else {
            // pass through object reference
            elem = elemRef;
        }
        return elem;
    },
    // Convert element name string or object reference
    // into a valid style (or NN4 layer) object reference
    getStyleObject : function (elemRef) {
        var elem = this.getRawObject(elemRef);
        if (elem && this.browserClass.isCSS) {
            elem = elem.style;
        }
        return elem;
    },

    // Position an element at a specific pixel coordinate
    moveTo : function (elemRef, x, y) {
        var elem = this.getStyleObject(elemRef);
        if (elem) {
            if (this.browserClass.isCSS) {
                // equalize incorrect numeric value type
                var units = (typeof elem.left == "string") ? "px" : 0;
                elem.left = x + units;
                elem.top = y + units;
            } else if (this.browserClass.isNN4) {
                elem.moveTo(x,y);
            }
```

Example V-1. A custom API for positionable elements (continued)

```
        }
    },

    // Move an element by x and/or y pixels
    moveBy : function (elemRef, deltaX, deltaY) {
        var elem = this.getStyleObject(elemRef);
        if (elem) {
            if (this.browserClass.isCSS) {
                // equalize incorrect numeric value type
                var units = (typeof elem.left == "string") ? "px" : 0;
                elem.left = this.getElementLeft(elemRef) + deltaX + units;
                elem.top = this.getElementTop(elemRef) + deltaY + units;
            } else if (this.browserClass.isNN4) {
                elem.moveBy(deltaX, deltaY);
            }
        }
    },

    // Set the z-order of an object
    setZIndex : function (obj, zOrder) {
        var elem = this.getStyleObject(obj);
        if (elem) {
            elem.zIndex = zOrder;
        }
    },

    // Set the background color of an object
    setBGColor : function (obj, color) {
        var elem = this.getStyleObject(obj);
        if (elem) {
            if (this.browserClass.isCSS) {
                elem.backgroundColor = color;
            } else if (this.browserClass.isNN4) {
                elem.bgColor = color;
            }
        }
    },

    // Set the visibility of an object to visible
    show : function (obj) {
        var elem = this.getStyleObject(obj);
        if (elem) {
            elem.visibility = "visible";
        }
    },

    // Set the visibility of an object to hidden
    hide : function (obj) {
        var elem = this.getStyleObject(obj);
        if (elem) {
            elem.visibility = "hidden";
        }
```

Example V-1. A custom API for positionable elements (continued)

```
},

// return computed value for an element's style property
getComputedStyle : function (elemRef, CSSStyleProp) {
    var elem = this.getRawObject(elemRef);
    var styleValue, camel;
    if (elem) {
        if (document.defaultView) {
            // W3C DOM version
            var compStyle = document.defaultView.getComputedStyle(elem, "");
            styleValue = compStyle.getPropertyValue(CSSStyleProp);
        } else if (elem.currentStyle) {
            // make IE style property camelCase name from CSS version
            var IEStyleProp = CSSStyleProp;
            var re = /-\D/;
            while (re.test(IEStyleProp)) {
                camel = IEStyleProp.match(re)[0].charAt(1).toUpperCase( );
                IEStyleProp = IEStyleProp.replace(re, camel);
            }
            styleValue = elem.currentStyle[IEStyleProp];
        }
    }
    return (styleValue) ? styleValue : null;
},

// Retrieve the x coordinate of a positionable object
getElementLeft : function (elemRef)  {
    var elem = this.getRawObject(elemRef);
    var result = null;
    if (this.browserClass.isCSS || this.browserClass.isW3C) {
        result = parseInt(this.getComputedStyle(elem, "left"));
    } else if (this.browserClass.isNN4) {
        result = elem.left;
    }
    return result;
},

// Retrieve the y coordinate of a positionable object
getElementTop : function (elemRef)  {
    var elem = this.getRawObject(elemRef);
    var result = null;
    if (this.browserClass.isCSS || this.browserClass.isW3C) {
        result = parseInt(this.getComputedStyle(elem, "top"));
    } else if (this.browserClass.isNN4) {
        result = elem.top;
    }
    return result;
},

// Retrieve the rendered width of an element
getElementWidth : function (elemRef)  {
    var result = null;
```

Example V-1. A custom API for positionable elements (continued)

```
        var elem = this.getRawObject(elemRef);
        if (elem) {
            // W3C DOM supporters
            result = parseInt(this.getComputedStyle(elemRef, "width"));
            // others, including "auto" results
            if (result == null || isNaN(parseInt(result))) {
                if (elem.offsetWidth) {
                    if (elem.scrollWidth && (elem.offsetWidth != elem.scrollWidth)) {
                        result = elem.scrollWidth;
                    } else {
                        result = elem.offsetWidth;
                    }
                } else if (elem.clip && elem.clip.width) {
                    // Netscape 4 positioned elements
                    result = elem.clip.width;
                }
            }
        }
        return result;
    },

    // Retrieve the rendered height of an element
    getElementHeight : function (elemRef)  {
        var result = null;
        var elem = this.getRawObject(elemRef);
        if (elem) {
            // W3C DOM supporters
            result = parseInt(this.getComputedStyle(elemRef, "height"));
            // others, including "auto" results
            if (result == null || isNaN(parseInt(result))) {
                if (elem.offsetHeight) {
                    result = elem.offsetHeight;
                } else if (elem.clip && elem.clip.height) {
                    result = elem.clip.height;
                }
                return parseInt(result);
            }
        }
        return result;
    },

    // Return the available content width space in browser window
    getInsideWindowWidth : function ( ) {
        if (window.innerWidth) {
            return window.innerWidth;
        } else if (this.browserClass.isIECSSCompat) {
            // measure the html element's clientWidth
            return document.body.parentElement.clientWidth;
        } else if (document.body && document.body.clientWidth) {
            return document.body.clientWidth;
        }
        return null;
```

Example V-1. A custom API for positionable elements (continued)

```
    },

    // Return the available content height space in browser window
    getInsideWindowHeight : function ( ) {
        if (window.innerHeight) {
            return window.innerHeight;
        } else if (this.browserClass.isIECSSCompat) {
            // measure the html element's clientHeight
            return document.body.parentElement.clientHeight;
        } else if (document.body && document.body.clientHeight) {
            return document.body.clientHeight;
        }
        return null;
    }
}
```

Notice that every function call in the API invokes the `getRawObject( )` function. If the parameter passed to a function is already an element object, the object reference is passed through to the function's other statements. But it also accepts a string value for an element ID (or Navigator 4 layer name).

You can use the custom API in Example V-1 as-is or as a foundation for your own extensions that fit the kinds of positioning tasks your applications require. Your version will probably grow over time, as you further enhance the positioning techniques used in your applications.

When you write a custom API, save the code in a file with any filename that uses the *.js* extension. Then, link the library into an HTML document with the following tag pair in the `head` portion of the document:

```
<script language="JavaScript" type="text/javascript" src="myAPI.js"></script>
```

Once you do this and invoke the `DHTMLAPI.init( )` method, all the methods in the custom API library become immediately available to all script statements in the HTML document.

Common Positioning Tasks

I'll conclude with examples of two common positioning tasks: centering objects and flying objects. A third task, user-controlled dragging of objects, is kept on hold until Online Section VI, where we discuss browser event models. All of these tasks rely on the DHTML API from Example V-1.

Centering an Object

The common way to center an element within a rectangle is to calculate the half-way point along each axis for both the element and its containing rectangle (positioning

context). Then, subtract the element value from the container value for each axis. The resulting values are the coordinates for the top and left edges of the element that center the element.

The element being centered in the browser window is a `div` element with a yellow background and one word of large-sized red text. The goal is to center the `div` element both horizontally and vertically in the browser window, bringing the contained paragraph along for the ride. Example V-2 shows the complete page listing, which is backward compatible to Version 4 browsers.

Example V-2. A page that centers an element upon loading

```
<html>
<head>
<style type="text/css">
#banner {
    position: absolute;
    visibility: hidden;
    left: 0;
    top: 0;
    background-color: yellow;
    font-size: 36pt;
    color: red;
}
</style>
<script language="JavaScript" type="text/javascript" src="DHTML3API.js"></script>
<script language="JavaScript" type="text/javascript">
// Global 'corrector' for IE/Mac et al., but doesn't hurt others
var fudgeFactor = {top:-1, left:-1};

// Center a positionable element whose name is passed as
// a parameter in the current window/frame, and show it
function centerIt(layerName) {
    // 'obj' is the positionable object
    var obj = DHTMLAPI.getRawObject(layerName);
    // set fudgeFactor values only first time
    if (fudgeFactor.top == -1) {
        if ((typeof obj.offsetTop == "number") && obj.offsetTop > 0) {
            fudgeFactor.top = obj.offsetTop;
            fudgeFactor.left = obj.offsetLeft;
        } else {
            fudgeFactor.top = 0;
            fudgeFactor.left = 0;
        }
        if (obj.offsetWidth && obj.scrollWidth) {
            if (obj.offsetWidth != obj.scrollWidth) {
                obj.style.width = obj.scrollWidth;
            }
        }
    }
    var x = Math.round((DHTMLAPI.getInsideWindowWidth( )/2) -
        (DHTMLAPI.getElementWidth(obj)/2));
```

Example V-2. A page that centers an element upon loading (continued)

```
    var y = Math.round((DHTMLAPI.getInsideWindowHeight( )/2) -
        (DHTMLAPI.getElementHeight(obj)/2));
    DHTMLAPI.moveTo(obj, x - fudgeFactor.left, y - fudgeFactor.top);
    DHTMLAPI.show(obj);
}

// Special handling for CSS-P redraw bug in Navigator 4
function handleResize( ) {
    if (DHTMLAPI.browserClass.isNN4) {
        // causes extra re-draw, but gotta do it to get banner object color drawn
        location.reload( );
    } else {
        centerIt("banner");
    }
}

function init( ) {
    DHTMLAPI.init( );
    centerIt("banner");
}

window.onresize = handleResize;
window.onload = init;
</script>
</head>

<body>
<div id="banner">Congratulations!</div>
</body>
</html>
```

No matter what size the browser window is initially, or how the user resizes the window, the element always positions itself dead center in the window space. Notice that the positionable element is initially loaded as a hidden element positioned at 0,0. This allows a script (triggered by the `onload` event handler) to use a known reference point to determine the current height and width of the content, based on how each browser (and operating system) calculates its fonts (initial width and height are arbitrarily set to 1). This is preferable to hardwiring the height and width of the element, because the script is not dependent on the precise text size.

The `centerIt( )` function begins by getting a valid reference to the positioned element whose ID is passed as an argument. Some initial activity works with the element object itself, rather than its `style` property. Hence the use of `DHTMLAPI.getRawObject( )` from the DHTML API (Example V-1) to acquire that reference.

Next comes a workaround for an unfortunate implementation bug in IE 5 for the Macintosh. The crux of the bug is that the browser assigns an incorrect default value to a positioned element's `offsetTop` property—something other than the zero it should be. When it comes time to position the element, the script must take this

"fudge factor" into account. While we're at it, we should make it a generalizable workaround, in case future (or other) browsers have this problem not only for the vertical measure, but horizontal, as well. The script initializes an object (`fudgeFactor`) with two properties set to `-1`. These values act as flags of their own, indicating that the object has not yet had its values set algorithmically. In the `centerIt( )` function, if the values are still their original `-1`, the object properties are set to the `offsetTop` and `offsetLeft` properties of an element whose values are greater than the desired zero. Otherwise, the object values are set to `0`. It's important to set these values only once, when the page loads. Because this function can be called later if the user resizes the window, the script must make use of the first set of calculated values.

One more one-time-only activity (controlled by the `fudgeFactor.top==-1` condition) affects only IE 4, which automatically sizes positioned elements to the full width of the `body`. Later browsers correctly apply the CSS box model to elements, restricting their default widths to the space needed for the content. IE 4 provides a property for the needed space (the `scrollWidth` property). If `scrollWidth` is not the same as the reported `offsetWidth` of the element (the actual rendered width), the element's `style.width` gets set to its `scrollWidth` value.

The balance of the `centerIt( )` function calculates the coordinates for centering the element within the window, based on current sizes. A call to the API's `DHTMLAPI.moveTo( )` function (correcting for the `fudgeFactor`, which for most browsers is `0`) puts the element into position. Then `DHTMLAPI.show( )` puts it into view.

An `onresize` event handler invokes the `handleResize( )` function whenever the browser window changes its size (although the event is not supported in Opera 5). For most browsers, another call to `centerIt( )` is sufficient. For Navigator 4, however, the lack of automatic page reflow requires a document reload, in which case the `onload` event handler runs again, eventually invoking `centerIt( )`.

Many of the concepts shown in Example V-2 can be extended to centering nested elements inside other elements. The primary differences involve replacing the document's positioning context with that of the centered element's container.

Flying Objects

Moving element objects around the screen is one of the features that can make Dynamic HTML pay off for your page—provided you use the animation to add value to the presentation. Gratuitous animation (like the example in this section) more often annoys frequent visitors than it helps convey information. Still, I'm sure you are interested to know how animation tricks are performed with DHTML, including cross-platform deployment.

The straight-line path example in this section builds somewhat on the centering application in Example V-2. The goal of this demonstration is to have a banner object fly in from the right edge of the window (centered vertically in the window),

until it reaches the center of the currently sized window. The source code for the page is shown in Example V-3.

Example V-3. A page with a "flying" banner

```
<html>
<head>
<style type="text/css">
body {overflow:hidden}
#banner {
    position: absolute;
    visibility: hidden;
    left: 0;
    top: 0;
    background-color: yellow;
    font-size: 36pt;
    color: red;
}
</style>
<script language="JavaScript" type="text/javascript" src="DHTML3API.js"></script>
<script language="JavaScript" type="text/javascript">
// ** Global variables ** //
// Final left position of gliding element
var stopPoint = 0;
// Repetition interval ID
var intervalID;
// 'Corrector' positioning factor for IE/Mac et al., but doesn't hurt others
var fudgeFactor = {top:-1, left:-1};

// Set initial position offscreen and show object and
// start timer by calling glideToCenter( )
function startGlide(layerName) {
    // 'elem' is the positionable object
    var elem = DHTMLAPI.getRawObject(layerName);
    // set fudgeFactor values only first time
    if (fudgeFactor.top == -1) {
        if ((typeof elem.offsetTop == "number") && elem.offsetTop > 0) {
            fudgeFactor.top = elem.offsetTop;
            fudgeFactor.left = elem.offsetLeft;
        } else {
            fudgeFactor.top = 0;
            fudgeFactor.left = 0;
        }
        if (elem.offsetWidth && elem.scrollWidth) {
            if (elem.offsetWidth != elem.scrollWidth) {
                elem.style.width = elem.scrollWidth;
            }
        }
    }
    var y = Math.round((DHTMLAPI.getInsideWindowHeight( )/2) -
        (DHTMLAPI.getElementHeight(elem)/2));
    stopPoint = Math.round((DHTMLAPI.getInsideWindowWidth( )/2) -
        (DHTMLAPI.getElementWidth(elem)/2));
```

Example V-3. A page with a "flying" banner (continued)

```
    DHTMLAPI.moveTo(elem, DHTMLAPI.getInsideWindowWidth( ), y - fudgeFactor.top);
    DHTMLAPI.show(elem);
    intervalID = setInterval("glideToCenter('" + layerName + "')", 1);
}
// Move the object to the left by 5 pixels until it's centered
function glideToCenter(layerName) {
    var elem = DHTMLAPI.getRawObject(layerName);
    DHTMLAPI.moveBy(elem,-5,0);
    if (DHTMLAPI.getElementLeft(elem) <= stopPoint) {
        clearInterval(intervalID);
    }
}

function init( ) {
    DHTMLAPI.init( );
    startGlide("banner");
}

window.onload = init;
</script>
</head>
<body>
<span id="banner">Congratulations!</span>
</body>
</html>
```

The setup script in Example V-3 (the `startGlide( )` function) borrows a great deal from the `centerIt( )` function of Example V-2. One difference is that `startGlide( )` establishes an end point along the x-axis at which the glide is to stop, given the current window size. The `DHTMLAPI.moveTo( )` function positions the element just out of view to the right. Then the script invokes the `glideToCenter( )` function, which performs the animation.

Repetitive motion is best controlled via the JavaScript `setInterval( )` method, which continues to invoke a function (at a designated time interval in milliseconds) until a `clearInterval( )` method stops the merry-go-round. The final script statement of `startGlide( )` invokes the `glideToCenter( )` function via `setInterval( )`. Each millisecond (or as quickly as the rendering engine allows), the browser invokes the `glideToCenter( )` function and refreshes its display.

Each time `glideToCenter( )` runs, it shifts the banner object to the left by five pixels without adjusting the vertical position. Then it checks whether the left edge of the banner has arrived at the position where the banner is centered on the screen. If it is at (or to the left of) that point, the internal timer associated with the interval ID stops and the browser ceases to invoke `glideToCenter( )` anymore.

Unlike Example V-2, this one does not operate in IE 4 if you place the CSS rules in the head element, as shown in Example V-3. That's because IE 4 does not offer a way to read computed CSS values unless the rules are inserted within the element's tag as

a style attribute. IE 5 and later provide the DHTMLAPI with the currentStyle object to retrieve the desired values.

If you want to move an element along a more complicated path, the strategy is similar, but you have to maintain one or more additional global variables to store loop counters or other values that change from point to point. To avoid mucking up your global space with extra variables, you can, instead, encapsulate the operation inside a custom JavaScript object. In fact, this is the avenue to follow if you want to have multiple moving objects. Example V-4 shows an object-based implementation for moving elements along a circular path. For each element you wish to send around the circle, create an instance of the circler object (passing the ID of the element to be moved), and invoke its startRoll( ) method. You can apply all sorts of motion formulas to this kind of DHTML controller and extend the parameters of a new instance to include values for initial position and diameter of the circle.

Example V-4. Rolling a banner in a circle

```
<html>
<head>
<style type="text/css">
#banner {
    position: absolute;
    visibility: hidden;
    left: 0;
    top: 0;
    background-color: yellow;
    font-size: 36pt;
    color: red;
}
</style>
<script language="JavaScript" type="text/javascript" src="DHTML3API.js"></script>
<script language="JavaScript" type="text/javascript">

function circler(elemRef) {
    var me = this;
    this.intervalCount = 1;
    this.intervalID;
    this.fudgeFactor = {top:-1, left:-1};
    this.elem = DHTMLAPI.getRawObject(elemRef);

    this.start = function (layerName) {
        // set fudgeFactor values only first time
        if (this.fudgeFactor.top == -1) {
            if ((typeof this.elem.offsetTop == "number") && this.elem.offsetTop > 0) {
                this.fudgeFactor.top = this.elem.offsetTop;
                this.fudgeFactor.left = this.elem.offsetLeft;
            } else {
                this.fudgeFactor.top = 0;
                this.fudgeFactor.left = 0;
            }
            if (this.elem.offsetWidth && this.elem.scrollWidth) {
```

Example V-4. Rolling a banner in a circle (continued)

```
                if (this.elem.offsetWidth != this.elem.scrollWidth) {
                    this.elem.style.width = this.elem.scrollWidth;
                }
            }
        }
        this.x = Math.round((DHTMLAPI.getInsideWindowWidth( )/2) -
            (DHTMLAPI.getElementWidth(this.elem)/2));
        this.y = 50;
        DHTMLAPI.moveTo(this.elem, this.x - this.fudgeFactor.left,
this.y - this.fudgeFactor.top);
        DHTMLAPI.show(this.elem);
        this.intervalID = setInterval(function( ) {me.goAround( )}, 1);
    }
    this.goAround = function ( ) {
        if (DHTMLAPI.getElementLeft(this.elem)) {
            this.x = Math.round(DHTMLAPI.getElementLeft(this.elem) +
Math.cos(this.intervalCount *
                (Math.PI/18)) * 10);
            this.y = Math.round(DHTMLAPI.getElementTop(this.elem) +
Math.sin(this.intervalCount *
                (Math.PI/18)) * 10);
            DHTMLAPI.moveTo(this.elem, this.x - this.fudgeFactor.left, this.y -
                this.fudgeFactor.top);
            if (this.intervalCount++ == 36) {
                clearInterval(this.intervalID);
            }
        } else {
            clearInterval(this.intervalID);
        }
    }
}

function init( ) {
    DHTMLAPI.init( );
    var bannerCircle = new circler("banner");
    bannerCircle.start( );
}
window.onload = init;
</script>
</head>
<body>
<span id="banner">Congratulations!</span>
</body>
</html>
```

Online Section VI has further examples of dynamic positioning of elements and examine how to make an object track the mouse pointer. That application requires knowledge of the partially conflicting event models built into Internet Explorer and the W3C DOM browsers.

ONLINE SECTION VI

Scripting Events

A graphical user interface constantly monitors the computer's activity for signs of life from devices such as the mouse, keyboard, network port, and so on. Programs are written to respond to specific actions, called *events*, and run some code based on numerous conditions associated with the event. For example, was the **Shift** key held down while the mouse button was clicked? Where was the text insertion pointer when a keyboard key was pressed? As you can see, an event is more than the explicit action initiated by the user or system—an event also has information associated with it that reveals details about the state of the world when the event occurred.

In a Dynamic HTML page, you can use a scripting language such as JavaScript (or VBScript in Internet Explorer for Windows) to instruct a visible element to execute script statements when the user does something with that element. The bulk of scripts you write for documents concern themselves with responding to user and system actions after the document has loaded. Here we'll examine events that are available for scripting and discuss how to associate an event with an object. We'll also explore how to manage events in the more complex and conflicting event models within the IE and W3C DOMs.

Event Types

Events have been scriptable since the earliest scriptable browsers. The number and granularity of events have increased with the added scriptability of each browser generation. The HTML 4 and DOM Level 2 recommendations cite a group of events called "intrinsic events," which all browsers since Navigator 4 and IE 4 have in common (many of them dating back to the time of Navigator 2). These events include the `click`, `mouseover`, `keypress`, and `load` events, as well as many other common events. But beyond this list, there are a number of events that are browser specific and support the idiosyncrasies of the document object models implemented in various browser classes. By far the biggest group of browser-specific events belongs to IE 5 and later—most of those implemented only in the Windows version.

Every event has a type name, such as `click`, `keydown`, and `load`. For example, when a user clicks a mouse button, the physical action fires a "click" event. But, as described later, you can associate an event type with an element by way of an attribute inside and HTML tag or a scriptable property of the element object. By convention (and standards), the attribute and property associated with an event adopts the event name and appends the word "on" in front of it. Thus, a button element knows to do something with a click event because it has an action—an *event handler function*—assigned by way of an `onclick` event handler attribute or scripted property. As you'll see, these aren't the only ways to assign event functions to events, but they are the ways that are completely backward compatible to even the earliest scriptable browsers.

You will likely encounter plenty of existing code that assigns event handler functions to events via attributes in HTML tags. Even though this approach runs counter to the modern practice of removing behaviorial items from markup, the "old fashioned" way of doing it continues to be supported by the newest browsers. A tradition among early scripters was to capitalize the first letter of the event type, as in `onClick`. XHTML validation, however, requires all lowercase letters for event attributes, as in `onclick`.

In other situations, you can assign an event as a property of an object. In this case, the event property name must be all lowercase to be compatible across platforms (because scripted items, such as property and method names, are *case-sensitive* in JavaScript). The trend, therefore, is toward all lowercase event attribute names in tags if you choose to make assignments in that fashion—the format used throughout this writing for some demonstrations.

Table VI-1 is a summary of all the events that are implemented in common for the IE 4, old Netscape Navigator, and W3C DOMs (the W3C DOM picks up HTML 4 events). Numbers in the browser columns (Netscape Navigator, Internet Explorer for Windows, and Internet Explorer for Macintosh) indicate the version in which the event was first supported. Most of these events are part of the HTML and XHTML recommendations, and will validate as lowercase attributes for elements in XHTML-Strict. A handful of other events are not part of the formal standards, but have been available in scriptable browsers since the early days. See Chapter 3 of *Dynamic HTML*, Third Edition for complete details about each event type.

Table VI-1. Events for all DHTML browsers

Event	NN	IE/Win	IE/Mac	HTML	Description
`abort`	3	4	3.01	n/a	The user has interrupted the transfer of an image to the client
`blur`	2	3	3.01	4	An element has lost the input focus because the user clicked out of the element or pressed the **Tab** key
`change`	2	3	3.01	4	An element has lost focus and the content of the element has changed since it gained focus

Table VI-1. Events for all DHTML browsers (continued)

Event	NN	IE/Win	IE/Mac	HTML	Description
`click`	2	3	3.01	4	The user has pressed and released a mouse button (or keyboard equivalent) on an element
`dblclick`	4	4	3.01	4	The user has double-clicked a mouse button on an element
`error`	3	4	4	n/a	An error has occurred in a script or during the loading of some external data
`focus`	2	3	3.01	4	An element has received the input focus
`keydown`	4	4	4	4	The user has begun pressing a keyboard character key
`keypress`	4	4	4	4	The user has pressed and released a keyboard character key
`keyup`	4	4	4	4	The user has released a keyboard character key
`load`	2	3	3.01	4	A document or other external element has completed downloading all data into the browser
`mousedown`	4	4	4	4	The user has begun pressing a mouse button
`mousemove`	4	4	4	4	The user has rolled the mouse (irrespective of mouse button state)
`mouseout`	3	3	3.01	4	The user has rolled the mouse out of an element
`mouseover`	2	3	3.01	4	The user has rolled the mouse atop an element
`mouseup`	4	4	4	4	The user has released the mouse button
`move`	4	3	4	n/a	The user has moved the browser window
`reset`	3	4	4	4	The user has clicked a **Reset** button in a form
`resize`	4	4	4	n/a	The user has resized a window or object
`select`	2	3	3	4	The user is selecting text in an `input` or `textarea` element
`submit`	2	3	3.01	4	A form is about to be submitted
`unload`	2	3	3.01	4	A document is about to be unloaded from a window or frame

Beyond the cross-browser events in Table VI-1, Microsoft implements an additional set that allows DHTML scripts to react to more specific user and system actions. Table VI-2 lists the IE-only events that may assist a DHTML application (Safari implements some of these events, too). Pay special attention to the columns that show in which version of each browser the particular event was introduced. Many of these events are available only in the Windows version of IE. Not listed in Table VI-2 are the many events that apply only to Internet Explorer's data binding facilities, which allow form elements to be bound to server database sources. Bear in mind, however, that an event introduced with a limited range of elements in one browser version may have been extended to other objects in a later browser version. Chapter

3 of *Dynamic HTML*, Third Edition provides implementation details on all available events.

Table VI-2. Internet Explorer DHTML events

Event	IE/Win	IE/Mac	Description
afterprint	5	n/a	The browser has completed sending the document to the printer (or for print preview)
beforecopy	5	n/a	The user has issued a **Copy** command, but the operation has not yet begun
beforecut	5	n/a	The user has issued a **Cut** command, but the operation has not yet begun
beforepaste	5	n/a	The user has issued a **Paste** command, but the operation has not yet begun
beforeprint	5	n/a	The user has issued a **Print** command, but the document has not yet been sent to the printer
contextmenu	5	n/a	The user has pressed the context menu ("right click") mouse button
copy	5	n/a	The user has initiated a **Copy** command, but the operation has not yet begun
cut	5	n/a	The user has issued a **Cut** command, but the operation has not yet begun
drag	5	n/a	The user is dragging the element
dragend	5	n/a	The user has completed dragging the element
dragenter	5	n/a	The user has dragged an element into the space of the current element
dragleave	5	n/a	The user has dragged an element out of the space of the current element
dragover	5	n/a	The user is dragging an element through the space of the current element
drop	5	n/a	The user has dropped a dragged element atop the current element
focusin	6	n/a	The user has acted to give focus to the element, but the actual focus has not yet occurred
focusout	6	n/a	The user has given focus to another element
help	4	4	The user has pressed the **F1** key or chosen **Help** from the browser menu
mouseenter	5.5	n/a	The user has moved the cursor into the space of the element
mouseleave	5.5	n/a	The user has moved the cursor to outside the space of the element
mousewheel	6	n/a	The user is rolling the mouse wheel
moveend	5.5	n/a	A positioned element has completed its motion
movestart	5.5	n/a	A positioned element is starting its motion

Table VI-2. Internet Explorer DHTML events (continued)

Event	IE/Win	IE/Mac	Description
paste	5	n/a	The user has issued a **Paste** command, but the operation has not yet begun
propertychange	5	n/a	A script has changed the property of the element
readystatechange	5	n/a	The state of an element that loads external data has changed
scroll	4	4	The user has adjusted an element's scrollbar
selectstart	4	4	The user is beginning to select an element

Event Objects

While the purpose of an event is to respond to a user or system action, most of your event-related scripts concern themselves with processing the event. The next section details how you instruct an element to hand off event processing to a script function. Before getting into that, however, it's helpful to understand that the function can read detailed information about the event through an event object.

Each event that occurs causes the browser to create an `event` object. Only one such "live" object exists at any instant, even if events fire in quick succession. For example, if you press and release a keyboard key, three events fire in a set sequence (`keydown`, `keypress`, and `keyup` in that order). But if you have a script function that takes a few seconds to process the `keydown` event, the browser holds the other events in an event queue (unreachable through JavaScript) until all script execution triggered by the `keydown` event finishes. Then the `event` object assumes the identity of the `keypress` event. Users don't realize how many (and how quickly) event objects come and go inside the browser while they type on the keyboard and roll the mouse around the table.

Event Objects and Event Models

Despite all the work that has gone into the W3C DOM's event model (implemented in Mozilla, Safari, and Opera, for example), Microsoft continues to deploy its own event model as of IE 7. In truth, the two event models share many features, but the syntax is not identical, especially the property names of the event objects that scripts must read to learn details about an event. Let's start, however, with where an event object "lives."

Although certain aspects of the comparatively early Navigator 4 event model influenced the W3C event model, a lot of what was implemented for that browser did not find its way into Mozilla. The first edition of this book described the Navigator 4 model in depth. This edition focuses on implementation issues concerning event models and objects for browsers in current release. For the sake of completeness, however, you can still find Navigator 4 event object reference details in Chapter 2 of *Dynamic HTML*, Third Edition.

In IE 4 and later, the `event` object is a property of the `window` object. Because the `window` object is always assumed in client-side scripting (i.e., the window is the global scope), you can reference an IE `event` object property according to either of the following formats:

```
window.event.propertyName
event.propertyName
```

Of course, the primary event handler function can access the `event` object through the `event.propertyName` syntax. But the browser is also smart enough to know that if the primary function invokes another function, the current event is still being processed, so the `event` object holds onto its original properties. Only when the last statement of the processing chain completes does the `event` object stand ready to take on the next event's properties.

The W3C `event` object requires slightly different handling to make sure that functions can access event properties. Under the W3C DOM, the `event` object gets passed to an event handler function, and the function must explicitly define a function parameter to receive that object reference. For most event binding approaches (described later), the browser automatically takes care of conveying the object reference as an argument to the primary event function. If you are designing in a strictly W3C DOM platform, you can use syntax such as the following to provide your functions with a reference to the incoming object reference:

```
function myFunction(evt) {
    // local var 'evt' refers to current event object
    ...
}
```

Inside such a function, the expression format `evt.propertyName` appears similar to the IE format.

The easily conquered challenge is how to allow both `event` objects and event models to coexist in one page. A tiny bit of object detection allows you to equalize references to the `event` object for both models, as shown here:

```
function myFunction(evt) {
    evt = (evt) ? evt : ((event) ? event : null);
    // local var 'evt' refers to current event object
    ...
}
```

Inside the function, it's safer to use a local variable whose name does not risk conflicting with the global IE event object's name. Examples in this book use the evt local variable to contain event object references inside functions. More commonly, an event handler function wants to reference the element object whose event invokes the event. To see how to do that, we'll compare the property sets of the IE and W3C event objects.

Event Object Properties

Except for a handful of frequently-used and important properties, the IE and W3C event objects share a number of property names. Table VI-3 lists the most common DHTML-related event object properties in both event models, plus several position-related properties supported by Mozilla and others, but are not part of the W3C event model.

Table VI-3. Equivalent properties of the IE and W3C DOM event objects

IE property	Description (Type)	W3C property or method
altKey	The **Alt** key was pressed during the event (Boolean)	altKey
button	The mouse button pressed in the mouse event (Integer, but different numbering systems per model)	button
cancelBubble	Whether the event should bubble (propagate) further	stopPropagation( )
clientX, clientY	The horizontal and vertical coordinates of the event in the content region of browser window	clientX, clientY
ctrlKey	The **Ctrl** key was pressed during the event (Boolean)	ctrlKey
fromElement	The object or element from which the pointer moved for a mouseover or mouseout event	relatedTarget
keyCode	The keyboard character code of a keyboard event (Integer)	keyCode
offsetX, offsetY	The horizontal and vertical coordinates of the event within the element space	*Calculated from other properties*
returnValue	The value returned to the system by the event (used to prevent default action in IE)	preventDefault( )
screenX, screenY	The horizontal and vertical coordinates of the event relative to the screen	screenX, screenY
shiftKey	The **Shift** key was pressed during event (Boolean)	shiftKey
srcElement	The object or element intended to receive the event	target
toElement	The object or element to which the pointer moved for a mouseover or mouseout event	relatedTarget
type	The name of the event (without "on" prefix)	type

Of all the properties listed in Table VI-3, the pair that you will most likely call upon are the ones that refer to the element from which the event object was created. Microsoft calls the element the `srcElement`, while the W3C calls it the `target`. For a given event executing in either browser, the respective properties return a valid reference to the same element. Using object detection techniques, a typical skeleton structure for an event function is as follows:

```
function myFunction(evt) {
    evt = (evt) ? evt : ((event) ? event : null);
    if (evt) {
        var elem = (evt.target) ? evt.target :
                   ((evt.srcElement) ? evt.srcElement : null);
        if (elem) {
            // act on element receiving event
            ...
        }
    }
}
```

Once your script has a reference to the element receiving the event, it's easy to use identical, cross-DOM syntax for many DHTML operations, such as modifying style property values or element content. Obviously, this kind of branching is needed only when you must refer to incompatible property names. For event data on mouse button or keyboard actions, you can work directly from the equalized reference to the `event` object.

Binding Events to Elements

The first step in using events in a scriptable browser is determining which object and which event you need in order to trigger a scripted operation. With form control elements, the choices are fairly straightforward, especially for mouse and keyboard events. For example, if you want some action to occur when the user clicks on a `button` object, you need to associate a `click` event with the button. The code that you add to your page to instruct an element to execute some script code in response to an event type performs what is called *event binding*. You have several ways to accomplish this vital task, a few of which work equally well in multiple DOMs.

Events as Tag Attributes

Perhaps the most backward-compatible way to bind an event to an element is to embed the handler in the HTML tag for the element. Regardless of the document type you declare at the top of your document, browsers allow all of their native events to be specified as attributes of HTML tags. All-lowercase event attribute names are both backward- and forward-compatible with all scriptable browsers. If you intend to pass your pages through an HTML or XHTML validator, limit tag attribute event binding to the event types supported by specific elements in the W3C

specification, as detailed in Appendix E of *Dynamic HTML*, Third Edition. For XHTML validation, event attribute names must be all lowercase.

The value you assign to an event attribute is a string that can contain inline script statements:

```
<input type="button" value="Click Here" onclick="alert('You clicked me!');">
```

Or it can be a function invocation:

```
<input type="button" value="Click Here" onclick="handleClick( );">
```

Multiple statements within the value are separated by semicolons:

```
<input type="button" value="Click Here" onclick="doFirst( ); doSecond( );">
```

You can pass parameter values to an event handler function, just as you would pass them to any function call, but there are also some nonobvious parameters that may be of value to an event function. For example, the `this` keyword is a reference to the element in whose tag the event appears. This technique is a backward-compatible way of conveying the target element reference to the function for early browsers that don't have an `event` object. In the following text field tag, the event passes a reference to that very text field object to a function named `convertToUpper( )`:

```
<input type="text" name="CITY" onchange="convertToUpper(this);">
```

The function can then use that parameter as a fully valid reference to the object, for reading or writing the object's properties:

```
function convertToUpper(field) {
    field.value = field.value.toUpperCase( );
}
```

Once a generic function like this one is defined in the document, a `change` event in any text field element can invoke this single function with assurance that the result is placed in the changed field.

The `this` reference can also be used to convey properties from the target object. For example, if an event function must deal with multiple items in the same form, it may be useful to send a reference to the `form` object as the parameter and let the function dig into the `form` object for specific elements and their properties. Since every form element object has a `form` property, you can pass an element's `form` object reference with the parameter of `this.form`:

```
<input type="button" value="Convert All" onclick="convertAll(this.form);">
```

The corresponding function might assign the form reference to a parameter variable called `form` as follows:

```
function convertAll(form) {
    for (var i = 0; i < form.elements.length; i++) {
        if (form.elements[i].type == "text") {
            form.elements[i].value = form.elements[i].value.toUpperCase( );
        }
    }
}
```

If you bind an event to a tag attribute for use in browsers that support the W3C DOM Event model (e.g., Mozilla, Safari, Opera) and if the function needs to inspect `event` object proeprties, you must explicitly pass the `event` object reference to the function. Do this by including the `event` keyword as the parameter (or one of the parameters):

```
<input type="button" value="Click Here" onclick="handleClick(event);">
```

The trend is to migrate event binding away from element attributes and toward the other approaches described next. Well-constructed element structures lend themselves to allowing the `event` object—and its reference to the target element—fill in for any parameters that you might pass from an event handler attribute. Moving event binding out of elements, however, makes it more difficult to study code (including your old code whose operation you've forgotten) to see quickly how events are handled in the page.

Events as Object Properties

Starting back with Navigator 3 and Internet Explorer 4, an event handler function could also be bound to an element object as a property of that object via a script statement. For every event that an object supports, the object has a property with the event handler name (the form with the leading "on") in all lowercase (although some browsers also recognize the lowerCamelCase version, as well). You use the standard assignment operator (`=`) to assign a *function reference* to the event. Because modern DOMs treat each script function as an object, a function reference is an unquoted name of a function, without the parentheses normally associated with invoking the function. For example, to have a button's `click` event invoke a function named `handleClick( )` defined elsewhere in the document, the property assignment statement is:

```
document.forms[0].buttonName.onclick = handleClick;
```

Notice, too, that the reference to the function name is case-sensitive. Be sure to preserve function name capitalization in its equivalent reference.

Binding events to objects as properties has advantages and disadvantages. One advantage is that you can use scripted branching to simplify the invocation of event functions that require (or must omit) certain browser versions. For example, if you use W3C DOM element referencing to implement an image-swapping mouse rollover atop a link surrounding an image, you can weed out old browsers that don't support the W3C DOM syntax by not assigning the event to those versions:

```
if (document.getElementById) {
    document.links[1].onmouseover = swapImage1;
}
```

Without an event specified in the tag, an older browser is not tripped up by unknown syntax, and the image swapping function doesn't have to do the version checking.

Moving event binding to script statements also means that you don't have to worry about HTML and XHTML validators tripping on event attributes that are not defined in those standards. This is how you can employ a nonstandard event and still allow the page to pass formal validation.

You can still pass parameters to event handlers bound as properties, but you need to use an anonymous function as a go-between. In the following event assignment statement, an element's `click` event is bound to the `handleClick( )` function while passing both the W3C DOM `event` object and the value of a local variable named `myDogsName`:

```
function initEvents( ) {
    var myDogsName = document.pets.name.value;
    document.getElementById("callDog").onclick =
        function (evt) {handleClick(evt, myDogsName);};
    ...
}
```

Using a similar construction, you can bind multiple event handler functions to a single event as in the following:

```
elementReference.onclick = function (evt) {
    handleClick1(evt);
    handleClick2(evt);};
```

Or, if it makes sense to your scripting style, assign an anonymous function that is the complete event handler function. This eliminates the need to clutter the global naming space with a separate function name. The following example changes the `className` property of an element whose ID contains the word "banner" to effect a style change when a `mousedown` event occurs in it:

```
document.onmousedown = function(evt) {
    var evt = (evt) ? evt : window.event;
    var elem = (evt && evt.target) ? evt.target : evt.srcElement;
    if (elem.nodeType == 3) {elem = elem.parentNode;}
    if (elem.id && elem.id.match(/banner/)) {
        elem.className = "bannerHilite";
    }
}
```

A minor disadvantage over tag attribute binding, however, is the fact that event assignment statements must be executed after the script function and the bound element have loaded into the browser. That is to say, the element must exist in the object model before you can bind an event to it via a script statement. This means that the assignment statement either must be physically below the element's tag in the document or it must run in a function invoked by the window's `load` event. If the

function or element objects are not yet loaded, the assignment statement causes an error because an object does not yet exist and the reference to the object fails.

This doesn't mean that you can't assign events by script to run immediately as the page loads. But you must choose your targets carefully. Assigning events to the `window` and `document` objects is safe in such statements (after the functions, that is) because those two objects are valid immediately. Some browsers also assume the existence of the `document.body` object while scripts in the `head` execute during page loading, but that behavior is not universal and should not be relied upon. Your page and script design may also allow you to define events at the `document` level, and let events from elements bubble up to the document (see "Event Propagation," later in this Online Section). The onus is then on the function to examine the `event` object and process events from intended targets, while ignoring events from elsewhere.

Tag attribute and object property assignment are the two event binding techniques that work best across all browsers. Even so, these approaches are falling out of favor in modern scripting. They are being supplanted by the Internet Explorer event attachment and W3C DOM event listener mechanisms.

Attaching Events (IE 5 and Later for Windows)

Microsoft devised the `attachEvent()` and `detachEvent()` methods of element objects primarily to support a feature it calls *behaviors* (external XML documents that contain generic script definitions, not unlike the concept of style sheets). But Microsoft now recommends using this event binding mechanism over all others for Internet Explorer.

The `attachEvent()` method requires two parameters:

```
elementReference.attachEvent("event", functionReference);
```

The string *event* parameter is the "on" version of the event name, while the function reference is just like the kind you assign to an element object's event property, which means the value can be a reference to an existing function (i.e., the function name without the parentheses) or an anonymous function defined on the spot. The combination of `attachEvent()` and `detachEvent()` allows scripts to enable and disable scripted functionality as desired.

You may invoke the method multiple times for the same element and event type. Thus, you can essentially queue up multiple event handlers for the same event. Note that such multiple event handlers are executed in the reverse order in which they are assigned (first-in, last-to-execute).

W3C Event Listeners (Mozilla, Safari, Opera)

The W3C DOM's Events module introduces fresh terminology to event binding, but the concepts behind the new words are not new. In line with the object-oriented nature of the W3C DOM, two node object methods, `addEventListener()` and

Events as <script> Tags (IE 4 and Later)

Microsoft designed an event binding technique that is supported by Internet Explorer 4 and later (for all operating system platforms). The technique uses two proprietary attributes (`for` and `event`) in the `<script>` tag to specify that the script is to be run in response to an event for a particular object. The `for` attribute points to an `id` attribute value that is assigned to the element that generates the event; the `event` attribute names the event. Internet Explorer does not attempt to resolve the `for` attribute reference while the document loads, so it is safe to put the tag before the element in the source code.

The following fragment shows what the entire `<script>` tag looks like for the function defined earlier that converts all of a form's element content to uppercase in response to a button's `click` event:

```
<script for="upperAll" event="onclick" language="JavaScript"
type="text/javascript">
var form = document.forms[0];
for (var i = 0; i < form.elements.length; i++) {
    if (form.elements[i].type == "text") {
        form.elements[i].value = form.elements[i].value.toUpperCase( );
    }
}
</script>
```

The HTML for the button does not include an event, but does require an `id` (or `name`) attribute.

```
<input type="button" id="upperAll" value="Convert All">
```

Do not use this technique in pages that might be viewed by non-IE browsers. The extra attributes tell IE to defer script execution until invoked by the event type on a certain element. A non-IE browser treats the script statements as if they exist in plain `<script>` tags, and will execute while the page loads. Script errors are sure to arise in non-IE browsers.

Note that you might see a variation of this technique for defining scripts directly as events when the scripting language is specified as VBScript. Instead of specifying the object name and event as tag attributes, VBScript lets you combine the two in a function name, separated by an underscore character, as in:

```
<script language="VBScript" type="text/vbscript">
Function upperAll_onclick
    script statements
End Function
</script>
```

The tag for the element requires only the `id` attribute to make the association.

`removeEventListener( )`, add and remove, respectively, the power to "hear" an event of a particular type as it passes by the node during event propagation (described later

in this Online Section). Parameters for both methods are the same, so we'll focus on how to perform the event binding portion.

The syntax for the `addEventListener( )` method is:

```
elementReference.addEventListener("eventType", functionReference, captureSwitch);
```

An *eventType* value is a string indicating the formal event type, which is the event name without the "on" prefix (i.e., just `click` instead of `onclick`). A function reference is the same kind that you use for object property event binding, including an anonymous function, if desired. The W3C DOM jargon calls the function invoked by an event listener an *event listener function*, which means little more than the function should have a parameter variable to receive the `event` object that automatically gets passed to it. The third parameter is a Boolean value that determines whether the node should "listen" for the event in the capture portion of event propagation (described later in this chapter). The typical setting of this parameter is `false`.

As with Microsoft's `attachEvent( )` method, you may queue up multiple event listener functions to be invoked for the same event targeting the same node. Unlike `attachEvent( )`, however, a given event invokes associated event listener functions in the order in which they were added (first-in, first-executed).

To remain true to the W3C model, the specification permits browsers to accept traditional event binding mechanisms, including tag attributes. Such bindings are to behave as if the code invokes `addEventListener( )` with the third parameter automatically set to `false`. This flexibility allows a browser such as Mozilla to implement the W3C DOM model, while allowing scripters to use event binding syntax that is compatible with other browsers, including older versions. But by using the newer syntax, you can explore several new event types that are linked directly to the W3C DOM's architecture. See Chapter 3 of *Dynamic HTML*, Third Edition for more details on W3C DOM events and `event` object properties.

Binding Multiple Events

If you are designing either a big project, or one that relies on several *.js* libraries, each of which needs initialization after the document loads (because they rely on element references that likely won't be valid until the browser has interpreted all of the HTML markup), you will want to bind multiple function calls to the window's `load` event. The typical ad hoc way to accomplish this is to assemble a customized initialization function in the document that invokes the desired functions, as in the following example:

```
function init( ) {
    initModule1( );  // from module1.js
    initModule2( );  // from module2.js
}
window.onload = init;
```

This approach, however, means that you'll have to change every such initialization function in every document of your project if you add a new module or change the name of the initialization routine in any one of them. That's not an appealing maintenance prospect.

Instead, you can queue up event handler function calls in such a way that each module simply adds its own initialization routine to the queue as it loads. When the `load` event fires, the calls to handler functions are made in the order in which they added themselves to the queue. The following example of a queue-building function should be among the first scripts loaded in a document (before any other *.js* files load):

```
function addOnLoadEvent(func) {
    var oldQueue = (window.onload) ? window.onload : function( ) {};
    window.onload = function( ) {
        oldQueue( );
        func( );
    }
}
```

Then, inside a *.js* library, and after the library's initialization function is defined, a single statement invokes the function, passing a reference to the initialization function, as in:

```
addOnLoadEvent(dragLibInit);
```

This function works reliably only if you do not have any other `onload` event bindings elsewhere in your documents through other means. In other words, do not attempt to mix this function with a page that has an `onload` attribute set in the `<body>` tag.

If you want to use the `attachEvent( )` and `addEventListener( )` methods where available, you can extend the `addOnLoadEvent( )` function to accommodate those methods, as well as support older browsers (e.g., Netscape 4 and IE 4) with the code shown above. The following example provides not only that all-encompassing function, but some sample event handler functions and calls to `addOnLoadEvent( )` that demonstrate how to pass parameters to an event handler function:

```
// sample functions
function do1( ) {
    alert("Doing 1");
}
function do2( ) {
    alert("Doing 2");
}
function do3(arg) {
    alert("Doing " + arg);
}

// generic load event binder
function addOnLoadEvent(func) {
    if (window.addEventListener) {
        window.addEventListener("load", func, false);
    } else if (window.attachEvent) {
```

```
            window.attachEvent("onload", func);
        } else {
            var oldQueue = (window.onload) ? window.onload : function( ) {};
            window.onload = function( ) {
                oldQueue( );
                func( );
            }
        }
    }

    // sample bindings
    addOnLoadEvent(do1);
    addOnLoadEvent(do2);
    addOnLoadEvent(function( ) {do3("whatever I want.");});
```

One important hazard of using the combination of methods shown above is that, as noted earlier, the IE `attachEvent( )` method causes queued calls to event handler functions to occur in the reverse order of the other event bindings operating in the function. If your libraries have dependencies on loading order, this discrepancy can cause problems. In that case, you may prefer to go with the simpler function queue approach shown earlier.

You can also extend this load event mechanism to other event types, and even build a generic function that accepts additional parameters to reference any desired element objects. Example VI-1 demonstrates a small library (*eventsManager.js*) that combines the window `load` event queue mechanism with functions that add and remove events for other element objects. It includes code that allows the library to work with older browsers, such as IE/Mac and Netscape 4 or earlier. This library is loaded into examples later in this chapter.

Example VI-1. eventsManager.js Library

```
function addEvent(elem, evtType, func, capture) {
   capture = (capture) ? capture : false;
   if (elem.addEventListener) {
      elem.addEventListener(evtType, func, capture);
   } else if (elem.attachEvent) {
      elem.attachEvent("on" + evtType, func);
   } else {
      // for IE/Mac, NN4, and older
      elem["on" + evtType] = func;
   }
}

function removeEvent(elem, evtType, func, capture) {
   capture = (capture) ? capture : false;
   if (elem.removeEventListener) {
      elem.removeEventListener(evtType, func, capture);
   } else if (elem.attachEvent) {
      elem.detachEvent("on" + evtType, func);
   } else {
      // for IE/Mac, NN4, and older
```

Example VI-1. eventsManager.js Library (continued)

```
      elem["on" + evtType] = null;
   }
}

function addOnLoadEvent(func) {
    if (window.addEventListener || window.attachEvent) {
        addEvent(window,"load", func, false);
    } else {
        var oldQueue = (window.onload) ? window.onload : function( ) {};
        window.onload = function( ) {
            oldQueue( );
            func( );
        }
    }
}
```

Preventing Default Event Actions

It is not uncommon to script an event to execute statements immediately prior to an element carrying out its normal activity in response to a user action. For example, a form's text field validation typically operates in response to the `submit` event of the `form` element. Without any kind of event, a `form` element obeys the submit-type `input` button, and sends the form's contents to the URI specified by the `action` attribute. But if you bind a `submit` event to that `form` element, and if the validation routines spot an error (e.g., a required text box is empty), the script can alert the user and prevent the default submission action from taking place. Many other elements and their events can benefit from this script technique.

You have several ways to prevent an element's default action, depending on the event binding style you use and the browsers you need to support. Some techniques work across all scriptable browsers.

Setting the return Value

When your events are in the form of element attributes, you can cancel the element's default action if the last statement of the event assignment statement evaluates to `return false`. This is different from simply having the handler function end with `return false`. The `return` statement must be in the value assigned to the event attribute.

The easiest way to implement this approach is to include a `return` statement in the event binding expression itself, while the function invoked by the handler returns `true` or `false` based on its calculations. For example, if a form requires validation prior to submission, you can have the `submit` event invoke the validation routine. If the routine finds a problem somewhere, it returns `false` and the submission is canceled because the entire event expression evaluates to `return false`; otherwise, the

function returns true and the submission proceeds as usual. Such a form element looks like the following:

```
<form method="POST" action="http://www.megaCo.com/cgi-bin/entry"
onsubmit="return validate(this);">
```

This technique also allows you to have a link navigate to a hardcoded URL for non-scriptable browsers, but execute a script when the user has a scriptable browser:

```
<a href="someotherURL.htm" onclick="doNavigation( ); return false;">...</a>
```

Here, the return false statement is set as the final statement of the event; it does not have to trouble the called function for a return value because all scriptable browsers are to follow the scripted navigation path.

The event.returnValue Property (IE 5 and Later)

Starting with Version 5 (Windows and Mac), IE's event object has a Boolean property, returnValue, that controls whether the element's default action occurs. The default value of the property is true. But to prevent the default action, your function script sets its value to false. The following IE-only function could be invoked from the keypress event of a text box (onkeypress="numberPlease( );" or txtBoxRef.onkeypress=numberPlease;). Its job is to let only numbers appear in the field by preventing the keypress event from performing its default action for other characters:

```
function numberPlease( ) {
    var charCode = event.keyCode;
    if (charCode < 48 || charCode > 57) {
        alert("Only whole numbers are allowed.");
        event.returnValue = false;
    }
}
```

Be careful not to confuse the event.returnValue property with the purpose of the JavaScript return statement. Use the former to control an event target's default behavior; use the latter when a function must return a value to a calling statement.

W3C preventDefault( ) Method

Instead of using a property of its event object for this purpose, the W3C event model gives the event object the preventDefault( ) method. You can invoke this method in an event listener function to stand in the way of the event continuing to the element. The syntax for its use is not far different from the IE returnValue property. The following function is a W3C version of the numbers-only real-time input checker. Due to an egregious bug in Mozilla versions prior to 1.4, however, the preventDefault( ) method works reliably in Netscape 6.x and 7.0.x only on event listeners that are added by way of tag attributes. Therefore, for the following function to do its job effectively including those browsers, it should be invoked from an input element tag's attribute (onkeypress="numberPlease(event);"):

```
function numberPlease(evt) {
    var charCode = evt.charCode;
    if (charCode < 48 || charCode > 57) {
        alert("Only whole numbers are allowed.");
        evt.preventDefault( );
    }
}
```

Cross-Browser Techniques

As with many event model incompatibilities, you can combine the syntax into a single function, letting object detection handle the branching for you, as shown in the following example. For good measure, the following function demonstrates branching techniques that can be used with all types of event bindings. A little bit of additional event key code detection allows helpful noncharacter keys (arrows, **Tab**, and **Backspace**, for example) to perform their normal jobs (in Mozilla, the `charCode` property for noncharacter keys is 0). The function is a cross-browser, backward-compatible version of the text box filter function shown earlier. This version works even with Netscape 4, provided you bind events via tag attributes, and make the assignment evaluate to `return false` (e.g., `onkeypress="return numberPlease(event);"`).

```
function numberPlease(evt) {
    evt = (evt) ? evt : ((window.event) ? window.event : null);
    if (evt) {
        var charCode = (evt.charCode || evt.charCode == 0) ? evt.charCode :
                       ((evt.keyCode) ? evt.keyCode : evt.which);
        if (charCode > 13 && (charCode < 48 || charCode > 57)) {
            alert("Only whole numbers are allowed.");
            if (evt.returnValue) {
                evt.returnValue = false;
            } else if (evt.preventDefault) {
               evt.preventDefault( );
            } else {
                return false;
            }
        }
    }
}
```

Event Propagation

In some DHTML applications, it is not efficient to have target elements process events. For example, if you have a page that allows users to select and drag elements around the page, it is quite possible that one set of centralized functions can handle that operation for all elements. Rather than bind events to each of those elements, it is more economical to have the mouse-related events go directly to an object or element that has scope over all the draggable elements. In other words, one event and event handler function can do the job of a dozen. For this kind of treatment to work,

events must be able to propagate through the hierarchy of objects or nodes in the document. IE 5 and later and the W3C event models share some, but not all, event propagation schemes. For the most typical applications, you can easily equalize the small differences in implementation details and syntax you use to override the natural flow.

W3C DOM event propagation can be summarized thus: in response to a user or system action, an event starts at the outermost container and follows the most direct route ("trickles down") through the node container hierarchy to the intended target; after it reaches its target, the event reverses course and "bubbles upward" through the same node hierarchy back to the top, from which it disappears. The trickle-down portion of the journey is called the *capture* phase, while the return trip is called the *bubbling* phase. The IE propagation model consists only of the bubbling phase. While IE 5 and later has an event feature related to capture (described later in this chapter), its operation is not along the lines of the W3C capture phase of propagation.

To better understand W3C DOM event propagation, consider the following skeletal structure of an HTML document (head element omitted for demonstration purposes):

```
<html>
<body>
    <form>
        <div id="div1">
            <input id="txt1" type="text">
        </div>
        <div id="div2">
            <input id="txt2" type="text">
        </div>
    </form>
</body>
</html>
```

As the user types a character into the `txt2` text input field, a `keypress` event begins its journey at an outermost container in Mozilla (and others), works its way through containers on its way to the text box (where IE's event starts), and then goes back to the outermost container. In bubble mode, the precise top-level container varies with browser. For Mozilla and Safari, the `window` object is the master container for event propagation purposes; Opera skips to the `window` before its final trigger at the `document`; IE bubbles no further beyond the `document` node. To help you visualize propagation sequences, Figure VI-1 depicts the `keypress` event propagation sequence through the objects of this document for four different browser types.

You can assign an event for any and all of the nodes in the hierarchy to process the event. In W3C DOM browsers, you can assign different event handler functions to the same node, but for different propagation modes, if you like. Event bubbling isn't

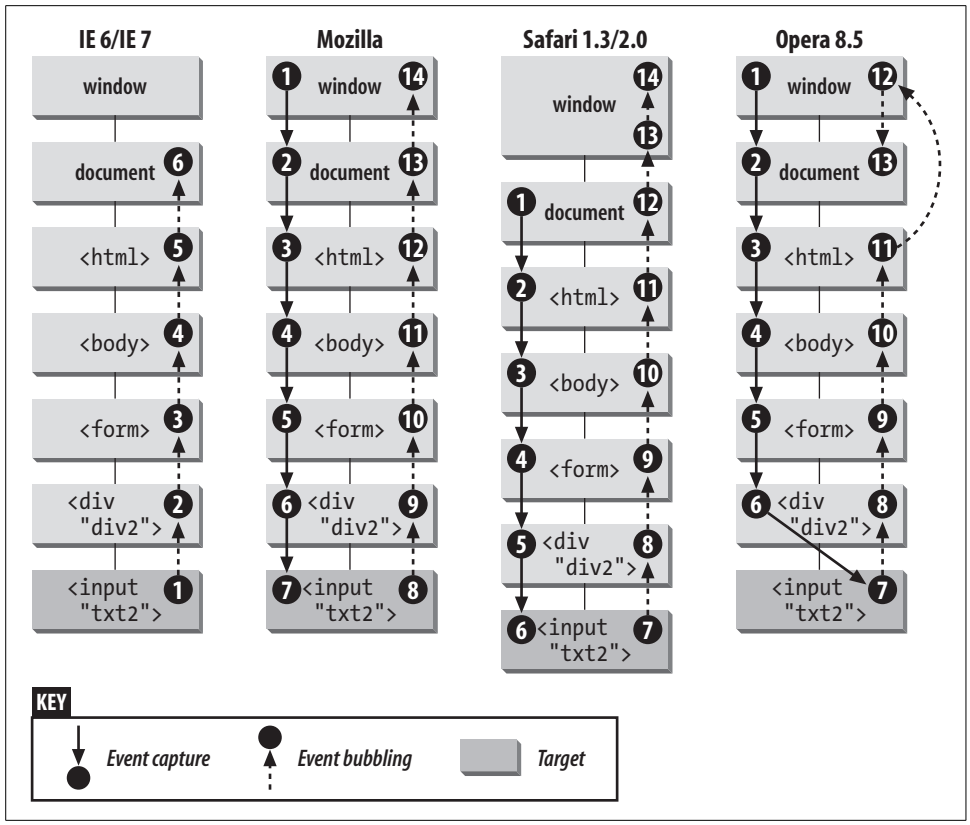


Figure VI-1. Sample event propagation sequences

as archaic as it sounds. In fact, it's quite flexible if you're careful to avoid conflicts that may occur at higher containment levels.

Event Bubbling

Event bubbling is the default propagation path for most events starting in IE 4 and W3C DOM browsers. Some system-fired events work only in their target elements. For example, if you have a `load` event listener assigned to a few `img` elements and the `body` element, you probably don't want an `img` element's `load` event to bubble up to the `window` object, firing each time an image's `src` property changes.

Event bubbling is often vital in scripting events for elements that display body text. The node-centric W3C DOM allows ordinary text nodes to be event listeners. Therefore, if you assign a mouse-related event to a text container, the target node of the event will be the text node within that container. By default the event bubbles outward to the container where the event can be easily processed, but the event object's properties point to the text node, not the container. This is different from the IE

event model, in which only elements are targets of events. To equalize this possibility in processing an event, your scripts must take the possibility of a text node target into account. Here is one cross-browser way to make sure that your function locates a reference to the element surrounding the text (or, as a last resort, the document node):

```
function myFunction(evt) {
    evt = (evt) ? evt : ((event) ? event : null);
    if (evt) {
        var elem = (evt.target) ? evt.target :
                   ((evt.srcElement) ? evt.srcElement : null);
        if (elem) {
            elem = (elem.nodeType == 1 || elem.nodeType == 9) ? elem :
                   elem.parentNode;
            // ok, now we're ready to work with the element/document
        }
    }
}
```

Starting with Mozilla 1.4, an event that targets an element's text node is automatically redirected to the containing element. Therefore, the target property references the element. You can still obtain a reference to the text node target by reading either the originalTarget or explicitOriginalTarget properties of the event object.

Most events triggered by user action with the mouse and keyboard bubble upward through the hierarchy from the target element. Conflicts can arise, however. For example: you assign a mousedown event for several images so they can swap *.jpg* files while the mouse is being held down. But you also define a mousedown event for the body element to act as a single handler to assist in dragging several positioned div elements around the page. To prevent the img element mousedown events from bubbling up through the hierarchy, you can explicitly instruct an event not to bubble beyond a specific element.

The cross-browser way of canceling event bubbling (starting in IE 4 and Mozilla) involves the Boolean cancelBubble property of the event object, adjusted within the event function. Although this property is not a member of the W3C DOM event object, Mozilla, Safari, and Opera implement it as a compatibility convenience. The default value for this property is false, meaning that event bubbling takes place. But if you set this property to true, the event does not bubble past the current event.

Set the cancelBubble property to true in a script statement executing in the current event's function. Thus, if you assign an event as an element property, the bubble cancelation can take place in the function invoked by the event:

```
function myFunction(evt) {
    evt = (evt) ? evt : ((event) ? event : null);
    if (evt) {
        // include next statement anywhere in this block
```

```
            evt.cancelBubble = true;
        }
    }
```

Only one event bubbles at any given instant, so this statement knows to cancel the right one. It also means that you can let an event bubble part of the way through the element hierarchy, but stop it at any desired element, so as not to interfere with other elements higher up the chain.

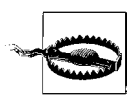
W3C Event Capture

Although event bubbling is the default mechanism in modern browsers, a propagating event in modern W3C DOM browsers starts its life by trickling down to the target. If you place an event listener for the event's type at a higher level, however, it will ignore the event as it trickles down unless event capture for that element and that event type is turned on.

To engage event capture, use the same `addEventListener( )` method that the W3C event model prefers for event binding. The third parameter is a Boolean value that controls event capture. When you set the parameter to `true`, the element invokes the listener function during capture phase; after that function completes its task, the event continues its journey toward to the target element. Therefore, it is perfectly "legal" to add two separate event listeners to an element for the same event type. In capture phase, one listener function runs; in bubbling phase, another listener function runs.

Using the same three parameters, you can eliminate the event listener for the desired propagation phase with the `removeEventListener( )` method. Thus, you could temporarily engage capture-phase processing and remove it without disturbing bubbling-phase event processing for the same element and event type.

At any point along W3C event propagation, you can prevent the event from going any further by invoking the `event` object's `stopPropagation( )` method in a statement inside the listener function. An event handler function that invokes this method when an event is being processed in capture mode prevents all further propagation of the event, preventing the event from reaching its target or bubbling up thereafter.



Although Mozilla, Safari, and Opera implement the IE `cancelBubble` property as a compatible alternative to the `stopPropagation( )` method, setting `cancelBubble` to `true` does not alter capture mode propagation in Safari. Only the `stopPropagation( )` method halts capture mode event propagation in Safari.

The W3C root `event` object (and therefore event objects of all types) implements a handful of other properties that may be helpful while processing events within the two-way propagation model. Table VI-4 lists these properties, for which IE (through

Version 7 on Windows) provides no analogues. A shared event listener function might use these (and other properties) to build code branches that execute when the desired combination of conditions exist (e.g., when the target's class name is "foo" and the event is being processed from a container of several elements of the same class).

Table VI-4. W3C event object propagation properties

Property	Description
`bubbles`	Boolean `true` if event can bubble
`currentTarget`	Reference to the node whose event listener invoked the current listener function
`eventPhase`	Integer indicating in which phase the event listener is processing (`1` is capture; `2` is at target; `3` is bubbling)

IE/Windows Event Capture

Microsoft's view of event capture is quite different from the W3C view. IE 5 and later event capture operates only with mouse events. In fact, when you invoke an element object's `setCapture( )` method, you instruct the browser to direct *all* mouse events on the page to that element rather than to their targets. Events bubble up from the capturing element, unless canceled.

This event mechanism is intended primarily for temporary activation within a page. For example, the `body` element can contain an `oncontextmenu` event handler that waits for a Windows user to click the right (nondominant) mouse button. You can take this opportunity not only to block the display of the browser's own context menu (by setting `event.returnValue` to `false`), but also to display your own menu composed of DHTML positioned elements. While the custom menu is visible, you want all mouse events to head for the menu so that nothing else on the page is accessible via the mouse until either a choice is made from the menu or the right mouse button is clicked again. Either action hides the custom context menu and invokes `releaseCapture( )` to allow mouse events to reach their normal targets again.

It's not uncommon for IE/Windows to implement proprietary DOM features that allow web applications to mimic operating-system-specific behaviors. In an intranet development environment targeting IE/Windows only, this tactic makes perfect sense. But such tight integration reduces the likelihood that these features will become part of an operating-system-agnostic W3C recommendation. It also makes it more difficult for Microsoft to implement some W3C recommendations that conflict with existing mechanisms.

Understanding Keyboard Event Data

As the W3C DOM working group discovered, keyboard events in a Unicode world are tricky things to mold into an acceptable global standard. Thus, keyboard events weren't covered in DOM Level 2 and, as of this writing, are still in Working Draft state as a Level 3 module. But keyboard events in one form or another have been implemented in browsers since Netscape 4 and IE 4. The W3C DOM Level 3 Events module Working Draft adopts some existing terminology, but also adds new syntax that keyboard event processing may adopt in the future. In the meantime, we have "old-fashioned" keyboard events, with processing that is not always straightforward due to differences in `event` object details across browser versions.

The most important data related to a keyboard event is the identity of either the physical key being activated or the character generated by that key. These are not the same things. Every key has a numeric code associated with it. For example, a U.S. English keyboard assigns the number 65 to the key labeled **A**. That same key, however, can produce at least two different characters (`A` and `a`) on every U.S. computer, and even more characters on operating systems like the Macintosh (where the **Option** and **Option-Shift** modifier keys let that **A** key generate even more characters).

It so happens that the code "65" is also the ASCII and Unicode value of the uppercase `A` letter (this isn't a coincidence as much as it reflects the English-centric basis of early computing). The character codes for the `A` and `a` characters are 65 and 97, respectively. For some scripting tasks, the character code is important—such as whether the character is a numeral, regardless of whether the user pressed a top row keyboard key or a numeric keypad key; for other tasks, the key pressed is important—such as whether the user pressed the **PageDown** key (which doesn't have a character associated with it).

Complicating the issue is that the IE `event` object has only one property value that conveys a code for a keyboard event (`event.keyCode`). Mozilla, Safari, and IE/Mac have a second `event` object property (`charCode`) that conveys additional event information. To expose both the key and character codes, even with only one property, the browsers let the different keyboard events deliver different information. To read the key code, use only the `keydown` or `keyup` events; use the `keypress` event to read the character code.

You must still reconcile the `event` object property differences among the browsers to work with keyboard events (although some browsers implement both approaches). The key code for the `keydown` and `keyup` events is available from the `keyCode` property of the `event` objects in IE and other browsers. For the `keypress` event's character code, use the `keyCode` property for IE, Safari, and Opera and `charCode` property for Mozilla browsers. Safari and IE 5/Mac provide the same character code data for both the `keyCode` and `charCode` properties of an `keypress` event.

W3C DOM Event Futures

The W3C DOM Level 3 Events module Working Draft divides what we currently think of as keyboard events into two separate event categories: text and keyboard. A `textinput` event is roughly equivalent to a `keypress` event, and reveals the character (via the `event.data` property) derived from the keystroke (or other input method); a keyboard event is either of type `keydown` or `keyup`, and reveals information (via the `event.keyIdentifier` property) about the key that is physically activated by the user. As of late-2006, no mainstream browser implements these recommendations.

To see how these different events and properties expose codes to scripts, Example VI-2 dynamically displays as many keyboard- and character-related `event` object properties as the browser supports (see Figure VI-2). In the course of processing each event type, the event listener functions display the `keyCode` properties for all browsers. If a browser also supports the `charCode` property, its values appear in the table.

You can witness interesting characteristics of the `keydown` and `keyup` events when you use modifier keys in some browsers (IE and Opera). For example, if you type an uppercase `A` by pressing and holding the **Shift** key before typing the **A** key, you see that the `keydown` event fires, and that the keycode for the **Shift** key (16) appears in the `keyCode` property. But when you then press the **A** key, a new `event` object comes on the scene, with its own `keyChar` property value.

Note that this is not how your scripts detect whether a modifier key is pressed during a keyboard or other event. The `event` object in both models has Boolean properties—`altKey`, `ctrlKey`, and `shiftKey`—that your character key event can inspect. If the `keyChar` property of a `keydown` event indicates the **C** key, and if the `event` object's `ctrlKey` value is `true`, the user has typed **Ctrl-C**.

Example VI-2. Keyboard events and codes

```
<html>
<head>
<title>Keyboard Events and Codes</title>
<style type="text/css">
body {font-family: Arial, sans-serif}
h1 {text-align: right}
td {text-align: center}
</style>
<script language="JavaScript" type="text/javascript" src="eventsManager.js">
</script>
<script language="JavaScript" type="text/javascript">
// array of table cell ids
var tCells = ["downKey", "pressKey", "upKey", "downChar", "pressChar",
"upChar", "keyTarget", "character"];
```

Example VI-2. Keyboard events and codes (continued)

```
// clear table cells for each key down event
function clearCells( ) {
    for (var i = 0; i < tCells.length; i++) {
        document.getElementById(tCells[i]).innerHTML = "&mdash;";
    }
}

// display target node's node name
function showTarget(evt) {
    var node = (evt.target) ? evt.target : ((evt.srcElement) ?
               evt.srcElement : null);
    if (node) {
        document.getElementById("keyTarget").innerHTML = node.nodeName;
    }
}

// decipher key down codes
function showDown(evt) {
    clearCells( );
    evt = (evt) ? evt : ((event) ? event : null);
    if (evt) {
        document.getElementById("downKey").innerHTML = evt.keyCode;
        if (evt.charCode) {
            document.getElementById("downChar").innerHTML = evt.charCode;
        }
        showTarget(evt);
    }
}

// decipher key press codes
function showPress(evt) {
    evt = (evt) ? evt : ((event) ? event : null);
    if (evt) {
        document.getElementById("pressKey").innerHTML = evt.keyCode;
        if (evt.charCode) {
            document.getElementById("pressChar").innerHTML = evt.charCode;
        }
        showTarget(evt);
        var charCode = (evt.charCode) ? evt.charCode : evt.keyCode;
        // use String method to convert back to character
        document.getElementById("character").innerHTML =
            String.fromCharCode(charCode);
    }
}

// decipher key up codes
function showUp(evt) {
    evt = (evt) ? evt : ((event) ? event : null);
    if (evt) {
        document.getElementById("upKey").innerHTML = evt.keyCode;
        if (evt.charCode) {
            document.getElementById("upChar").innerHTML = evt.charCode;
```

Example VI-2. Keyboard events and codes (continued)

```
        }
        showTarget(evt);
    }
}

// bind events to text box
function setKeyboardEvents( ) {
    addEvent(document.forms["myForm"].elements["entry"], "keydown", showDown, false);
    addEvent(document.forms["myForm"].elements["entry"], "keypress", showPress, false);
    addEvent(document.forms["myForm"].elements["entry"], "keyup", showUp, false);
}
// do event binding now that elements exist
addOnLoadEvent(setKeyboardEvents);
</script>
</head>
<body>
<h1>Key and Character Codes vs. Event Types</h1>
<hr>
<p>Enter some text with uppercase and lowercase letters:<br>
<form name="myForm" id="myForm">
<input type="text" id="entry" size="60">
</form>
</p>
<table border="2" cellpadding="5" cellspacing="5">
<caption>Keyboard Event Properties</caption>
<tr><th>Data</th><th>keydown</th><th>keypress</th><th>keyup</th></tr>
<tr><td>keyCode</td>
    <td id="downKey">&mdash;</td>
    <td id="pressKey">&mdash;</td>
    <td id="upKey">&mdash;</td>
</tr>
<tr><td>charCode</td>
    <td id="downChar">&mdash;</td>
    <td id="pressChar">&mdash;</td>
    <td id="upChar">&mdash;</td>
</tr>
<tr><td>Target</td>
    <td id="keyTarget" colspan="3">&mdash;</td>
</tr>
<tr><td>Character</td>
    <td id="character" colspan="3">&mdash;</td>
</tr>
</table>
</body>
</html>
```

Figure VI-2 shows some sample output for Example VI-2.

Regardless of the operating system you use, you should try Example VI-2 on at least two different browsers. This way you can see how the three events stuff values into event object properties in different event models.

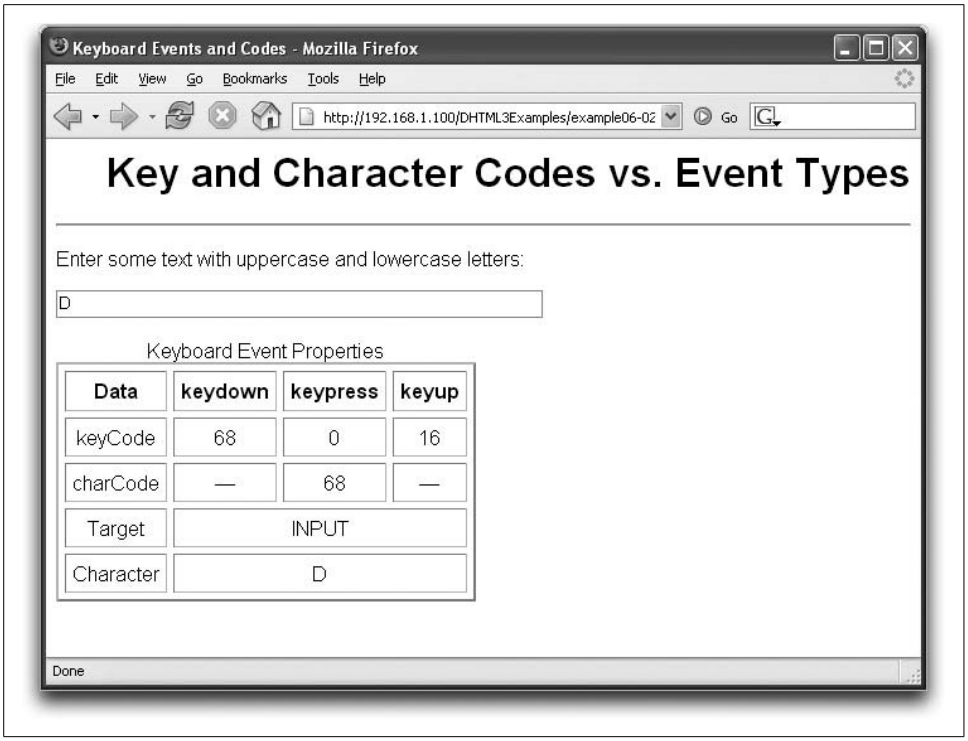


Figure VI-2. Viewing keyboard event data

Dragging Elements

The final example in this chapter, Example VI-3, demonstrates binding and unbinding events as needed for the purpose of dragging and dropping elements. Although earlier editions of this book included code to work in Netscape Navigator 4's outdated layer environment, Example VI-3 employs techniques that preclude Netscape 4 from allowing the dragging operations to work. In particular, Netscape does not allow scripts to find elements in a document based on their `class` attributes—a facility that puts less of a burden on programmers to denote which elements are draggable, without imposing more rigid (and prone to maintenance breakage) element naming demands. With this library, all you need to do to make an element draggable is to use CSS to turn it into an absolute-positioned element, and then include the `draggable` identifier in the element's `class` attribute (remember that you can assign multiple class names to an element). The library is compatible with browsers starting with IE 4/Windows, IE 5/Mac, Mozilla 1.0, Opera 5, and Safari.

The only other requirements for Example VI-3 are two *.js* libraries that you've seen earlier in this book. One is the *DHTMLapi3.js* library from Online Section V; the other is the *eventsManager.js* library from earlier in this chapter. Although

Example VI-3 shows the dragging code within the document for the sake of convenience, you would normally put this code, too, into its own external *.js* library.

This version implements all dragging code inside a single custom object, named `dragObject`. This object handles all event bindings, dragging operation, and cleanup. By encapsulating all properties and methods inside the object, the code keeps those names out of the global identifier space, thus minimizing potential conflicts with other scripts and element IDs.

The dragging system implemented in this example has a simple design behind it. Three mouse events—`mousedown`, `mousemove`, and `mouseup`—control the action. Only the `mousedown` event is assigned at first; the others are bound to the element being dragged, and then released after the user drops the element. Initial event binding occurs in the `dragObject.init( )` method. After grabbing a collection of all elements nested inside the body, the method assigns the `mousedown` event type to each element whose `className` property contains `draggable`.

When a user clicks on one of the draggable elements, the `dragObject.engage( )` method preserves a reference to the element in the `dragObject.selectedObject` property (other methods will refer to it). In some browsers, if the draggable element contains other nodes, the event target thinks it's the nested node, rather than the draggable container. In that case, the code here works its way out from the target to the draggable container to make sure the desired element is registered as being draggable. Other jobs for the `dragObject.engage( )` method include:

- Raising the CSS `z-index` property so that the element is in front of all others. If your elements have specific `z-index` values, you could also preserve the original setting as another `dragObject` property for restoration after dragging.
- Calculating the pixel offset within the container where the `mousedown` event occurred. This is necessary to allow the dragging to position the dragged element up and to the left of the cursor accurately so that the element doesn't jump to position its top-left corner under the cursor. W3C DOM and IE browsers calculate these coordinates differently, with IE requiring consideration of scrolling of the current page.
- Binding temporary `mousemove` and `mouseup` events to the `document` node (via the "private" `dragObject.setDragEvents( )` method). This allows events that might slip out of the element (e.g., the user scoots the mouse too fast for the dragging to catch up) to be processed as if they were inside the dragged element.
- Preventing the `mousedown` event from propagating any further or returning any values. Unless these actions are cancelled, Macintosh users, in particular, see contextual menus if they press and hold the mouse button in a browser window.

As the user moves the mouse (with the button still pressed), the `dragObject.dragIt( )` method runs repeatedly. It invokes the `DHTMLAPI.moveTo( )` method (from the *DHTMLapi3.js* library) to keep the element positioned under the cursor. Because the

target of the event might not be the container you wish to be dragged, the method refers to the `dragObject.selectedObject` property to obtain the reference of the real draggable element.

The instant the user releases the mouse button, the `dragObject.releaseDrag()` method performs three operations:

- Restoring the CSS `z-index` property to zero (or saved value, if you wish to implement it that way).
- Removing the `mousemove` and `mouseup` event bindings from the just-dragged element.
- Setting `dragObject.selectedObject` to `null` because no element is currently being dragged.

Draggable elements in Example VI-3 consist of two images and one `div` element containing nothing more than some text. The code works with any rendered element.

Example VI-3. Dragging elements around the window

```
<html>
<head>
<title>It's a Drag</title>
<style type="text/css">
.draggable {position:absolute}
#textbox {left: 200px; top: 225px; width: 150px;
    height: 50px; border: solid black 1px;
    background-color: lime; z-index: 0; text-align: center}
#imgA {left: 50px; top: 200px; width: 120px; height: 90px;
       border: solid black 1px; z-index: 0}
#imgB {left: 110px; top: 245px; width: 120px; height: 90px;
       border: solid black 1px; z-index: 0}
</style>
<script type="text/javascript" src="eventsManager.js"></script>
<script type="text/javascript" src="DHTML3api.js"></script>
<script type="text/javascript">
// dragObject contains data for currently dragged element
var dragObject = {
    selectedObject : null,
    offsetX : 0,
    offsetY : 0,
    // invoked onmousedown
    engageDrag : function(evt) {
        evt = (evt) ? evt : window.event;
        dragObject.selectedObject = (evt.target) ? evt.target : evt.srcElement;
        var target = (evt.target) ? evt.target : evt.srcElement;
        var dragContainer = target;
        // in case event target is nested in draggable container
        while (target.className != "draggable" && target.parentNode) {
            target = dragContainer = target.parentNode;
        }
        if (dragContainer) {
            dragObject.selectedObject = dragContainer;
```

Example VI-3. Dragging elements around the window (continued)

```
            DHTMLAPI.setZIndex(dragContainer, 100);
            dragObject.setOffsets(evt, dragContainer);
            dragObject.setDragEvents( );
            evt.cancelBubble = true;
            evt.returnValue = false;
            if (evt.stopPropagation) {
                evt.stopPropagation( );
                evt.preventDefault( );
            }
        }
        return false;
    },
    // calculate offset of mousedown within draggable element
    setOffsets : function (evt, dragContainer) {
        if (evt.pageX) {
            dragObject.offsetX = evt.pageX - ((dragContainer.offsetLeft) ?
                      dragContainer.offsetLeft : dragContainer.left);
            dragObject.offsetY = evt.pageY - ((dragContainer.offsetTop) ?
                      dragContainer.offsetTop : dragContainer.top);
        } else if (evt.offsetX || evt.offsetY) {
            dragObject.offsetX = evt.offsetX - ((evt.offsetX < -2) ?
                      0 : document.body.scrollLeft);
            dragObject.offsetY = evt.offsetY - ((evt.offsetY < -2) ?
                      0 : document.body.scrollTop);
        }
    },
    // invoked onmousemove
    dragIt : function (evt) {
        evt = (evt) ? evt : window.event;
        var obj = dragObject;
        if (evt.pageX) {
            DHTMLAPI.moveTo(obj.selectedObject, (evt.pageX - obj.offsetX),
                (evt.pageY - obj.offsetY));
        } else if (evt.clientX || evt.clientY) {
            DHTMLAPI.moveTo(obj.selectedObject, (evt.clientX - obj.offsetX),
                (evt.clientY - obj.offsetY));
        }
        evt.cancelBubble = true;
        evt.returnValue = false;
    },
    // invoked onmouseup
    releaseDrag : function (evt) {
        DHTMLAPI.setZIndex(dragObject.selectedObject, 0);
        dragObject.clearDragEvents( );
        dragObject.selectedObject = null;
    },
    // set temporary events
    setDragEvents : function ( ) {
        addEvent(document, "mousemove", dragObject.dragIt, false);
        addEvent(document, "mouseup", dragObject.releaseDrag, false);
    },
    // remove temporary events
```

Example VI-3. Dragging elements around the window (continued)

```
    clearDragEvents : function ( ) {
        removeEvent(document, "mousemove", dragObject.dragIt, false);
        removeEvent(document, "mouseup", dragObject.releaseDrag, false);
    },
    // initialize, assigning mousedown events to all
    // elements with class="draggable" attributes
    init : function (tagName) {
        var elems = [];
        if (document.all) {
            // IE 5 & 5.5 don't know wildcard for getElementsByTagName
            // so use document.body.all, which lets IE 4 work OK
            elems = document.body.all;
        } else if (document.body && document.body.getElementsByTagName) {
            elems = document.body.getElementsByTagName("*");
        }
        for (var i = 0; i < elems.length; i++) {
            if (elems[i].className.match(/draggable/)) {
                addEvent(elems[i], "mousedown", dragObject.engageDrag, false);
            }
        }
    }
};
// set onload event via eventsManager.js
addOnLoadEvent(dragObject.init);
</script>
</head>
<body>
<h1>Element Dragging</h1>
<hr>
<div id="textbox" class="draggable">This is a draggable text box."</div>
<img id="imgA" name="imgA" class="draggable" alt="draggable image A"
src="myImage1.gif" width="120" height="90" border="0" />
<img id="imgB" name="imgB"  class="draggable" alt="draggable image B"
src="myImage4.gif" width="120" height="90" border="0"/>
</body>
</html>
```

Event Futures

That the W3C DOM Working Group has had some of its greatest and longest-lasting struggles with the Events modules in DOM Level 2 and 3 is not all that surprising. Many kinds of events are tied directly to an operating-system implementation, which makes it difficult for interested parties to reach consensus on a common denominator that acknowledges the way users interact with data within graphical user interfaces, while remaining implementation-independent. It may be some time before the many practical events available in IE for Windows are applicable and implementable across a range of operating systems and devices. In the meantime, the list of intrinsic events remains rather basic.

That Microsoft has, as of IE 7, ignored the W3C DOM event model leaves one to wonder. Is it waiting for the Level 3 keyboard event model to be finalized before implementing any of the W3C model? Or will IE continue with its own exclusive model? If you are trying to get your code in shape for a long, maintenance-free future, IE's delay in supporting the W3C event model works in your favor. The longer the delay, the larger the installed base of code that relies on the IE model, which means that the model will be supported longer in the future. Therefore, the two-model equalization techniques described in this chapter should serve your code well for quite awhile.

ONLINE SECTION VII

XMLHttpRequest and Ajax

Although blending XML data into an HTML document on the client was possible at least as early as Microsoft Internet Explorer 5—released to the world in August 1999—the concept didn't blaze across web developers' radar screens until it gained a catchy acronym in early 2005. That acronym—Ajax—stands for Asynchronous JavaScript and XML, and was coined by Jesse James Garrett.

More specifically, Ajax uses a scriptable object known as the `XMLHttpRequest` object to request server data (and post client data to a server) as a background process, invisible to the user (except perhaps for evidence of network traffic in modem indicators). As the name of the object implies, it is intended to fetch XML data via the time-tested Hypertext Transfer Protocol. Once an XML document is received by the object, client-side scripts can use Document Object Model properties and methods to examine and copy data, usually with the goal of adding to or replacing HTML document data currently on display. The benefit for the user is that instead of waiting for an entire page to be delivered from the server with only a little bit of updated information, a quick request and tiny bit of data can be sent to the client, where a script updates the HTML in the current document much more quickly and with less visual distraction. Ajax thus becomes one contributor to what is commonly called a Rich Internet Application—one that typically attempts to mimic the behavior and user experience of a standalone program.

A Brief History Lesson

The first implementation of the `XMLHttpRequest` object was available with Internet Explorer 5 for Windows, but it was only an ActiveX object (labeled Microsoft.XMLHTTP), which had to be loaded by way of IE's proprietary `ActiveXObject` constructor function. By the time Internet Explorer 5.5 came around, Microsoft had updated the ActiveX object and had given it a new name (Msxml2.XMLHTTP).

Meanwhile, the Mozilla engineers implemented the `XMLHttpRequest` object as a global object, meaning that a script can invoke the object's constructor function

directly, without resorting to ActiveX. An instance of Mozilla's object had all of the properties and methods of the Microsoft version, and even added some more as time went on. Apple also saw the potential of the object for both web development and other technologies it had up its sleeve (Dashboard, for example). This activity led to Safari gaining a global `XMLHttpRequest` object beginning with version 1.2. Opera 8 joined the party. Finally, Microsoft implemented the object as a global object for IE 7, which allows the same code to create an instance of the `XMLHttpRequest` object as the other browsers.

It should be noted that "XMLHttpRequest" is such a cumbersome term, it's not uncommon to see the term abbreviated as XHR in forum discussions and articles. Because the full name already contains two acronyms (XML and HTTP), you might call XHR a "macronym."

Application Design Considerations

Before diving into how to use the `XMLHttpRequest` object, it will be helpful to understand some of the challenges that you will likely face along the way when deploying an application that uses the technology. All too often in the DHTML world, high-flying ideas are quickly brought down to Earth when harsh realities of non-technical users' expectations get in the way. The earlier you can start planning for these challenges, the greater your chance at defining the right specifications for your application.

Ajax and the Server

In truth, Ajax technologies extend beyond just the browser `XMLHttpRequest` object. A robust application that takes advantage of the browser technology will also likely have a major server programming component to assist. The server will be doing jobs such as database retrieval, assembly of XML or JSON output for the client, and retrieving data from third-party sources to effect "mash-ups" of two or more external applications blended into your super-duper application. Application blending is often cited as one part of a tech trend that has (for better or worse) been labeled "Web 2.0."

The primary need for the server preprocessing is to bypass browser and scripting security restrictions that you have likely encountered while pursuing other application dreams. Specifically, the `XMLHttpRequest` object in the browser is subject to the identical same-origin security restrictions that affect scripting tactics such as cross-frame or cross-window communication. In the case of XMLHttpRequest, your scripts cannot access the data returned from the server unless it originates from the same domain from which the current web page arrived (even getting the technology to work across different servers of the same domain will lead to gut-wrenching workarounds). Your dream of client-side mash-ups just went out the window.

Another factor to consider is that if you intend to (or must) provide the same data to all visitors, including those who do not visit with a JavaScript-enabled browser, your server should be programmed to supply updates the old-fashioned way—through full-page refreshes. That combination of server- and client-side functionality is in the best spirit of DHTML, whereby those who visit with modern browsers benefit from the nifty technology (e.g., faster response), but every visitor has a chance to use the data. Since we're talking about DHTML in browsers, this Online Section addresses stricty the client-side deployment for script-enabled browsers.

Browser History and the Back Button

With the long heritage of web pages making individual requests for entire pages at every click of a link or button, users are quick to aim for the browser's Back button to return to the previous view of the page or site they're visiting. But now think for a minute what happens when a click of a link or button doesn't refresh the entire page, but simply fetches a few tidbits of data to update a portion of the current page. Unfortunately for the user, requests and postings made through the `XMLHttpRequest` object are not automatically logged anywhere, nor does the transaction register with the browser's history. A click of the Back button zooms the user back to a completely different page—perhaps a different web site altogether.

One possible workaround for this issue is to modify the `location.hash` property with each request so that its value contains information your scripts can use to recreate the current state of your DHTML page. Each change to the property is not only reflected in the browser's Address field, but the history also receives an entry. Unfortunately, Safari shows the spinning "loading" animation after you assign a value to the `location.hash` property, even though the page has completed loading.

URLs and Bookmarks

An issue related to the browser history problem is what happens when a user wants to add the current page and data as a browser bookmark (or, in IE's terminology, "favorite"). If a user has been doing a bunch of Ajax operations to reach a particular state of the data, how can that state be bookmarked?

Normally, a bookmark request grabs only the current URL of the page, which would be the initial state of the page prior to any user interaction. Google Maps faced this problem because a user could have navigated through dozens or hundreds of twists and turns around a map to reach a particular destination. To solve the problem (partially, anyway), the page provides a dynamically updated link that is hard-coded with enough information for the server to open the current page with the current map state—another reason to have the same capabilities duplicated on the server. A user, however, must know to right-click on the link to reach the contextual menu choice that bookmarks the clicked link.

Using XMLHttpRequest

If the `XMLHttpRequest` object causes consternation among first-timers, it's primarily because of two characteristics that are not like other DHTML objects you work with. First, you have to follow a specific sequence of operations to get the object to work correctly; second, it operates entirely in the background, leaving you to use your best JavaScript debugging tools and skills to solve problems. This section covers the first part, while debugging is addressed later.

Overview

The sequence of operations is not complicated. More code of a simple function responsible for loading an XML document is devoted to browser syntax differences than any other aspect.

All operations are performed on an instance of the `XMLHttpRequest` object, just like you do for the JavaScript `Date` object—spawning an instance that has specific information about just one "live" copy of the object. For the sake of convenience throughout this discussion, I'll call this instance a *request object*.

The following sequence is the typical way you work with a request object:

1. Null out a variable holding a previous request object (if any).
2. Create a new instance of the `XMLHttpRequest` object, assigning it to a variable.
3. Open the request object, specifying the type of request and URL.
4. Bind the request object to an event handler function that is to run upon completion of data retrieval.
5. Optionally, specify a request header, such as content-type.
6. Send the request.
7. Process the results, inspecting one or more properties of the request object containing the data.

In most applications, each instance of a request object should be stored as a global variable (or as a global custom object that incorporates the XMLHttpRequest mechanism code). This allows other functions to access the data of the request object.

Example VII-1 is a listing of an object-based approach to working with the `XMLHttpRequest` object. It allows for both reuse of the same instance for retrieving multiple XML documents as well as multiple instances if you wish to keep several instances running at once (although bear in mind that browsers can use no more than two simultaneous connections to the same domain). The next few sections refer to Example VII-1.

Example VII-1. Custom XMLHttpRequest object constructor function

```
// constructor function for an XML request object;
function XMLDoc( ) {
    var me = this; // for event handler parameter, below
    var req = null;
    // branch for native XMLHttpRequest object
    if (window.XMLHttpRequest) {
        try {
            req = new XMLHttpRequest( );
        } catch(e) {
            req = null;
        }
    // branch for IE/Windows ActiveX version
    } else if (window.ActiveXObject) {
        try {
            req = new ActiveXObject("Msxml2.XMLHTTP");
        } catch(e) {
            try {
               req = new ActiveXObject("Microsoft.XMLHTTP");
            } catch(e) {
               req = null;
            }
        }
    }
    // preserve reference to request object for later
    this.request = req;
    // "public" method to be invoked whenever
    this.loadXMLDoc = function(url, loadHandler) {
        if (this.request) {
            this.request.open("GET", url, true);
            this.request.onreadystatechange = function ( ) {loadHandler(me)};
            this.request.setRequestHeader("Content-Type", "text/xml");
            this.request.send("");
        }
    };
}
// create request object instances
var request1 = new XMLDoc( );
var request2 = new XMLDoc( );
```

Creating a Request Object

Thanks to browser implementation differences, more code lines go into creating an instance of a request object than any other part of the process of retrieving an XML document. These are the opening lines of the `XMLDoc( )` constructor function shown in Example VII-1. The top branch is for the native global `XMLHttpRequest` object so that browsers that implement both ways (such as IE 7) use the native object. The other branches load the ActiveX version, giving preference to the more modern version, Msxml2.XMLHTTP.

Request Object Methods

In typical operation, you primarily use methods of a request object to prepare the request and send it on its way. Different browsers implement different sets of methods (see Chapter 2 of *Dynamic HTML*, Third Edition), but for basic operation you can rely on the lowest common denominator of methods, dictated by Microsoft's earliest implementations. Table VII-1 lists those methods.

Table VII-1. Lowest common denominator methods of a request object

Method	Description
`abort( )`	Stops current request transaction
`getAllResponseHeaders( )`	Returns a list of headers arriving with the response
`getResponseHeader( )`	Returns a value of a specific header name
`open( )`	Assigns method, URL, and other attributes to a request waiting to be sent
`send( )`	Sends the previously specified request over the network
`setRequestHeader( )`	Sets a header name/value pair for a request waiting to be sent

Every visit of the request object requires both the `open( )` and `send( )` methods. The `open( )` method takes a minimum of two, and as many as five, parameters. The two required parameters are the transaction method (`GET` or `POST` as strings) and the destination URL as a string. URLs may be complete or relative. Three optional parameters are: Boolean flag for whether the transmission is executed asynchronously (the default is `true`); an account user name; and account password. The last two parameters are discouraged for client-side use because the values would be retrievable in a source code view.

The parameter to the `send( )` method is an empty string when the URL contains a search string conveying data submitted with a GET method or is a simple URL to a server program or XML file (as shown in Example VII-1). For submitting via the POST method (discussed later), the actual data to be submitted to the server (as a string) is the parameter of the `send( )` method.

Request Object Properties

As with request object methods, request object properties have a lowest common denominator set that work across all implementations. Scripts have need for most of these properties after the server data has been retrieved. Table VII-2 lists those properties.

Table VII-2. Lowest common denominator properties of a request object

Property	Description
`onreadystatechange`	Event property to bind to a function handler that acts during or after the request transaction
`readyState`	Request object status (0 = uninitialized; 1 = loading; 2 = loaded; 3 = interactive; 4 = complete)
`responseText`	String version of all data retrieved in last request
`responseXML`	XML document node (nodeType of 9) if data is an XML data type
`status`	Numeric status code returned by server (e.g., 200 for success)
`statusText`	Message string (if any) returned with status integer

All properties of Table VII-2 are read-only except for `onreadystatechange`. This is the property that lets your script direct the request object to invoke a particular function at every state change of the request object. Assign a function reference to this property after the request object has been opened, but before it has been sent. Object states (readable through the `readyState` property) are similar to the `readyState` property of other objects in the Internet Explorer object model. In a typical XMLHttpRequest transaction, the event fires a minimum of three times, once each in the states of loading (1), loaded (2), and complete (4). For larger chunks of data and under the influence of network and server load, the event may fire multiple times in the loading (1) state. In Example VII-1, the event handler function is called with a parameter consisting of a reference to the request object so that the function can receive a reference to the object and inspect its properties while processing the results.

Handling the Request Response

Among the first jobs of the function handler is to confirm that the request has completed successfully. The following code demonstrates a typical series of statements at the start of a handler function. As noted earlier, the handler function receives as its parameter a reference to the request object instance. Grab that object's `request` property, through which you can access properties that reveal everything your scripts want to know about the transaction.

```
function requestFunctionHandler(req) {
    req = req.request;
    if (req.readyState == 4 && req.status == 200) {
        var xDoc = req.responseXML;
        // further processing of document here
    }
}
```

Examine first whether the transaction has completed satisfactorily by looking for a `readyState` value of 4 and a `status` value of 200. If those conditions are met, then you can retrieve the XML document as a DOM node through the `responseXML` property. That value is a reference to the root node of the document. From that reference (the

variable xDoc in the example above) you can use W3C DOM node methods to obtain, for example, an array of all elements with a particular tag name (e.g., all elements coded with the <record> tag). After that, you're on your way to read XML element content and plug that data into placeholders in your HTML document, as described in more detail in Online Section IV.

Example VII-2 is a revised version of Example IV-10, using the object-based and reusable request object from Example VII-1 to load data alternately between two separate XML sources. Once the data is retrieved, it is parsed and inserted into an HTML table. This revision also uses the *eventManager.js* library from Online Section VI to handle event binding to the interactive buttons. Notice that the XML source URLs are passed to the request object via a call to one of the custom-defined methods, loadXMLDoc( ), while the drawTable( ) function contains knowledge about the tbody element it modifies.

Example VII-2. Object-based XMLHttpRequest application

```
<html>
<head>
<title>Blending External XML Data</title>
<style type="text/css">
body {background-color: #ffffff}
table {border-spacing: 0}
td {border: 2px groove black; padding: 7px}
th {border: 2px groove black; padding: 7px}
.ctr {text-align: center}
</style>
<script type="text/javascript" src="eventsManager.js"></script>
<script type="text/javascript">
// Draw table from parse XML document tree data
function drawTable(req) {
    req = req.request;
    tbodyID = "bowlData";
    if (req.readyState == 4 && req.status == 200) {
        var tr, td, i, j, oneRecord;
        var tbody = document.getElementById(tbodyID);
        clearTable(tbody)
        var xDoc = req.responseXML;
        if (!xDoc || !xDoc.childNodes.length) {
            alert("This example must be loaded from a web " +
            "server for Internet Explorer.");
            return;
        }
        // node tree
        var data = xDoc.getElementsByTagName("season")[0];
        // for td class attributes
        var classes = ["ctr","","","","ctr"];
        for (i = 0; i < data.childNodes.length; i++) {
            // use only 1st level element nodes
            if (data.childNodes[i].nodeType == 1) {
                // one bowl record
```

Example VII-2. Object-based XMLHttpRequest application (continued)

```
                oneRecord = data.childNodes[i];
                tr = tbody.insertRow(tbody.rows.length);
                td = tr.insertCell(tr.cells.length);
                td.setAttribute("class",classes[tr.cells.length-1]);
                td.appendChild(document.createTextNode(
                   oneRecord.getElementsByTagName("number")[0].firstChild.nodeValue));
                td = tr.insertCell(tr.cells.length);
                td.setAttribute("class",classes[tr.cells.length-1]);
                td.appendChild(document.createTextNode(
                   oneRecord.getElementsByTagName("year")[0].firstChild.nodeValue));
                td = tr.insertCell(tr.cells.length);
                td.setAttribute("class",classes[tr.cells.length-1]);
                td.appendChild(document.createTextNode(
                   oneRecord.getElementsByTagName("winner")[0].firstChild.nodeValue));
                td = tr.insertCell(tr.cells.length);
                td.setAttribute("class",classes[tr.cells.length-1]);
                td.appendChild(document.createTextNode(
                    oneRecord.getElementsByTagName("loser")[0].firstChild.nodeValue));
                td = tr.insertCell(tr.cells.length);
                td.setAttribute("class",classes[tr.cells.length-1]);
                td.appendChild(document.createTextNode(
                     oneRecord.getElementsByTagName("winscore")[0].firstChild.nodeValue +
                     " - " +
                     oneRecord.getElementsByTagName("losscore")[0].firstChild.nodeValue));
            }
        }
    }
    // prepare for replacement with data from another source
    function clearTable(tbody) {
        while (tbody.rows.length > 0) {
            tbody.deleteRow(0);
        }
    }
}

// constructor function for an XML request object;
function XMLDoc( ) {
    var me = this;
    var req = null;
    // branch for native XMLHttpRequest object
    if (window.XMLHttpRequest) {
        try {
            req = new XMLHttpRequest( );
        } catch(e) {
            req = null;
        }
    // branch for IE/Windows ActiveX version
    } else if (window.ActiveXObject) {
        try {
            req = new ActiveXObject("Msxml2.XMLHTTP");
        } catch(e) {
            try {
```

Example VII-2. Object-based XMLHttpRequest application (continued)

```
                req = new ActiveXObject("Microsoft.XMLHTTP");
            } catch(e) {
                req = null;
            }
        }
    }
    // preserve reference to request object for later
    this.request = req;
    // "public" method to be invoked whenever
    this.loadXMLDoc = function(url, loadHandler) {
        if (this.request) {
            this.request.open("GET", url, true);
            this.request.onreadystatechange = function ( ) {loadHandler(me)};
            this.request.setRequestHeader("Content-Type", "text/xml");
            this.request.send("");
        }
    };
}
// create request object instances
var request1 = new XMLDoc( );

// assign event handlers to two buttons, each tied to a different XML source
function setHandlers( ) {
    if (document.getElementById) {
        addEvent(document.getElementById("btn1"), "click",
            function( ) {request1.loadXMLDoc("superBowls.xml", drawTable);}, false);
        addEvent(document.getElementById("btn2"), "click",
            function( ) {request1.loadXMLDoc("superBowls2.xml", drawTable);}, false);
    }
}
addOnLoadEvent(setHandlers);

</script>
</head>
<body">
<h1>Super Bowl Games</h1>
<hr>
<form>
<input id="btn1" type="button" value="First Set">
<input id="btn2" type="button" value="Second Set">
</form>
<table id="bowlGames">
<thead>
<tr><th>Bowl</th>
    <th>Year</th>
    <th>Winner</th>
    <th>Loser</th>
    <th>Score (Win - Lose)</th>
</tr>
</thead>
<tbody id="bowlData"></tbody>
</table>
```

Example VII-2. Object-based XMLHttpRequest application (continued)

```
</body>
</html>
```

Debugging XMLHttpRequest Code

It's not uncommon for early experimentation with the `XMLHttpRequest` object to be somewhat frustrating because its results are not rendered in the browser until it has successfully done its job. Here are some tips to follow when looking for the source of problems. These tips assume you either have access to a JavaScript debugger (such as the Venkman debugger for Mozilla) or use alerts or other ways of examining values of expressions while script statements are executing.

Inspect the readyState Property After a Request

After the `send( )` method executes, you should be receiving frequent calls to the function handler bound to the `readystatechange` event. If the values never change from 1 or 2, you are probaby having either network or server problems. Verify that the URL you are supplying works in other environments, such as a browser. Also inspect the `status` and/or `statusText` properties of the request object to see if the server is issuing any error messages. Your goal is to obtain a final `status` value of 200.

Inspect the nodeType and childNodes of the responseXML Property

If the server sent the data to your request object, the `responseXML.nodeType` property should evaluate to 9, the `nodeType` value of a `document` node. After that, make sure that the XML data has been received such that the contents of the document are in the correct format. Test for the `responseXML.childNodes.length` property to verify that there is at least one child of the document node. The exact number will depend on the structure of your XML data, but a value of zero means no XML data is in the response. If either one of these tests fails, the most likely culprit is that the server sent the data in a content type that is not `text/xml` or one of the other valid XML content-types. First check the `responseText` property to see if the raw data is received in string form. If the data is there, but it's not showing up as an XML document node, you have content-type issues on the server. Setting the request header as shown in Examples VII-1 and VII-2 may help. Otherwise you will have to check the server configuration to make sure it outputs data for the URL with the correct content-type.

Make Sure Your XML is Well-Formed

It is also imperative that the XML output from your server be well-formed XML. Flaws in well-formedness commonly cause XMLHttpRequest to fail to convert the

output to an XML document node. If you don't have an XML validator among your local tools, search the web for "xml validator" to find free third-party validators that should help you find problems in your output.

REST Versus SOAP

The `XMLHttpRequest` object is capable of interacting with Representational State Transfer (REST) and Simple Object Access Protocol (SOAP), each of which has its zealots. Typically, details of a REST server request are appended to the end of a URL as a search string, and the GET method is used for the transfer. In contrast, SOAP calls are POSTed to the server, and the details must be sent in a string consisting of a well-formed XML document (i.e., not a `document` node object). Your decision about which way to go with the `XMLHttpRequest` object is determined solely by which approach the server uses.

Examples shown earlier are suitable for REST requests. If some of the data submitted to the server comes from form controls modified by users, your script can assemble the search string by reading the form control names and value, formatting them in the *`name1=value1&name2=value2&etc.`* style, and then appending that group to the URL (placing a `?` symbol between the address and search string).

A SOAP submission is called a *message*. Its basic job is to convey one or more commands, each with zero or more parameters to the server. The message is formatted as XML and conforms to a structure that resembles the following pseudo-code:

```
<Envelope>
    <Body>
        <command>
            <parameter>value</parameter>
            <parameter>value</parameter>
        </command>
    </Body>
</Envelope>
```

To demonstrate how to make a SOAP call with XMLHttpRequest, I'll assume that the server delivering the HTML page also has a SOAP service that returns the up-to-the-minute currency conversion rates between two countries. This would allow a client-side script to perform the conversions locally within a single web page. A SOAP message to this service might look like the following:

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">
    <SOAP-ENV:Body>
        <ns1:getCurrencyRate
            xmlns:ns1="urn:xmethods-CurrencyRate"
            SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
            <countryfrom xsi:type="xsd:string">estonia</countryfrom>
```

```
            <countryto xsi:type="xsd:string">australia</countryto>
        </ns1:getCurrencyRate>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Only two small segments of this message—the text nodes inside the `countryfrom` and `countryto` elements—vary from one submission to the next. Therefore, your client-side script need only insert those user-supplied chunks of data (perhaps as values of `select` elements) into an otherwise hard-coded data string.

The following function receives two parameters for the desired country names, assembles the data as a long string (which wraps in the display below), and then passes execution to another function that sends the data via the `XMLHttpRequest` object. Parameters variables are highlighted to show where they are used within the function.

```
function fetchConversionRate(fromCountry, toCountry) {
    var url = 'http://example.com/soap';
    var data = '<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsi="http://www.w3.org/1999/
XMLSchema-instance" xmlns:xsd="http://www.w3.org/1999/XMLSchema"><SOAP-ENV:Body><ns1:
getCurrencyRate xmlns:ns1="urn:xmethods-CurrencyExchange" SOAP-ENV:
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><countryfrom xsi:type="xsd:
string">' + fromCountry + '</countryfrom><countryto xsi:type="xsd:string">' +
toCountry + '</countryto></ns1:getCurrencyRate ></SOAP-ENV:Body></SOAP-ENV:Envelope>
';
    loadSOAP(url, data, recalc);
}
```

To send the message to the server, the XMLHttpRequest code differs slightly from the REST examples shown earlier (shown here in a procedural version, analogous to Example IV-10). The `open( )` method specifies the POST method, and two `setRequestHeader( )` methods prepare the headers. Finally, the `send( )` method includes the XML-as-text data as its parameter, as shown in the following function.

```
var soapReq;
function loadSOAP(url, data, loadHandler) {
    // native object only for demonstration
    soapReq = new XMLHttpRequest( );
    soapReq.open("POST", url);
    soapReq.onreadystatechange = loadHandler;
    soapReq.setRequestHeader("Content-Type","text/xml; charset=utf-8");
    soapReq.setRequestHeader("SOAPAction","");
    soapReq.send(data);
}
```

You also have the option of assembling posted data as an XML document, out of the user's view. For example, if you use XMLHttpRequest to obtain a template for submission (such as the currency conversion SOAP command shown above), you could use DOM parsing to insert the text field values into that unseen XML document. You may then pass that completed XML document as the data parameter of the

request object's `send( )` method—a request object automatically serializes an XML document (i.e., converts it to a string) prior to sending.

After the data returns from the server, you can access it via the same `responseXML` or `responseText` properties of the instance of the request object described earlier.

Using XMLHttpRequest for Other Data Types

Despite the "xml" part of its name, the `XMLHttpRequest` object can retrieve other types of data which can be accessed through the `responseText` property. This includes HTML source code or any other kind of string, such as a string representation of a JavaScript custom object or array (JSON). Just be aware that data such as HTML source code is strictly in string format, which means that you cannot use DOM parsing on it to extract data that way. You could, of course, write that data into a hidden `iframe` element, and then use DOM parsing on the document inside the iframe.

But it is the JavaScript Object Notation possibilities that are more intriguing. Instead of having to script your way through an XML document's DOM, have the server output its results as a string representation of the objects and/or arrays that make the client-side scripting most efficient, such as preset associative arrays that facilitate table iterations or client-side lookups. Converting a JSON string to objects entails the JavaScript `eval( )` global function. Although I'm generally not a fan of this function because of its detrimental effects on performance, occasional one-shot use (as opposed to repeated usage inside a tight repeat loop) has no impact on performance as perceived by the user. Visit *http://www.json.org/js.html* for additional details, including a link to a small open source library that provides robust conversion between objects and strings while thwarting the possibilities of executing "evil" code in the browser.

Without question, the `XMLHttpRequest` object has emerged from its humble, almost hidden, beginnings as an ActiveX control in IE for Windows to become a corner piece of the DHTML jigsaw puzzle. We developers are fortunate that Mozilla, Apple, and Opera have implemented the object across so many browser brands. It invites our imaginations to soar to new heights of application design and user convenience.