The Complete Reference

Part III

Fundamental Client-Side JavaScript

The Complete Reference

Chapter 9

Traditional JavaScript Object Models

An object model defines the interface to the various aspects of the browser and the document that can be manipulated by JavaScript. In JavaScript, two primary object models are employed—a Browser Object Model (BOM) and a Document Object Model (DOM). The BOM provides access to the various characteristics of a browser, such as the browser window itself, the screen characteristics, the browser history, and so on. The DOM, on the other hand, provides access to the contents of the browser window—namely the document and including the various HTML elements, ranging from anchors to images as well as any text that may be enclosed by such elements. Unfortunately, the division between the DOM and the BOM is at times somewhat fuzzy, and the exact document-manipulation capabilities of different browsers' implementations of JavaScript vary significantly. This section begins our exploration of the use of the various aspects of JavaScript object models that are fundamental to the effective use of the language. We'll begin with an exploration of the traditional, though nonstandardized, object models found in the two major browsers. We'll find that many of the ideas of what is related to the browser and what is related to the document are mixed up in these models. The next chapter will begin our discussion of the W3C standard Document Object Model that provides a standard way to manipulate HTML (or even XML) documents for a variety of languages, including JavaScript. Finally, we will explore the various event models supported by the browser vendors. Readers are encouraged to study these next few chapters carefully, as a complete understanding of the evolution and capabilities of the various object models discussed here is fundamental to understanding the applications found in Part IV of the book.

Object Model Overview

An object model is an interface describing the logical structure of an object and the standard ways in which it can be manipulated. Figure 9-1 presents the "big picture" of all the various aspects of JavaScript, including its object models. There are four primary pieces:

1. The core JavaScript language (data types, operators, statements, and so on)
2. The core objects primarily related to data types (**Date**, **String**, **Math**, and so on)
3. The browser objects (**Window**, **Navigator**, **Location**, and so on)
4. The document objects (**Document**, **Form**, **Image**, and so on)

Up until this point we have focused on primarily the first and second aspects of JavaScript. This part of the language is actually fairly consistent between browser types and versions and corresponds to the features defined by the ECMAScript specification (**http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM**). However, the actual objects with which we can manipulate the browser and document do vary. In Figure 9-1 you'll notice that the Browser Object Model (BOM) and Document Object Model (DOM) appear somewhat intermixed. In fact, in older browser versions there really wasn't much of a distinction between the Browser Object Model and the Document Object Model—it was just one big mess!

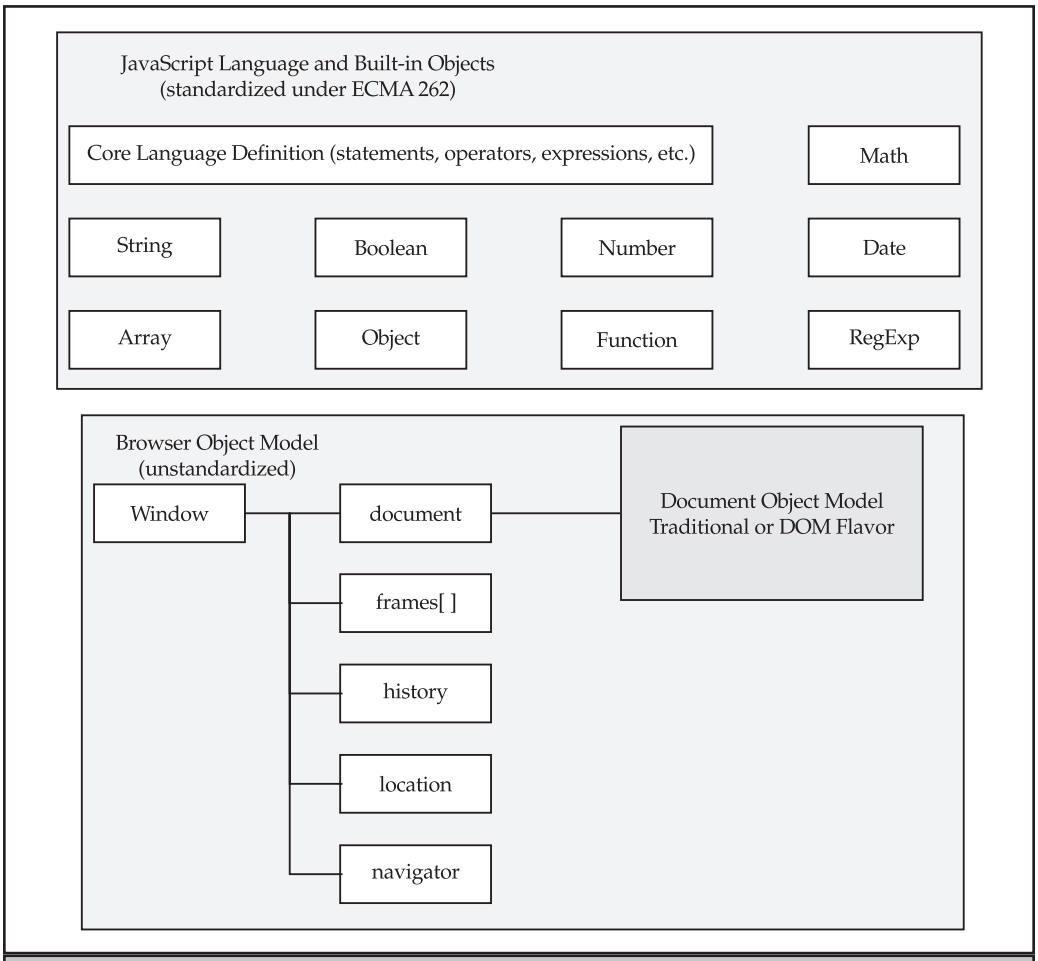

Figure 9-1. *JavaScript: The "Big Picture"*

By studying the history of JavaScript, we can bring some order to the chaos of competing object models. There have been four distinct object models used in JavaScript:

- Traditional JavaScript Object Model (Netscape 2 and Internet Explorer 3)
- Extended Traditional JavaScript Object Model (Netscape 3)
- Dynamic HTML-Flavored Object Models
  - Internet Explorer 4
  - Netscape 4
- Traditional Browser Object Model + Standard DOM (NS6 and IE 5)

We'll look at each of these object models in turn and explain what features as well as problems each introduced. Fortunately, today standards have begun to emerge that have helped straighten this mess out, but it will take some time before JavaScript programmers can safely let go of all browser-specific knowledge they have. Before we get into all that, let's go back to a much simpler time and study the traditional object model that is safe to use in any JavaScript-aware browser.

The Traditional JavaScript Object Model

If you recall the history of JavaScript presented in Chapter 1, one of the primary purposes of the language at first was to check or manipulate the contents of forms before submitting them to server-side programs. Because of these modest goals, the initial JavaScript object model first introduced in Netscape 2 was rather limited, and it focused on the basic features of the browser and document. Figure 9-2 presents the traditional object model, which is pretty similar in both Netscape 2 and Internet Explorer 3.

You might be curious how the various objects shown in Figure 9-2 are related to JavaScript. Well, we've actually used them. For example, **Window** defines the properties and methods associated with a browser window. When we used the JavaScript statement

```
alert("hi");
```

to create a small alert dialog, we actually invoked the **alert()** method of the **Window** object. In fact, we could have just as easily written

```
window.alert("hi");
```

to create the same window. The "window" prefix is generally omitted, because most of the time the interpreter can infer that we are using the current **Window** object. More specifically, the **Window** object is almost always within the scope of your scripts. This means that the interpreter always checks the **Window** object for identifiers (for example, function names) that are used without being explicitly defined in the script.

The containment hierarchy shown in Figure 9-2 should also make sense once you consider a statement like this:

```
window.document.write("<strong>Hi there from JavaScript!</strong>");
```

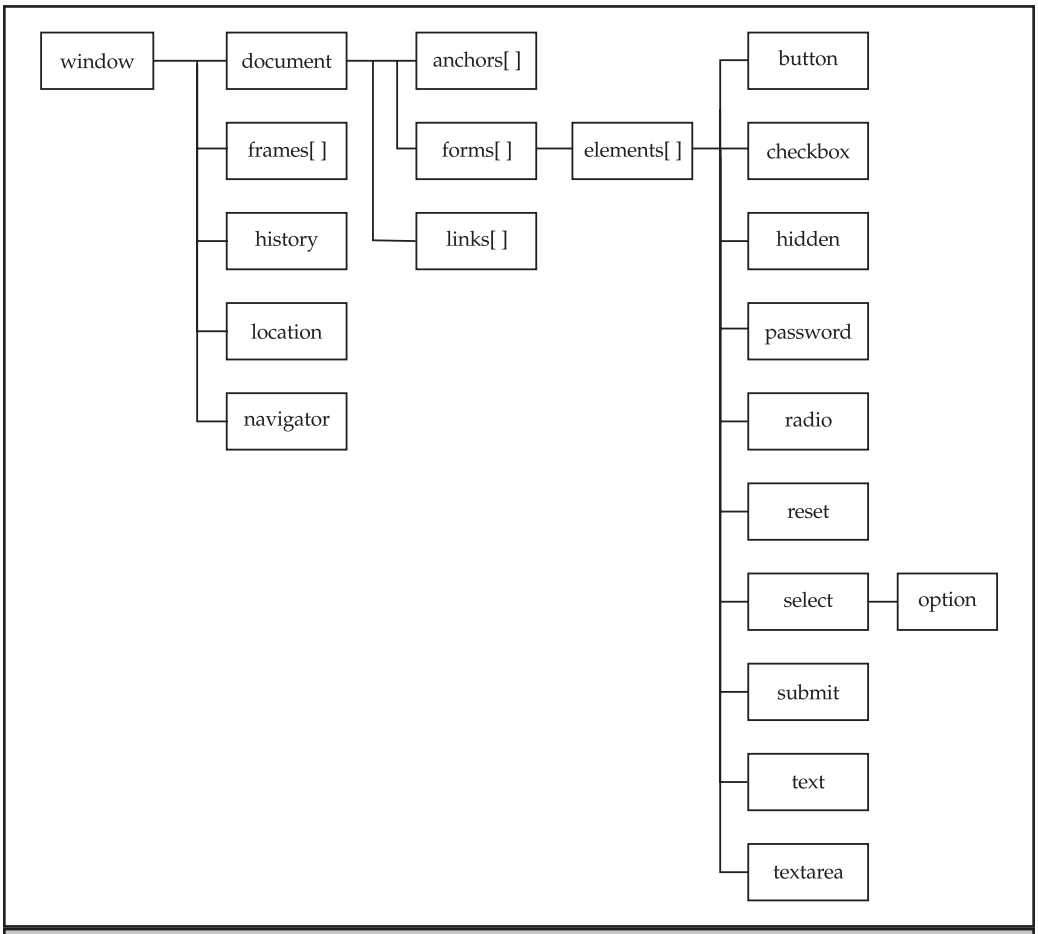

Figure 9-2. *The traditional JavaScript object model*

This should look like the familiar output statement used to write text to an HTML document. Once again, we added in the "prefix window," this time to call attention to the hierarchy, as we have tended to use just **document.write( )** in our examples. You might be curious about what all the various objects shown in Figure 9-2 do, so in Table 9-1 we present an overview of the traditional browser objects. The bulk of the objects are contained within the **Document** object, so we'll look at that more closely now.

Object	Description
Window	The object that relates to the current browser window.
Document	An object that contains the various HTML elements and text fragments that make up a document. In the traditional JavaScript object model, the **Document** object relates roughly to the HTML **<body>** tag.
Frames[ ]	An array of the frames in the page (if it contains any). Each frame in turn references another **Window** object that may also contain more frames.
History	An object that contains the current window's history list, namely the collection of the various URLs visited by the user recently.
Location	An object that contains the current location of the document being viewed in the form of a URL and its constituent pieces such as protocols, host names, and paths.
Navigator	An object that describes the basic characteristics of the browser, notably its type and version.

Table 9-1. *Overview of Core Browser Objects*

The Document Object

The **Document** object provides access to page elements such as anchors, form fields, and links as well as page properties such as background and text color. We will see that the structure of this object varies considerably from browser to browser and from version to version. Table 9-2 and Table 9-3 list those **Document** properties and methods, respectively, that are the "least common denominator" for the earliest browsers providing a document object model (Netscape 2.0 and Internet Explorer 3.0). For the sake of brevity, some details and **Document** properties will be omitted in the following discussion. Complete information about the **Document** properties can be found in Appendix B.

An examination of Tables 9-2 and 9-3 reveals that the early object model was very primitive. The only parts of a document that could be directly accessed were document-wide properties, links, anchors, and forms. There was no support for the manipulation of text or images, no support for applets or embedded objects, and no way to access

Document Property	Description	HTML Relationship	Writeable in NS?	Writeable in IE?
alinkColor	The color of "active" links—by default red	<body alink="color value">	Only in NS6+	Yes
anchors[ ]	Array of anchor objects in the document	 	No	No
bgColor	The page background color	<body bgcolor="color value">	Yes	Yes
cookie	String giving access to the cookies the browser has for the page	n/a	Yes	Yes
fgColor	The color of the document's text	<body text="color value">	Only in NS6+	Yes
forms[ ]	Array containing the form elements in the document	<form>	No	No
lastModified	String containing the date the document was last modified	n/a	No	No
links[ ]	Array of links in the document	linked content	No	No
linkColor	The color of unvisited links—by default blue	<body link="color value">	Only in NS6+	Yes
location	String containing URL of the document (deprecated: use document.URL or the **Location** object instead)	n/a	Only in NS6+	Yes
referrer	String containing URL of the document from which the current document was accessed (broken in IE3 and IE4)	n/a	No	No
title	String containing the document's title	<title>Document Title</title>	Only in NS6+	Only in IE4+
URL	String containing the URL of the document	n/a	No	Yes
vlinkColor	The color of visited links—by default purple or dark blue	<body vlink="color value">	Only in NS6+	Yes

Table 9-2. *Lowest Common Denominator Document Properties*

Method	Description
close()	Closes input stream to the document
open()	Opens the document for input
write()	Writes the argument to the document
writeln()	Writes the arguments to the document followed by a newline

Table 9-3. *Lowest Common Denominator Document Methods*

the presentation properties of most elements. An example showing the various **Document** properties printed for a sample document is presented here:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Traditional Document Object Test</title>
<script language="JavaScript" type="text/javascript">
<!--
function showProps()
{
 var i;

 document.write("<h1 align='center'>Document Object Properties</h1><hr><br>");
 document.write("<h2>Basic Page Properties</h2>");
 document.write("Location = "+document.location + "<br>");
 document.write("URL = " + document.URL + "<br>");
 document.write("Document Title = "+ document.title + "<br>");
 document.write("Document Last Modification Date = " + document.lastModified + "<br>");

 document.write("<h2>Page Colors</h2>");
 document.write("Background Color = " + document.bgColor + "<br>");
 document.write("Text Color = " + document.fgColor + "<br>");
 document.write("Link Color = " + document.linkColor +"<br>");
 document.write("Active Link Color = " + document.alinkColor +"<br>");
```

```
 document.write("Visited Link Color = " + document.vlinkColor + "<br>");
 if (document.links.length > 0)
  {
     document.write("<h2>Links</h2>");
     document.write("# Links = "+ document.links.length + "<br>");
     for (i=0; i < document.links.length; i++)
         document.write("Links["+i+"]=" + document.links[i] + "<br>");
  }
 if (document.anchors.length > 0)
  {
    document.write("<h2>Anchors</h2>");
    document.write("# Anchors = " + document.anchors.length + "<br>");
    for (i=0; i < document.anchors.length; i++)
    document.write("Anchors["+i+"]=" + document.anchors[i] + "<br>");
  }

 if (document.forms.length > 0)
  {
    document.write("<h2>Forms</h2>");
    document.write("# Forms = " + document.forms.length + "<br>");
    for (i=0; i < document.forms.length; i++)
       document.write("Forms["+i+"]=" + document.forms[i].name + "<br>");
  }
}
//-->
</script>
</head>
<body bgcolor="white" text="green" link="red" alink="#ffff00">
<h1 align="center">Test Document</h1>
<hr>
<a href="http://www.pint.com/">Sample link</a>
<a name="anchor1"></a>
<a name="anchor2" href="http://www.javascriptref.com">Sample link 2</a>

<form name="form1"></form>
<form name="form2"></form>
```

```
<hr>
<br><br>
<script>
<!--
//-->
</script>
</body>
</html>
```

An example of the output of the previous example is shown in Figure 9-3.

One thing to note, about this example is that many of the properties will not be set if you do not run this with a document containing forms, links, and so on. Notice the result of the same script on a document with the following simple **<body>** contents:

```
<body>
<h1 align="center">Test 2 Document</h1>
<hr>
<script>
<!--
//-->
</script>
</body>
</html>
```

JavaScript will not instantiate an object for an element that is not present, so notice that you do not have nearly as many properties to view as shown in Figure 9-4. However, you will notice that browsers will tend to define default values for certain types of properties, such as text and link colors, regardless of the presence of HTML elements or attributes.

Although it might not be obvious at first glance, the **Document** object is set up as a *containment hierarchy*. Such a hierarchy is induced by the placement of more specific objects "inside" more general objects. Notice how document-wide properties like link color are available within the **Document** object itself, but to examine form elements one must first access the **forms[]** array found inside of the **Document** object. This sort of containment is characteristic of document object models and is carried to its logical conclusion by later models that present the document as a "tree" reflecting the structure of the original HTML.

Figure 9-3. *Simple Document properties*

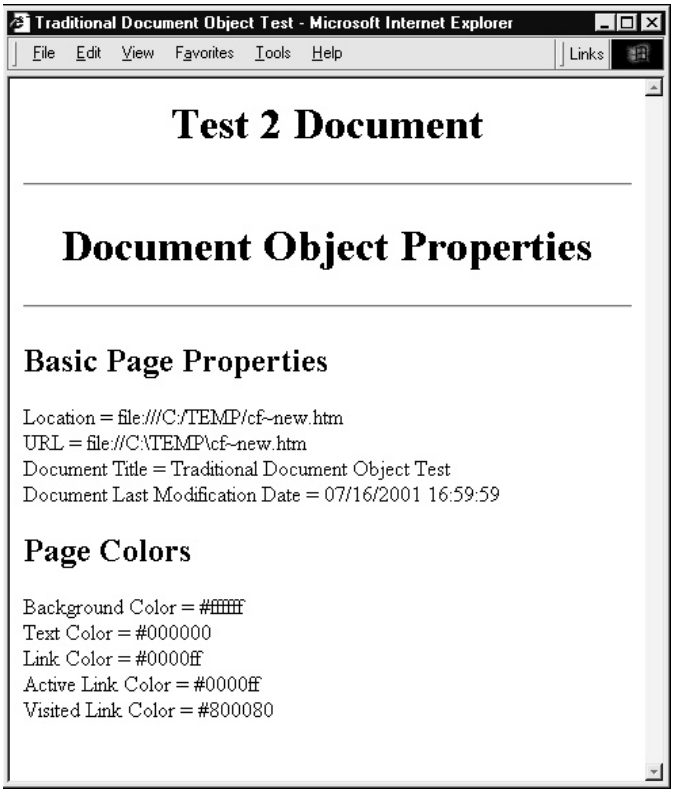

Figure 9-4. *Some Document properties require no HTML elements.*

Accessing Document Elements by Position

When the browser reads an HTML document, JavaScript objects are instantiated for all HTML elements that are scriptable. The number of HTML elements that are scriptable in the first few browsers was fairly limited, but we'll see that in later browsers it is possible to access arbitrary HTML elements. For now, however, let's concentrate on the HTML elements accessible via the traditional JavaScript object model which include anchors, links, forms, and form elements. For example, consider an HTML document like this:

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

```
<html>
<head>
<title>Simple Form</title>
</head>
<body>
<form>
  <input type="text">
</form>
<br><br>
<form>
  <input type="text">
  <br>
  <input type="text">
</form>
</body>
</html>
```


Using the traditional JavaScript object model, we can access the first **<form>** tag using

```
window.document.forms[0]
```

To access the second **<form>** tag, we would use

```
window.document.forms[1]
```

As there are no more **<form>** tags in the document, accessing *window.document.forms[5]* or other values would cause a problem.

If we look again at Figure 9-2, notice that the **forms[ ]** collection also contains an **elements[ ]** collection. This collection contains the various form fields, like text fields, buttons, pull-downs, and so on. Following the basic containment concept, in order to reach the first form element in the first form of the document we would use

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

While this array-based access is straightforward, the major downside is that it relies on the position of the HTML tag in the document. If the tags are moved around, the JavaScript might actually break. A better approach is to rely on the name of the object.

Accessing Document Elements by Name

HTML elements in a Web page really should be named to allow scripting languages to easily read and manipulate them. The basic way to attach a unique identifier to an HTML element under HTML 4 is by using the **id** attribute. The **id** attribute is associated with nearly every HTML element.

The point of the **id** attribute is to bind a unique identifier to the element. To name a particular enclosed bolded piece of text "SuperImportant," you could use the markup shown here:

```
<b id="SuperImportant">This is very important.</b>
```

Just like choosing unique variable names within JavaScript, HTML naming is very important. HTML document authors are encouraged to adopt a consistent naming style and to avoid using potentially confusing names that include the names of HTML elements themselves. For example, *button* does not make a very good name for a form button and will certainly lead to confusion in code and may even interfere with scripting language access.

Before HTML 4, the **name** attribute was used instead of **id** to expose items to scripting. For backward compatibility, the **name** attribute is commonly defined for **<a>**, **<applet>**, **<button>**, **<embed>**, **<form>**, **<frame>**, **<iframe>**, **<img>**, **<input>**, **<object>**, **<map>**, **<select>**, and **<textarea>**. Notice that the occurrence of the **name** attribute corresponds closely to the traditional Browser Object Model.

Note *Both **<meta>** and **<param>** support attributes called **name**, but these have totally different meanings.*

Page developers must be careful to use **name** where necessary to ensure backward compatibility with older browsers. Earlier browsers will not recognize the **id** attribute, so use **name** as well. For example, **<a name="link1" id="link1">** would be interpreted, it is hoped, both by older script-aware browsers as well as by the latest standards-supporting browser.

Note *There are some statements in standards documentation that suggest that it is not a good idea to set the **name** and **id** attributes to the same value, but practice shows this appears to be the only way to ensure backward browser-compatibility.*

To access the form defined by

```
<form name="myform" id="myform">
<input type="text" name="username" id="username">
</form>
```

by name in JavaScript, use either

```
window.document.myform
```

or simply

```
document.myform
```

because the **Window** object can be assumed. The field and its value can be accessed in a similar fashion. To access the text field, use *document.myform.username.*

Accessing Objects Using Associative Arrays

Most of the arrays in the **Document** object are associative. That is, they can be indexed with an integer, as we have seen, or with a string denoting the name of the element you wish to access. The name, as we have also seen, is assigned either with HTML's **name** or **id** attribute for the tag. Of course many older browsers will only recognize the setting of an element's name using the **name** attribute. Consider the following HTML:

```
<form name="myForm">
<input name="user" type="text" value="">
</form>
```

You can access the form as *document.forms["myForm"]* or the **elements[ ]** array of the **Form** object to access the field as *document.forms["myForm"].elements["user"].* Internet Explorer generalizes these associative arrays a bit and calls them *collections*. Collections in IE can be indexed with an integer, with a string, or with the special **item()** method. The **item()** method is used to retrieve a named object from a collection and accepts a string indicating the name of the object to retrieve. For example:

```
document.forms.item("myForm")
```

accesses the previous **Form** object. Internet Explorer collections are discussed in more detail in Chapter 23, but for now just assume that *collection.item("name")* is the same as *collection["name"].*

Event Handlers

The primary way in which scripts respond to user actions is through *event handlers*. An event handler is JavaScript code associated with a particular part of the document and a particular "event." The code is executed if and when the given event occurs at the part of the document to which it is associated. Common events include **Click**, **MouseOver**, and **MouseOut**, which occur when the user clicks, places the mouse over,

or moves the mouse away from a portion of the document. These events are commonly associated with form buttons, form fields, images, and links and are used for rollover buttons and tasks like form field validation. It is important to remember that not every object is capable of handling every event. The events an object can handle are largely a reflection of the way the object is most commonly used.

Setting Event Handlers

You have probably seen event handlers before in HTML. The following simple example shows users an alert box when they click the button:

```
<form name="myForm" id="myForm">
<input name="myButton" type="button" value="Click me" onclick="alert('That
tickles!')">
</form>
```

The **onclick** attribute of the input element binds the given code to the button's **Click** event. Whenever the user clicks the button, the browser sends a **Click** event to the **Button** object, causing it to invoke its **onclick** event handler.

How does the browser know where to find the object's event handler? This is dictated by part of the document object model known as the *event model*. An event model is simply set of interfaces and objects that enable this kind of event handling. In most major browsers, an object's event handlers are accessible as properties of the object itself. So instead of using HTML to bind an event handler to an object, we can do it with pure JavaScript. The following code is equivalent to the previous example:

```
<form name="myForm" id="myForm">
<input name="myButton" type="button" value="Click me">
</form>
<script language="JavaScript" type="text/javascript">
<!--
document.myForm.myButton.onclick = new Function("alert('That tickles!')");
// -->
</script>
```

We define an anonymous function containing the code for the event handler and then set the button's **onclick** property equal to it.

Event Models

There is obviously much more to event handlers than we have described here. Both major browsers implement sophisticated event models that allow applications great flexibility when it comes to events. For example, if you have to define the same event handler for a large number of objects, you can bind the handler once to an object higher up the hierarchy rather than binding it to each child individually. A more complete discussion of event handlers is found in Chapter 11.

Putting It All Together

Now that we have seen all the components of the traditional object model, it is time to show how all the components are used together. As we have seen previously using a **name** or **id** it is fairly easy to reference an occurrence of an HTML element that is exposed in the JavaScript object model. For example, given:

```
<form name="myform" id="myform">
<input type="text" name="username" id="username">
</form>
```

we would use

```
document.myform.username
```

to access the field named *username* in this form. But how do you manipulate that tag's properties? The key to understanding JavaScript's object model is that, generally, an HTML element's attributes are exposed as JavaScript object properties. So, given that a text field in HTML has the basic syntax of:

```
<input type="text" name="unique identifier" id="unique identifier"
            size="number of characters" maxlength="number of characters"
            value="default value">
```

then *document.myform.username.name* references the **name** attribute of a text field from our example, *document.myform.username.size* references its displayed screen size in characters, and so on. The following simple example puts everything together and shows how the content of a form field is accessed and displayed dynamically in an alert window by referencing the fields by **name**.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Meet and Greet</title>
<script language="JavaScript" type="text/javascript">
<!--
function sayHello()
{
 var theirname = document.myform.username.value;
 if (theirname ! = "")
   alert("Hello "+theirname+"!");
```

```
 else
   alert("Don't be shy.");
}
// -->
</script>
</head>
<body>
<form name="myform" id="myform">
<b>What's your name?</b>
<input type="text" name="username" id="username"  size="20"><br><br>
<input type="button" value="Greet" onclick="sayHello()">
</form>
</body>
</html>
```

Aside from reading the contents of an element with JavaScript, it also is possible, in some cases, to update the contents of certain elements, such as form fields. The following code shows how this might be done:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Meet and Greet 2</title>
<script language="JavaScript" type="text/javascript">
<!--
function sayHello()
{
 var theirname = document.myform.username.value;
 if (theirname != "")
   document.myform.response.value="Hello "+theirname+"!";
 else
   document.myform.response.value="Don't be shy.";
}
// -->
</script>
</head>
<body>
<form name="myform" id="myform">
<b>What's your name?</b>
<input type="text" name="username" id="username"  size="20">
<br><br>
<b>Greeting:</b>
```

```
<input type="text" name="response" id="response" size="40">
<br><br>
<input type="button" value="Greet" onclick="sayHello()">
</form>
</body>
</html>
```

One final item to note is that some of the **Document** properties are writeable, while others are not. Notice how the **anchors[ ]**, **forms[ ]**, and **links[ ]** arrays are listed as read-only in Table 9-2. This does not imply that you cannot modify data contained *in* elements of these arrays. Rather, it means that you cannot modify the elements of the arrays themselves. For example, although you can modify data in a particular form, for instance *document.forms[0].userName.value*, you cannot replace the form *forms[0]* with another **Form** object. Because of the containment hierarchy, all interesting information is contained inside each object anyway, so this restriction does not present a problem.

The previous examples simply show how to access elements using an object model. Later on, in Part IV of the book, we'll see how to do something interesting like form checking using these techniques. Now that we understand the basics of using an object model, it is time to take a look at the specific object models supported by the popular Web browsers.

The Object Models

So far, the discussion has focused primarily on the generic features common to all document object models, regardless of browser version. Not surprisingly, every time a new version was released, browser vendors extended the functionality of the **Document** object in various ways. Bugs were fixed, access to a greater portion of the document was added, and the existing functionality was continually improved.

The gradual evolution of document object models is a good thing in the sense that more recent object models allow you to carry out a wider variety of tasks more easily. However it also poses some major problems for Web developers. The biggest issue is that the object models of different browsers evolved in different directions. New proprietary tags were added to facilitate the realization of Dynamic HTML (DHTML), and new, nonstandard means of carrying out various tasks became a part of both Internet Explorer and Netscape. The result is that the brand-new DHTML code a developer writes using the Netscape object model probably will not work in Internet Explorer (and vice versa). The following sections discuss the object models of major browser versions. In particular, we highlight the new features to be found in each and their relevance to common programming tasks.

Netscape 2

The object model of Netscape 2 is that of the basic object model presented earlier in the chapter. It was the first browser to present such an interface to JavaScript, and its

capabilities are limited. With these limitations, the main uses of JavaScript in this browser are form validation and very simple page manipulation, such as printing the last date of modification.

Netscape 3

Netscape 3's **Document** object opened the door for the first primitive DHTML-like applications. It exposes more of the document content to scripts by providing the ability to access embedded objects, applets, plugins, and images. This object model is shown in Figure 9-5, and the major additions to the **Document** object are listed in Table 9-4.

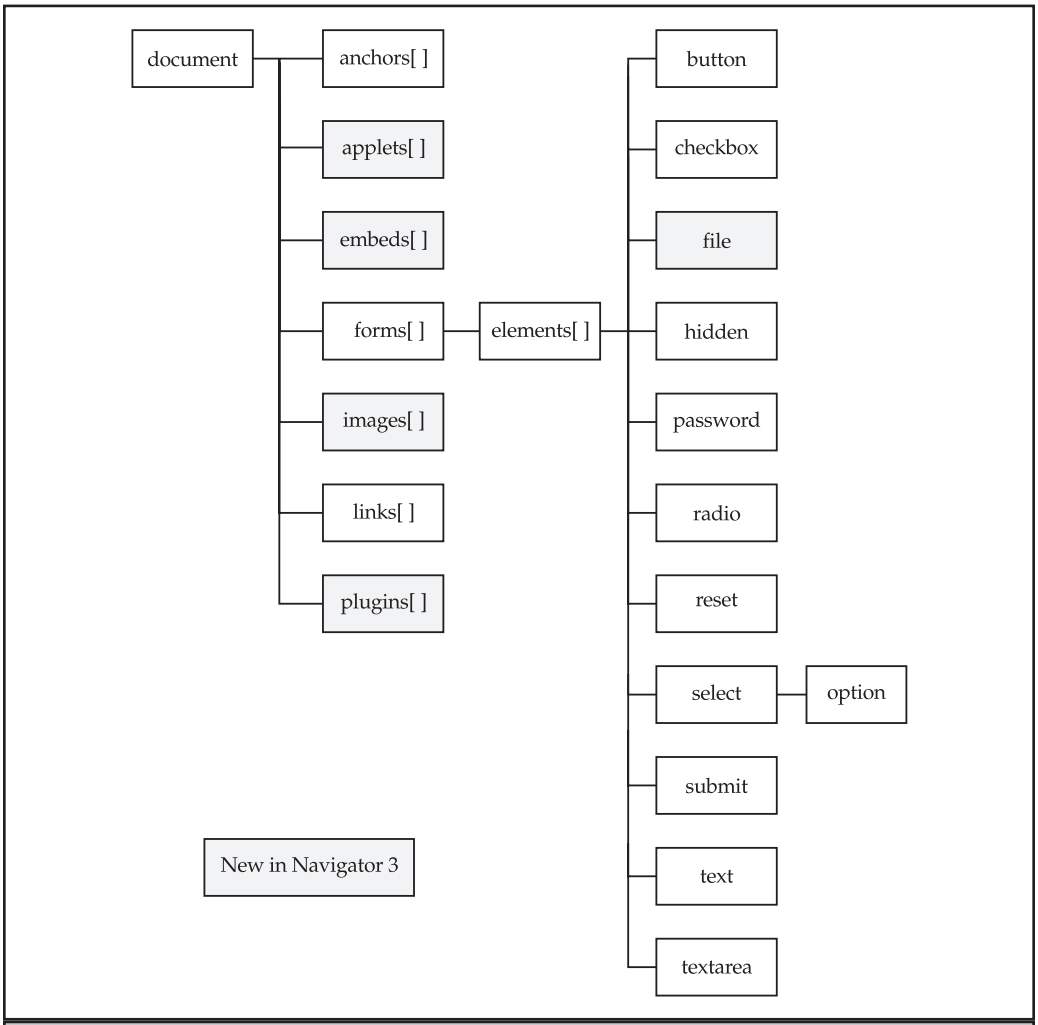

Figure 9-5. *Netscape 3 object model*

Property	Description
applets[ ]	Array of applets (**<applet>** tags) in the document.
domain	String containing the hostname of the Web server from which the document was fetched. Can be changed only to a more general hostname (for instance, www.w3c.org to w3c.org).
embeds[ ]	Array of embedded objects (**<embed>** tags) in the document.
images[ ]	Array of images (**<img>** tags) in the document.
plugins[ ]	Array of plugins installed in the browser.

Table 9-4. *New Document Properties in Netscape 3*

Arguably the most important addition to the **Document** object in this version is the inclusion of the **images[ ]** array. Although most of the properties of each **Image** object are read-only, the **src** property is writeable. This afforded for the first time the ability to change images dynamically in response to user events. A typical application of this capability is *rollover buttons*—buttons that change appearance when the user's mouse is placed over them. Unfortunately, because **Image** objects had no event handlers in this model, rollovers were usually implemented by placing the image inside of an anchor tag; for example:

```
<a href="#"
onmouseover="document.images[0].src='/images/buttonOn.gif'"
onmouseout="document.images[0].src='/images/buttonOff.gif'">
<img src="/images/buttonOff.gif">
</a>
```

Whenever a user's mouse is placed over the link (and thereby over the image), the anchor tag's **onmouseover** event handler swaps images. When the user moves the mouse away from the link, the anchor's **onmouseout** handler swaps the original image back in. Note how the **href** attribute is included in the anchor tag; omitting it may cause the script to fail since the **<a>** tag will not instantiate a **Link** object unless it is properly formed. We'll take an in-depth look at image effects in Chapter 15.

Netscape also made a few additions to its support for forms. The **reset()** method was added to the **Form** object and resets the form upon invocation, regardless of the presence of a reset button. Other, subtler changes were made to the event handlers available for each object. The major addition was the document-wide event handlers **onblur** and **onfocus.** Although these properties are specified in the **<body>** tag of the document, they are actually event handlers for the **Window** object.

More significant additions to the **Window** object were also made. The **History** object was added at this point along with two new properties, **closed** and **opener**. The **window.closed** property indicates if the user has closed that window and helps to avoid using an invalid window. The **window.opener** property lets a "child" window access its "parent," the window containing the JavaScript that created it.

Netscape 4

The document object model of version 4 browsers marks the point at which support for DHTML begins to mature. Outside of swapping images in response to user events, there was little one could do to bring Web pages alive before Netscape 4. Major changes in this version include support for the proprietary **<layer>** tag, additions to Netscape's event model, and the addition of **Style** objects and the means to manipulate them. Figure 9-6 shows the essentials of Netscape 4's object model; the most interesting new properties of the **Document** object are listed in Table 9-5.

DHTML Additions

One of the most important features of Netscape 4's document model is its exposure of stylistic elements to scripts. In fact, the interaction between style sheets and JavaScript under Netscape 4 is so pronounced that Netscape initially termed its style sheet implementation JavaScript Style Sheets (JSSS). Further, if you disable JavaScript under Netscape 4, you will find that style sheets will not work regardless of how they are included.

Under Netscape 4, JavaScript programmers can create or manipulate CSS attributes for every element in the document. It is important to note that this capability is primarily useful for creating or changing stylistic elements *before* the content they apply to is displayed on the screen. Under Netscape 4, the style of most HTML elements cannot be changed once they are displayed. This is very different from the Internet Explorer model discussed later in the chapter.

The new **Document** properties introduced in Netscape 4 that enable manipulation of style are **classes[ ]**, **ids[ ]**, and **tags[ ]**. With JSSS you can access individual tags named using HTML 4's core **id** attribute with the **ids[ ]** property of the **Document** object. The syntax for this is:

document.ids.*idName.propertyName*

where

- *idName* indicates the particular ID you wish to access
- *propertyName* designates the CSS property

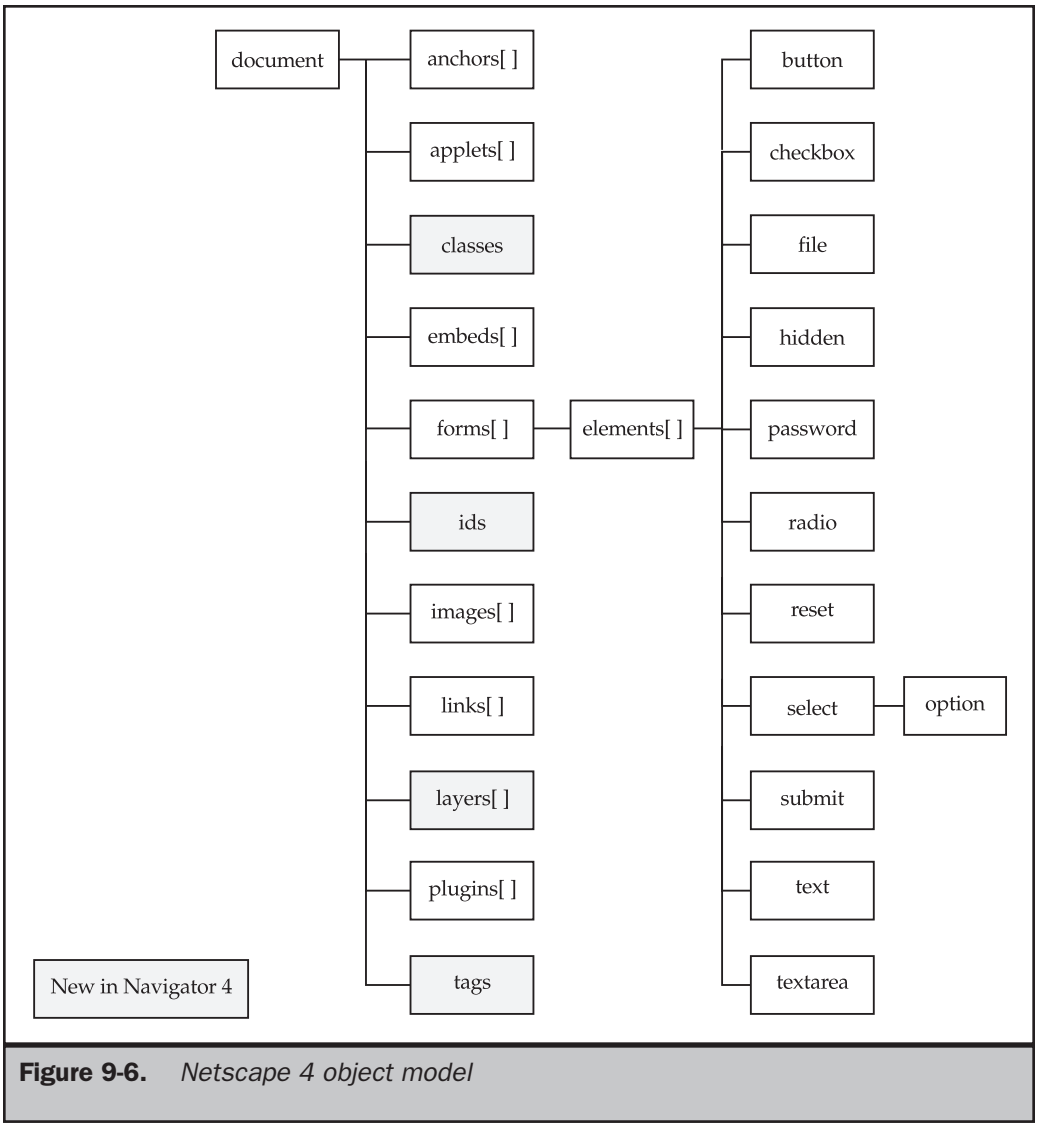

Figure 9-6. *Netscape 4 object model*

For example, to change the size of the element with **id="myHeading"**, you would write

```
document.ids.myHeading.fontSize = "64pt";
```

Property	Description
classes	Creates or accesses CSS style for HTML elements with **class** attributes set.
ids	Creates or accesses CSS style for HTML elements with **id** attributes set.
layers[ ]	Array of layers (**<layer>** tags or positioned ** ** elements) in the document. If indexed by an integer, the layers are ordered from back to front by z-index (where z-index of 0 is the bottommost layer).
tags	Creates or accesses CSS style for arbitrary HTML elements.

Table 9-5. *New Document Properties in Netscape 4*

Similarly, to change a particular group of tags identified with **class** attributes, you use the **classes[ ]** property of Netscape 4's **Document** object using the following syntax:

document.**classes**.*className*.*tagName*.*propertyName*

where

- *className* indicates the name of the class you wish to access
- *tagName* is the name of the specific tag in that class that you are interested in
- *propertyName* indicates the particular CSS property to access

Note that if you wish to access the style information for all tags in a particular class, you can use "all" for the *tagName*.

For example, suppose that you have defined two classes in the **<style>** tag called "important" and "codeListing." You could change the color of list items in the "important" class as:

```
document.classes.important.li.color = "red";
```

or the font of all elements with the "codeListing" class as:

```
document.classes.codeListing.all.fontFamily = "monospace";
```

Lastly, the **tags[ ]** property lets you define or manipulate style globally for a particular HTML element. Its syntax is

document.**tags**.*tagName*.*propertyName*

where

- *tagName* specifies the HTML element
- *propertyName* indicates the particular CSS property to access

For example, to change the text color of **<h1>** to blue, you might write:

```
document.tags.h1.color = "blue";
```

There are quite a few other aspects of JSSS, but the point is not to cover the syntax in detail. In fact, this discussion is primarily historical in nature. Using this syntax is not encouraged at all. It works only in Netscape 4.x generation browsers. Even hardcore Netscape fanatics didn't give much thought to this syntax, since it affects only the presentation of an HTML element before the element is rendered. For example, consider the following simple JSSS example.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Simple Netscape JSSS Example</title>
</head>
<body>
<script language="JavaScript" type="text/javascript">
<!--
 // Danger! Example only works in Netscape 4.x Browsers
  document.tags.h1.color = "blue";
  document.ids.myHeading.fontSize = "64pt";
  document.classes.important.p.backgroundColor = "orange";
  document.classes.important.all.fontStyle = "italic";
// -->
</script>
<h1 class="important">This is an H1 with class important</h1>
<hr>
<p class="important">This is a paragraph with class important</p>
<h2 id="myHeading">This is an H2 with id myHeading</h2>
<p class="important">This is a paragraph with class important</p>
</body>
</html>
```


Note *The output is shown in Figure 9-7; but remember, if we place the script block after the HTML elements the page presentation will not be modified.*

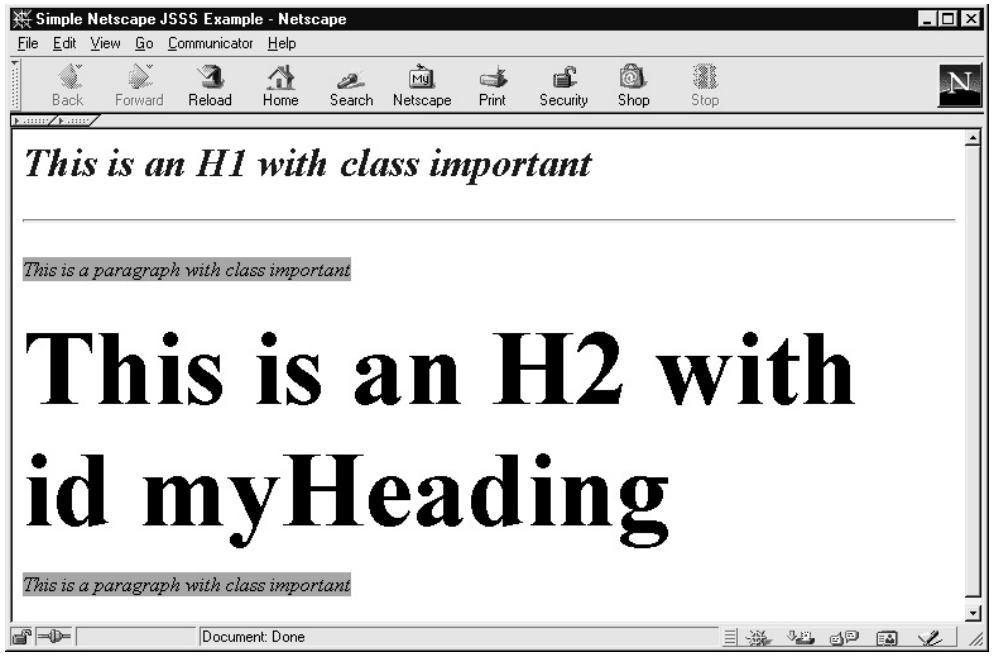

Figure 9-7. *Netscape JSSS affects presentation only before page load.*

One aspect of the Netscape 4.x generation of browsers that is important to understand from a historical perspective is the **Layer** object.

Netscape Layers

Outside of the manipulation of style, Netscape 4 recognizes a new, proprietary HTML tag: **<layer>**. The **<layer>** tag allows you to define content areas that can be precisely positioned, moved, and overlapped as well as rendered hidden, visible, or even transparent. You can write new content to a layer using its **write()** and **writeln()** methods, enabling functionality akin to some of the more advanced features of Internet Explorer. Netscape envisioned this tag as one of the primary foundations upon which DHTML applications would be built. However, the **<layer>** tag never made it into the W3C's HTML standard and was never included by any competing browser vendors. As a result, Netscape abandoned the tag in version 6 of its browser. Thus the **<layer>** tag is available only in Netscape 4.x browsers. It is for this reason that its utility is considered to be fairly limited. Even in the Netscape 4.x browser, the ** ** element combined with Cascading Style Sheets positioning rules provide very similar capabilities.

However, if you must perform some form of positioning in Netscape 4.x, you may be forced to use a **<layer>** tag or at least manipulate a more standard ** ** tag with CSS through the **Layer** object. For example, in the script here, a region was positioned using CSS—but to dynamically modify it under Netscape 4.x, you must use the **Layer** object.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>NS4 Layer Example</title>
<style type="text/css">
<!--
      #div1 { position: absolute;
              top: 200px;
              left: 350px;
              height: 100px;
              width: 100px;
              background-color: orange;}
-->
</style>
</head>
<body>
<h1 align="center">Netscape 4 Layer Example</h1>
<div id="div1">An example of a positioned region</div>
<form>
<input type="button" value="hide"
onclick="document.layers['div1'].visibility='hide'">
<input type="button" value="show"
onclick="document.layers['div1'].visibility='show'">
</form>
</body>
</html>
```


Only the first level of nested layers is available via **document.layers[ ]**, because each layer receives its own **Document** object. To reach a nested layer, you must navigate to the outer layer, then through its **Document** to the nested layer's **layers[ ]** array, and so on. For example, to reach a layer within a layer you might write:

```
var nestedLayer = document.layers[0].document.layers[0].document;
```

Although the use of layers will eventually die off, layers do provide a good way to detect version 4 Netscape browsers. Under Netscape 4, accessing **document.layers** will

return the **layers[ ]** array. Under any other browser, accessing **document.layers** will return **undefined**, so you can conveniently check for Netscape 4 like this:

```
if (document.layers) { /*  do something Netscape specific  */ }
```

Window Additions

A tremendous number of new properties were added to the **Window** object in Netscape 4. Included are properties that give information about screen size, height, and width; properties that give information about the configuration of various toolbars in the user's browser; base-64 encoding and decoding methods; methods to move, resize, and offset the screen; and methods to simulate the click of the browser's forward and back buttons.

Event Model Additions

Aside from the addition of new mouse and keyboard event handlers, such as **onmouseup/down** and **onkeyup/down**, Netscape fleshed out its event model in several significant ways. Events in this browser begin at the top of the object hierarchy and "trickle down" to the object upon which they are acting. This traversal allows **Window** and **Document** objects (including **Layer** objects) to perform *captures* that halt or alter the downward "flow" of an event. Event capturing is achieved with the **captureEvents()** method and permits a captured event to be dealt with at the "higher" level—modified, redirected, or simply passed through to continue on its way down the hierarchy. This feature is used to simplify programming pages where numerous lower-level objects like form fields all require the same event handlers. A detailed discussion of Netscape 4 event handling can be found in Chapter 11.

Netscape 6

The release of Netscape 6 marks a new era for Netscape browsers. The main emphasis of this browser is standards compliance, a refreshing change from the ad hoc proprietary document object models of the past. It is backwards compatible with the so-called DOM Level 0, the W3C's DOM standard that incorporates many of the widespread features of older document object models, in particular that of Netscape 3. However, it also implements DOM Level 1 and parts of DOM Level 2, the W3C's object models for standard HTML, XML, CSS, and events. These standard models differ in significant ways from older models and are covered in detail in the next chapter.

Support for nearly all of the proprietary extensions supported by Netscape 4, most notably the **<layer>** tag and corresponding JavaScript object, have been dropped in Netscape 6. This breaks the paradigm that allowed developers to program for older browser versions knowing that such code would be supported by newer versions. Like many aspects of document models, this is both good and bad; older code may not work in Netscape 6, but future code written for this browser will have a solid standards foundation.

Internet Explorer 3

The object model of IE3 is the basic "lowest common denominator" object model presented at the beginning of this chapter. It includes several "extra" properties in the **Document** object not included in Netscape 2, for example the **frames[ ]** array, but for the most part it corresponds closely to the model of Netscape 2. The Internet Explorer 3 object model is shown in Figure 9-8.

For the short period of time when Netscape 2 and IE3 coexisted as the latest versions of the respective browsers, object models were in a comfortable state of unity. It wouldn't last long.

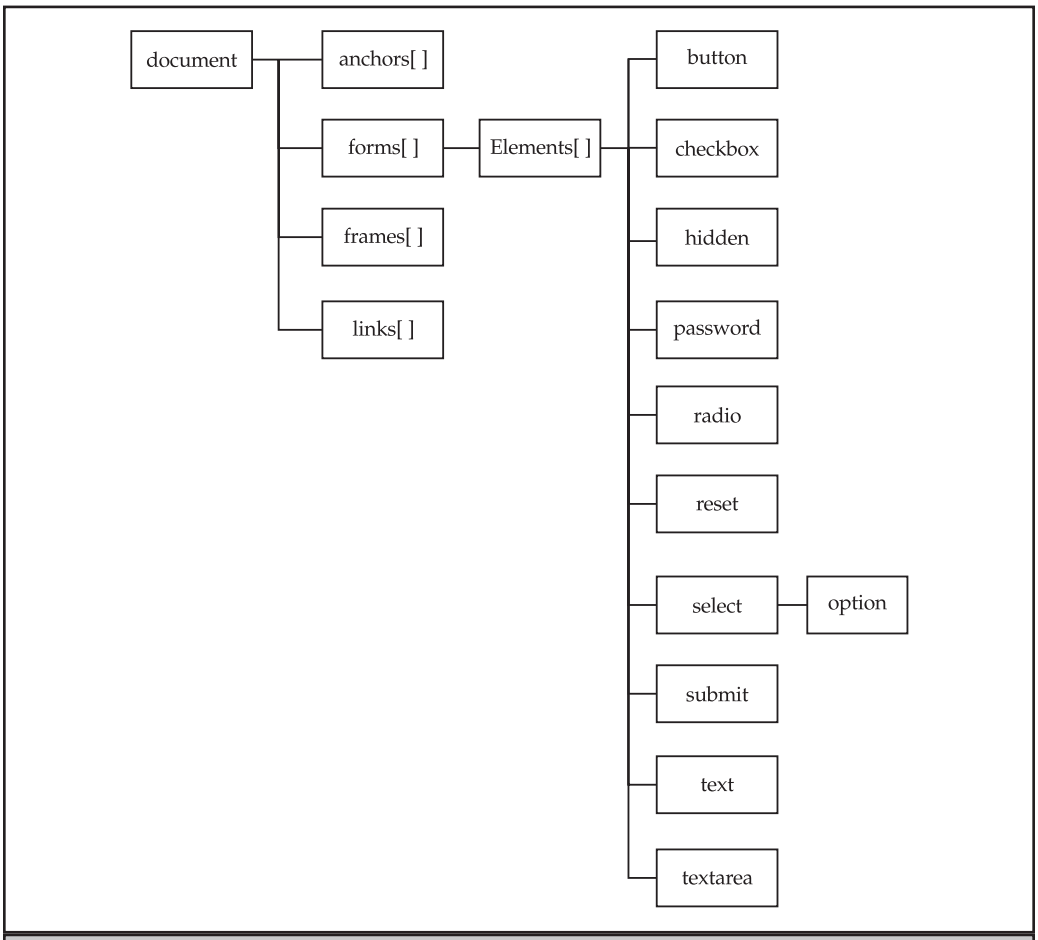

Figure 9-8. *Internet Explorer 3 object model mimics Netscape 2.*

Internet Explorer 4

Like version 4 of Netscape's browser, IE4 lays the foundations for DHTML applications by exposing much more of the page to JavaScript. In fact, it goes much further than Netscape 4 by representing *every* HTML element as an object. Unfortunately, it does so in a manner incompatible with Netscape 4's object model. The basic object model of Internet Explorer 4 is shown in Figure 9-9.

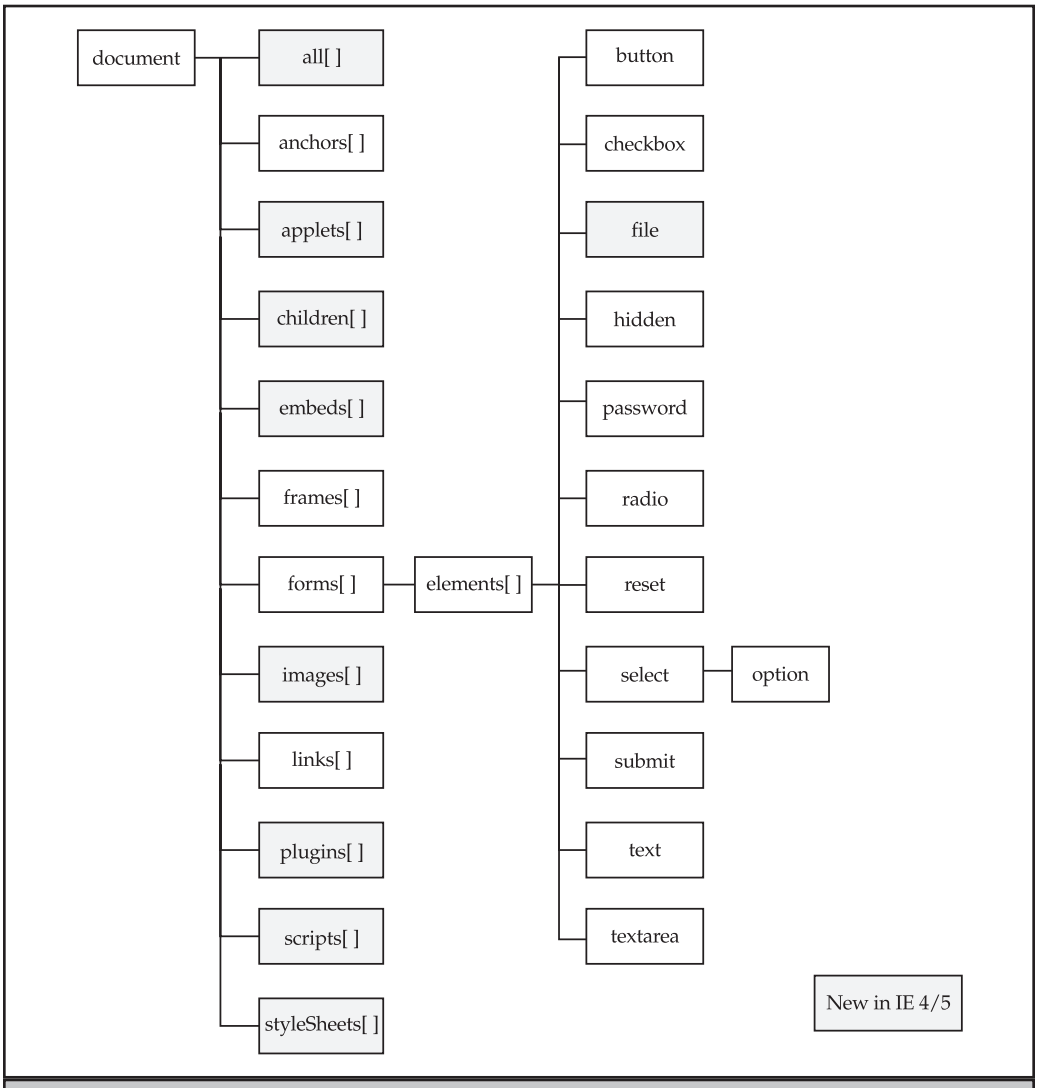

Figure 9-9. *Internet Explorer 4 object model*

Inspection of Figure 9-9 reveals that IE4 supports the basic object model of Netscape 2 and IE3, plus most of the features of Netscape 3 as well as many of its own features. This suggests that code can be written for the set of properties that IE4 and Netscape 3 have in common without too many problems. Table 9-6 lists some important new properties found in IE4. You will notice that Figure 9-9 and Table 9-6 show that IE4 also implements new **Document** object features radically different from those present in Netscape 4. It is in version 4 of the two major browsers where the object models begin their radical divergence.

DHTML Additions

One of the most important new JavaScript features introduced in IE4 is the **document.all[ ]** collection. This array provides access to every element in the document. It can be indexed in a variety of ways and returns a collection of objects matching the index, **id** or **name** attribute provided; for example:

```
// sets variable to the fourth element in the document
var theElement = document.all[3];

// finds tag with id or name = myHeading
var myHeading = document.all["myHeading"];

// alternative way to find tag with id or name = myHeading
var myHeading = document.all.item("myHeading");

// returns array of all <em> tags
var allEm = document.all.tags("EM");
```


Property	Description
all[ ]	Array of all HTML tags in the document
applets[ ]	Array of all applets (**<applet>** tags) in the document
children[ ]	Array of all child elements of the object
embeds[ ]	Array of embedded objects (**<embed>** tags) in the document
images[ ]	Array of images (**<img>** tags) in the document
scripts[ ]	Array of scripts (**<script>** tags) in the document
styleSheets[ ]	Array of **Style** objects (**<style>** tags) in the document

Table 9-6. *New Document Properties in Internet Explorer 4*

As you can see there are many ways to access the elements of a page, but, regardless of the method used, the primary effect of the **document.all[ ]** collection is that it flattens the document object hierarchy to allow quick and easy access to any portion of an HTML document. The following simple example shows that Internet Explorer truly does expose all the elements in a page; its result is shown in Figure 9-10.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Document.All Example</title>
</head>
<body>
<h1>Example Heading</h1>
<hr>
<p>This is a <em>paragraph</em>.  It is only a <em>paragraph.</em></p>
<p>Yet another <em>paragraph.</em></p>
<p>This final <em>paragraph</em> has <em id="special">special emphasis.</em></p>
<hr>
<script language="JavaScript" type="text/javascript">
<!--
 var i, origLength;
 origLength = document.all.length;
 document.write('document.all.length='+origLength+"<br>");
 for (i = 0; i < origLength; i++)
   {
   document.write("document.all["+i+"]="+document.all[i].tagName+"<br>");
   }
// -->
</script>
</body>
</html>
```

Note

The previous example will result in an endless loop if you do not use the origLength *variable and rely on the* ***document.all.length*** *as your loop check. The reason is that the number of elements in the* ***document.all[ ]*** *collection will grow every time you output the element you are checking!*

Similar to the trick to detect Netscape 4 through the existence of the **Layer** object, many programmers rely on the existence of **document.all** to detect Internet Explorer browsers and often rely on statements like:

```
if (document.all) { /*  do something IE specific  */ }
```

to deal with browser-specific code.

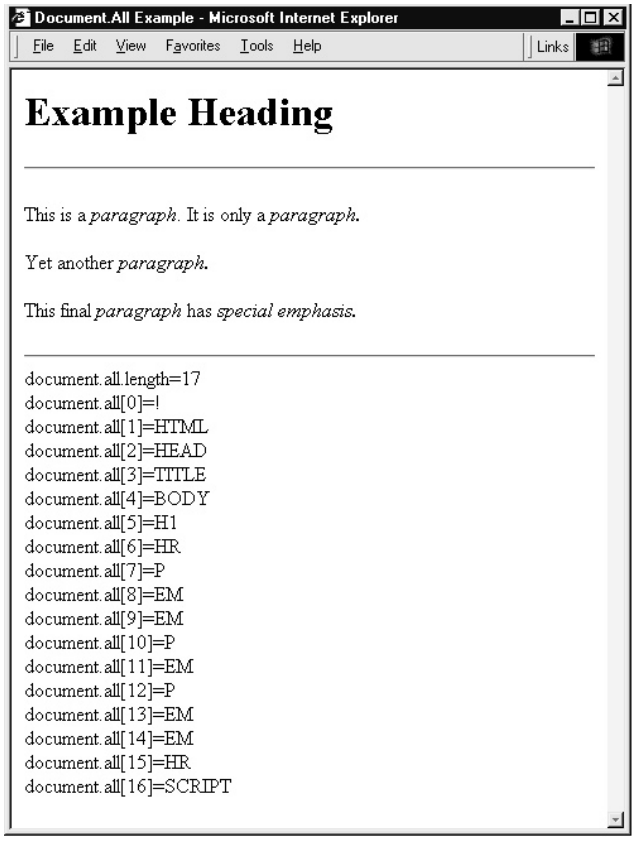

Figure 9-10. *IE's document.all[ ] collection exposes all document elements.*

Once a particular element has been referenced using the **document.all** syntax, you will find a variety of properties and methods associated with it, including the **all** property itself, which references any tags enclosed within the returned tag. Tables 9-7 and 9-8 show some of the more interesting, but certainly not all of these new properties and methods. Note that inline elements will not have certain properties (like **innerHTML**) because by definition their tags cannot enclose any other content.

If Tables 9-7 and 9-8 seem overwhelming, do not worry. At this point you are not expected to fully understand each of these properties and methods. Rather, we list them to illustrate just how far the Netscape and Internet Explorer object models diverged in a very short period of time.

Examination of the new features available in IE4 should reveal that this is the first browser where real DHTML is possible. It provides the means to manipulate style dynamically (after the page has loaded, unlike with Netscape 4) and to insert,

Property	Description
all[ ]	Collection of all elements contained by the object.
children[ ]	Collection of elements that are direct descendents of the object.
className	String containing the CSS class of the object.
innerHTML	String containing the HTML content enclosed by, but not including, the object's tags. This property is writeable for most HTML elements.
innerText	String containing the text content enclosed by the object's tags. This property is writeable for most HTML elements.
outerHTML	String containing the HTML content of the element, including its start and end tags. This property is writeable for most HTML elements.
outerText	String containing the outer text content of the element. This property is writeable for most HTML elements.
parentElement	Reference to the object's parent in the object hierarchy.
style	**Style** object containing CSS properties of the object.
tagName	String containing the name of the HTML tag associated with the object.

Table 9-7. *Some New Properties for Document Model Objects in IE4*

Method	Description
click()	Simulates clicking the object, causing the **onclick** event handler to fire
getAttribute()	Retrieves the argument HTML attribute for the element
insertAdjacentHTML()	Allows the insertion of HTML before, after, or inside the element
insertAdjacentText()	Allows the insertion of text before, after, or inside the element
removeAttribute()	Deletes the argument HTML attribute from the element
setAttribute()	Sets the argument HTML attribute for the element

Table 9-8. *Some New Methods for Document Model Objects in IE4*

modify, and delete arbitrary HTML and text. For the first time, JavaScript can manipulate the structure of the document, changing content and presentation of all aspects of the page as desired. The following example illustrates this feature using Internet Explorer syntax:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Document.All Example #2</title>
</head>
<body>
<!-- Only works in Internet Explorer 4 and greater -->
<h1 id="heading1" align="center">DHTML Fun!!!</h1>

<form name="testform" id="testform">
<br><br>
<input type="button" value="Align Left"
onclick="document.all['heading1'].align='left'">
<input type="button" value="Align Center"
onclick="document.all['heading1'].align='center'">
<input type="button" value="Align Right"
onclick="document.all['heading1'].align='right'">
<br><br>
<input type="button" value="Bigger"
onclick="document.all['heading1'].style.fontSize='larger'">
<input type="button" value="Smaller"
onclick="document.all['heading1'].style.fontSize='smaller'">
<br><br>
<input type="button" value="Red"
onclick="document.all['heading1'].style.color='red'">
<input type="button" value="Blue"
onclick="document.all['heading1'].style.color='blue'">
<input type="button" value="Black"
onclick="document.all['heading1'].style.color='black'">
<br><br>
<input type="text" name="userText" id="userText" size="30">
<input type="button" value="Change Text"
onclick="document.all['heading1'].innerText=document.testform.userText.value"><br>
</form>
</body>
</html>
```


As this short introduction demonstrates, there are tremendous possibilities for DHTML in IE4. The examples given here barely scratch the surface of IE's powerful document object model, though they do show how a highly structured DOM might be presented to scripts. Notice how the interface represents each element as a mutable object and that each object

has well-defined parent-child inclusion properties. These characteristics foreshadow many of the features of the W3C DOM discussed in the next chapter.

Event Model Additions

IE4 implements an "opposite" event model from that of Netscape 4. Events begin at the bottom of the hierarchy at the object where they occur and "bubble up" through parent objects to the **Window**. Any object along the path from the origin of the event to the "top" of the hierarchy can intercept an event, process it, and cancel it or pass it along up the tree. To complicate matters, the properties of IE4's **Event** object are different from those of Netscape's. We'll see these differences in great detail in Chapter 11.

Internet Explorer 5, 5.5, and 6

The document object model of Internet Explorer 5.x is very similar to that of IE4. New features include an explosive rise in the number of properties and methods available in the objects of the document model and proprietary enhancements allowing the development of reusable DHTML components.

Many of the additions to objects of the document model implement portions of the W3C DOM. These features are significantly less complete than those found in Netscape 6, but they are a step in the right direction. Microsoft has also included numerous properties that build on the existing IE4 object model, making it much more robust and capable of more powerful document manipulation. Also greatly increasing in number are the event handlers added in IE5. As of IE5.5, the browser supports almost 40 different events, ranging from specific mouse and keyboard actions to editing events such as cutting and pasting.

IE5 supports two new features called *DHTML Behaviors* and *HTML Applications*. DHTML Behaviors allow programmers to define reusable DHTML components that can be applied to arbitrary elements. HTML Applications (HTA's) are HTML documents that act more like a real programs than web applications. These features are not yet in widespread use, but might make their way into future W3C standards. We'll discuss both of these technologies in Chapter 23 where we cover proprietary Internet Explorer features.

Internet Explorer 5.5 and 6 continue Microsoft's trend of adding features that work only in its browsers, including new behaviors, new forms of popup windows, scrollbar changes, and so on. However, Internet Explorer 5.5 continues to improve its DOM support, and the pre-release versions of Internet Explorer 6 available at the time of this book's writing suggest that IE6 will be totally CSS1- and DOM1-compliant, but that developers must "switch on" the "standards-compliant mode" by including a valid **<DOCTYPE>**. The browser will be backwards compatible with previous IE document object models in order to preserve the functionality of old code. The question of whether or not further standards will be implemented in IE6 is still up in the air.

Opera, Mozilla, Konqueror, and Other Browsers

Although rarely considered by some Web developers, there are some other browsers that have a small but loyal following in many tech-savvy circles. Most third-party browsers are "strict standards" implementations, meaning that they implement W3C and ECMA standards and ignore most of the proprietary object models of Internet Explorer and Netscape. Some provide support for the basic Netscape 2 and IE3 object models, but most focus their development efforts on the W3C standards. If the demographic for your Web site includes users likely to use third-party browsers (for example, Linux users), it might be a good idea to avoid IE- and Netscape-specific features and use the W3C DOM instead.

The Nightmare of Cross-Browser Object Support

The common framework of the **Document** object shared by Internet Explorer and Netscape dates back to 1996. It might be hard to believe, but in the intervening years there has been very little improvement to the parts of the document object model the major browsers have in common.

As a result, when faced with a non-trivial JavaScript task, Web developers have become accustomed to writing two separate scripts, one for Internet Explorer 4+ and one for Netscape 4+. The IE version usually utilizes the extensive style manipulation capabilities of IE4, often relying on the **document.all[ ]** collection, while the Netscape version relies heavily on the **<layer>** tag and associated **document.layers[ ]** array for similar functionality. Because the object models evolved in such incompatible directions, this process can be tiresome and susceptible to errors and oversights. If the developer does not take care in verifying support for particular features and testing applications with a variety of browsers, he or she might be forced to rewrite pages of code.

Continuing down this road will only make the situation worse. If browser vendors continue to make proprietary extensions document objects, the number of different object models developers must support will continue to increase. How long can the average developer be expected to accommodate the myriad of ways to perform seemingly simple tasks like changing text color, replacing pieces of text, and interacting with browser plugins?

Aside from the tedious nature of writing each piece of JavaScript code multiple times for different browsers, there are other problems with traditional document object models. One major question is how scripts will interact with newer technologies like XML, given the diverse and heterogeneous nature of the traditional models. Are browser wars again going to force developers into maintaining a different piece of JavaScript code for each proprietary object model? Are we to be stuck with a fixed, browser-based interface to each object model regardless of application requirements such as memory footprint, speed, and presentation on alternative media?

It should be clear that the situation with traditional object models is less than optimal. With many developers and users frustrated with incompatibilities in object and events models, the Web is ripe for a change.

A Solution?

Such a change has been proffered by the standards put together by the World Wide Web Consortium in the form of its Document Object Model. The W3C gives a succinct description of the DOM: "It is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents, both HTML and XML." This is music to Web developers' ears. Combined with the standardized ways of describing presentation characteristics defined by CSS, the DOM promises a uniform paradigm for creating interactive documents not only for the Web, but for a variety of offline business and technical endeavors as well.

Progress towards realization of this standard is already being made. Netscape has made a firm commitment to implementation of standards, a commitment it is already making good on with Netscape 6. Netscape is also actively supporting several Open Source projects aimed at bringing standards compliance to the browser community. In the year 2001, the Mozilla project is nearing the completion of developing its fully featured standards-based browser. Microsoft is also coming around, promising more DOM features than ever in version 6 of Internet Explorer.

It is interesting to observe that if browser vendors wholeheartedly embrace the W3C DOM, the document models will have come full circle. In the mid-nineties, there was a brief period of time when access to the document was uniform across both major browsers. Version 4 of the browsers marked the beginning of the radical divergence of object models, and there has been little compatibility since. If browser vendors continue to support the W3C DOM, there might be a point in the future when developers have access to a powerful, robust, and standardized interface for the manipulation of structured documents.

Summary

This chapter gives a basic introduction to the traditional document object models. The **Document** object is structured as a containment hierarchy and accessed by "navigating" through general objects to those that are more specific. The most useful **Document** properties are found in associative arrays, like **images[ ]**, which can be indexed by an integer or name when an element is named using an HTML tag's **name** or **id** attribute. Event handlers were introduced as a means to react to user events and may be set with JavaScript or traditional HTML. The main portion of this chapter introduced the specific document object nodels of the major browsers. Netscape 2 and Internet Explorer 3 implement the most basic object model upon which other models are built. Netscape 3 added a few useful new properties, while the bulk of the changes occur in Netscape 4.

Netscape 6 was introduced as a standards-based browser not necessarily supporting all portions of previous models. We saw that Internet Explorer 4 introduced some powerful DHTML applications by exposing all portions of the document to scripts. The IE4 object model was further extended by IE5, and further extensions are likely in IE6. In the course of the chapter, the divergent and incompatible nature of the different object models was stressed, and near the end a possible solution, the W3C DOM, was proposed. The next chapter explains the details of the DOM and why it might revolutionize the way scripts manipulate documents.