PART

Fundamental Client-Side JavaScript

CHAPTER 9

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, there are a variety of object models based upon browser type and version, but in general we see two primary object models employed—a Browser Object Model and a Document Object Model (DOM). The Browser Object Model 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 including the various (X)HTML elements, CSS properties, and any text items.

While it would seem clear, the unfortunate reality is that the division between the DOM and the Browser Object Model is at times somewhat fuzzy and the exact document manipulation capabilities of a particular browser's implementation of JavaScript vary significantly. This section starts our exploration of the use of the various aspects of JavaScript object models that are fundamental to the proper use of the language. We begin this chapter with an exploration of JavaScript's initial object model and then examine the various additions made to the object model by browser vendors. This apparent history lesson will uncover the significant problems with the "DHTML" object models introduced by the browser vendors and still used by many of today's JavaScript programmers and will motivate the rise of the standard DOM model promoted by the W3C, which is covered in the following chapter.

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 various aspects of JavaScript including its object models. We see four primary pieces:

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

Up until this point we have focused on primarily the first and second aspects of JavaScript. This part of the language is actual fairly consistent between browser types

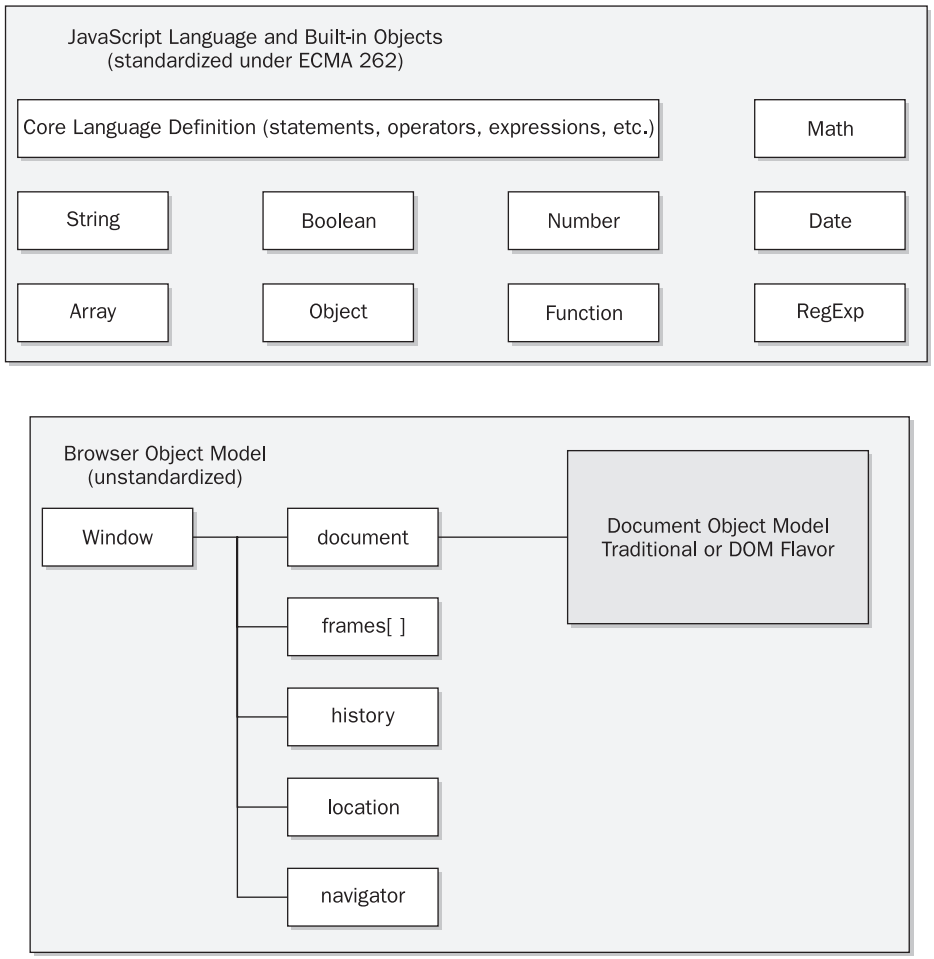

FIGURE 9-1 The "big picture"

and versions, and corresponds to the features defined by the ECMAScript specification (**http://www.ecma-international.org/publications/standards/Ecma-262.htm**). However, the actual objects with which we can manipulate the browser and document do vary. In fact, in Figure 9-1 you'll notice that it appears that the Browser Object Model (BOM) and Document Object Model (DOM) are somewhat intermixed. In previous versions of the browser there really wasn't much of a distinction between the Browser Object Model and the Document Object Model—it was just one big mess.

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, including:

1) Traditional JavaScript object model (Netscape 2 and Internet Explorer 3)

2) Extended JavaScript object model (Netscape 3)—basis of DOM Level 0

3) Dynamic HTML flavored object models
   a. Internet Explorer 4.*x* and up
   b. Netscape 4.*x* only
4) Extended Browser Object Model + Standard DOM (modern browsers)

We'll look at each of these object models in turn and explain what features, as well as problems, that each introduced. Fortunately, standards have emerged that have helped to 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 first object model used by JavaScript, which is safe to use in any JavaScript-aware browser.

The Initial JavaScript Object Model

If you recall the history of JavaScript presented in Chapter 1, the primary purpose 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 focused on the basic features of the browser and document. Figure 9-2 presents JavaScript's initial object model that is pretty similar between 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 have used the JavaScript statement:

L 9-1
```
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:

L 9-2
```
window.alert("hi");
```

to create the same window. Most of the time because we can infer that we are using the current **Window** object, it is generally omitted.

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

L 9-3
```
window.document.write("<strong>Hi there from JavaScript!</strong>");
```

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 show the hierarchy, as we 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 a brief overview of the traditional browser object. As you can see, the bulk of the objects are contained within the **Document** object, so we'll look at that one more closely now.

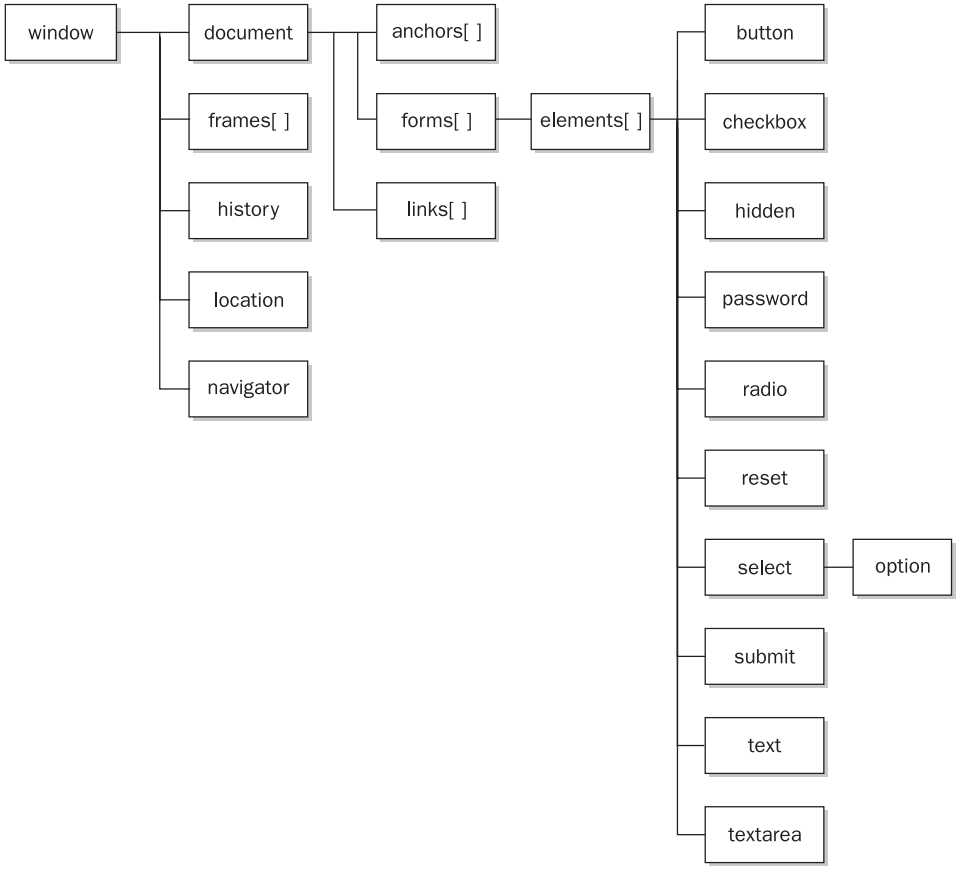

FIGURE 9-2 The initial JavaScript object model

Object	Description
Window	The object that relates to the current browser window.
Document	An object that contains the various (X)HTML elements and text fragments that make up a document. In the traditional JavaScript object model, the **Document** object relates roughly to the **<body>** tag.
Frames[]	An array of the frames if the **Window** object 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	Contains the current location of the document being viewed in the form of a URL and its constituent pieces.
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. Tables 9-2 and 9-3 list those **Document** properties and methods, respectively, that are the "least common denominator" and available since the very first JavaScript-aware browsers. 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.

Document Property	Description	HTML Relationship
alinkColor	The color of "active" links—by default, red	<body alink="color value">
anchors[]	Array of anchor objects in the document	
bgColor	The page background color	<body bgcolor="color value">
cookie	String giving access to the page's cookies	N/A
fgColor	The color of the document's text	<body text="color value">
forms[]	Array containing the form elements in the document	<form>
lastModified	String containing the date the document was last modified	N/A
links[]	Array of links in the document	linked content
linkColor	The unvisited color of links—by default, blue	<body link="color value">
location	String containing URL of the document. (Deprecated.) Use **document.URL** or **Location** object instead.	N/A
referrer	String containing URL of the document from which the current document was accessed. (Broken in IE3 and IE4)	N/A
Title	String containing the document's title	<title>Document Title</title>
URL	String containing the URL of the document	N/A
vlinkColor	The color of visited links—by default, purple	<body vlink="color value">

TABLE 9-2 Lowest Common Denominator Document Properties

NOTE *The **document.referrer** attribute is spelled correctly despite the actual misspelling of the HTTP referer header.*

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

Examination of Tables 9-2 and 9-3 reveals that the early Document Object Model was very primitive. In fact, the only parts of a document that can be directly accessed are document-wide properties, links, anchors, and forms. There is no support for the manipulation of text or images, no support for applets or embedded objects, and no way to access the presentation properties of most elements. We'll see all these capabilities are presented later but first let's focus on the most basic ideas. The following example shows the various document properties printed for a sample document.

L 9-4

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Traditional Document Object Test</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script 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" action="#" method="get"></form>
<form name="form2" action="#" method="get"></form>
<hr />
<br /><br />
<script type="text/javascript">
<!--
  // Needs to be at the bottom of the page
  showProps();
//-->
</script>
</body>
</html>
```

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

One thing to note with this example, however, is the fact 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 shown in Figure 9-4.

L 9-5

```
<body>
<h1 align="center">Test 2 Document</h1>
<hr />
<script type="text/javascript">
<!--
  // Needs to be at the bottom of the page
  showProps();
//-->
</script>
</body>
</html>
```

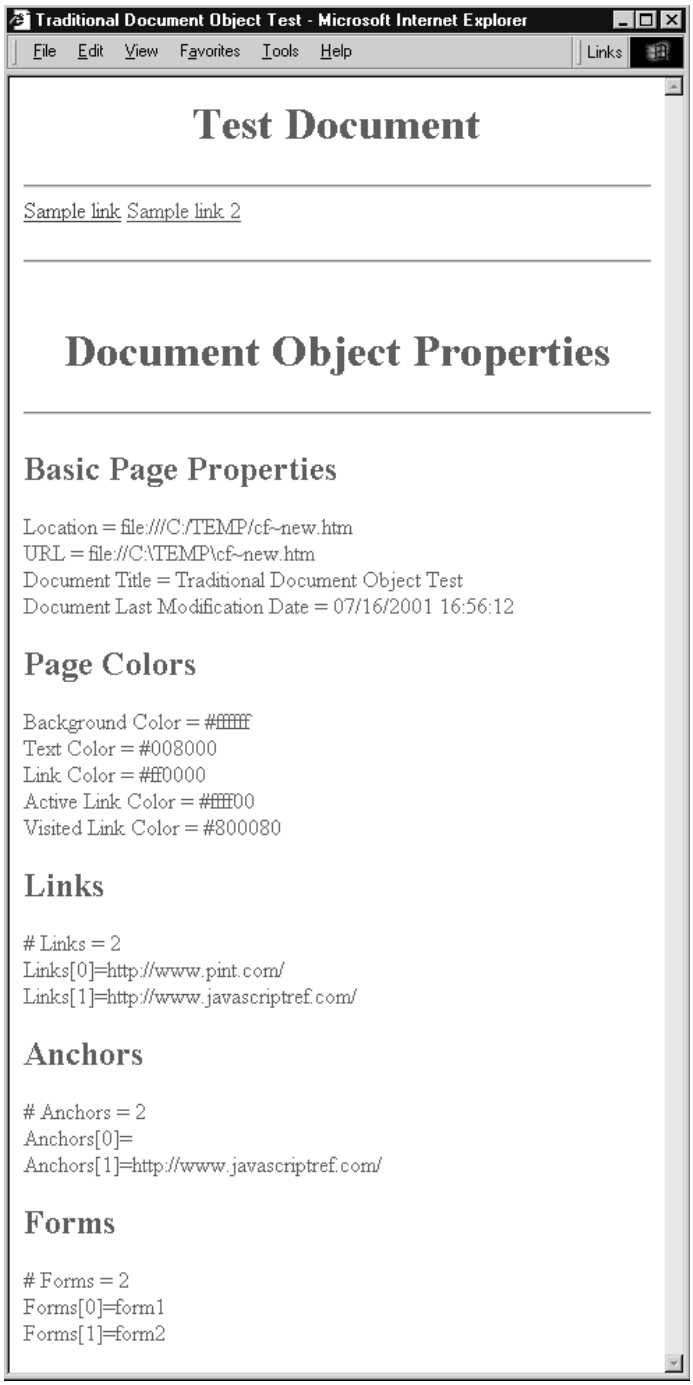

FIGURE 9-3 Simple **Document properties**

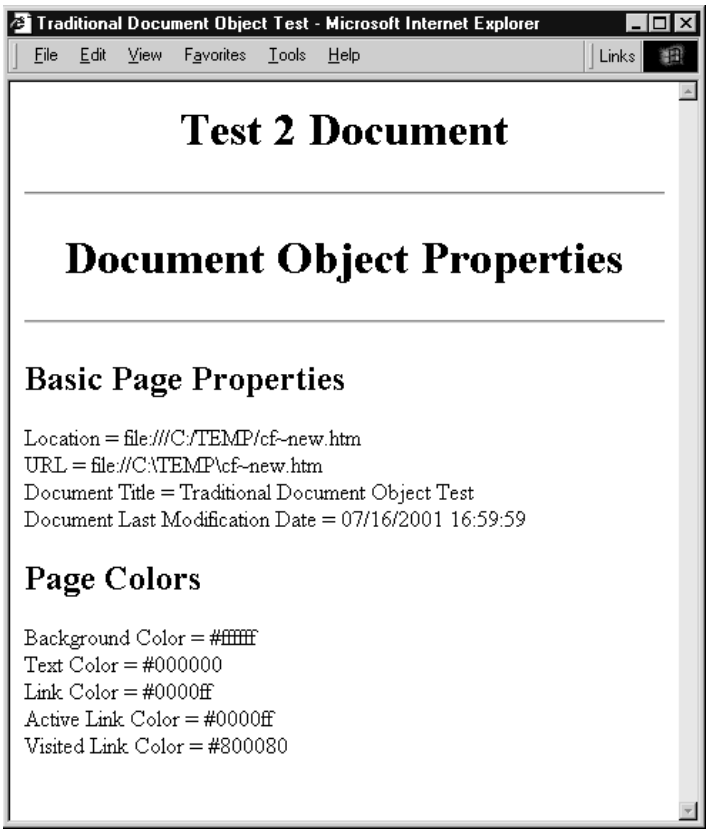

FIGURE 9-4 Some **Document properties require no HTML elements.**

JavaScript will not create, or more correctly in programming parlance *instantiate*, a JavaScript object for an element that is not present. While you will notice that browsers tend to define default values for certain types of properties such as text and link colors regardless of the presence of certain (X)HTML elements or attributes, we do not have **Form** objects, **Anchor** objects, or **Link** objects in the second example because we lacked **<form>** and **<a>** tags in the tested document. It should be very clear that the (X)HTML elements have corresponding objects in the JavaScript Document object, and that is how the two technologies interact. This last idea is the heart of the object model—the bridge between the world of markup in the page and the programming ideas of JavaScript. We now explore how to access and manipulate markup elements from JavaScript.

TIP *Given the tight interrelationship between markup and JavaScript objects, it should be no wonder that with bad (X)HTML markup you will often run into problems with your scripts. You really need to know your (X)HTML despite what people might tell you if you want to be an expert JavaScript programmer.*

Accessing Document Elements by Position

As the browser reads an (X)HTML document, JavaScript objects are instantiated for all elements that are scriptable. Initially, the number of markup elements that were scriptable in browsers was limited, but with a modern browser it is possible to access any arbitrary HTML element. However, for now let's concentrate on the (X)HTML elements accessible via the traditional JavaScript object model, particularly <**form**> and its related elements, to keep things simple. For example, if we have a document like so:

L 9-6
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Simple Form</title>
</head>
<body>
<form action="#" method="get">
  <input type="text" />
</form>
<br /><br />
<form action="#" method="get">
  <input type="text" />
  <br />
  <input type="text" />
</form>
</body>
</html>
```

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

L 9-7
```
window.document.forms[0]
```

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

L 9-8
```
window.document.forms[1]
```

However, accessing *window.document.forms[5]* or other values would cause a problem since there are only two form objects instantiated by each of the **<form>** tags.

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

L 9-9
```
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 (X)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

Markup 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 (X)HTML element is by using the **id** attribute. The **id** attribute is associated with nearly every (X)HTML element. For example, to name a particular enclosed embolded piece of text "SuperImportant," you could use the markup shown here:

L 9-10
```
<b id="SuperImportant">This is very important.</b>
```

Just like choosing unique variable names within JavaScript, naming tags in markup is very important since these tags create objects within JavaScript. If you have name collisions in your markup you are probably going to break your script. Web developers 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 confusing code and may even interfere with scripting language access.

Before the standardization of HTML 4 and XHTML 1, the **name** attribute was used to expose items to scripting instead of **id**. 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 an attribute called **name**, but these have totally different meanings beyond script access.*

Page developers must be careful to use **name** where necessary to ensure backward compatibility with older browsers. Even if this is not a concern to you, readers should not be surprised to find that many modern browsers prefer the **name** attribute on tags that support it. To be on the safe side, use **name** and **id** attributes on the tags that support both and keep them the same value. So we would write:

L 9-11
```
<form name="myForm" id="myForm" method="get" action="#">
 <input type="text" name="userName" id="userName" />
</form>
```

And then to access the form from JavaScript, we would use either:

L 9-12
```
window.document.myForm
```

or simply:

L 9-13
```
document.myForm
```

because the **Window** object can be assumed. The text field would be accessed in a similar fashion by using *document.myForm.userName*.

NOTE *Having matching **name** and **id** attribute values when both are defined is a good idea to ensure backward browser compatibility. However, be careful—some tags, particularly radio buttons, must have consistent names but varying **id** values. See Chapter 14 for examples of this problem.*

Accessing Objects Using Associate Arrays

Most of the arrays in the **Document** object are associative. That is, they can be indexed with an integer as we have seen before 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 (X)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:

L 9-14

```
<form name="myForm2" id="myForm2" method="get" action="#">
 <input name="user" type="text" value="" />
</form>
```

You can access the form as *document.forms["myForm2"]* or even use the *elements[]* array of the **Form** object to access the field as *document.forms["myForm2"].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 using the special *item()* method mentioned later in this chapter.

Event Handlers

Now that we have some idea of how to access page objects, we need to see how to monitor these objects for user activity. 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, respectively. These events are commonly associated with form buttons, form fields, images, and links, and are used for tasks like form field validation and rollover buttons. It is important to remember that not every object is capable of handling every type of 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:

L 9-15

```
<form method="get" action="#">
<input type="button" value="Click me" onclick="alert('That tickles!');" />
</form>
```

The **onclick** attribute of the **<input>** tag is used to bind 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 the part of the Document Object Model known as the *event model*. An event model is

simply a 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:

L 9-16
```
<form name="myForm" id="myForm" method="get" action="#">
<input name="myButton" id="myButton" type="button" value="Click me" />
</form>
<script 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.

Invoking Event Handlers

You can cause an event to occur at an object just as easily as you can set its handler. Objects have a method named after each event they can handle. For example, the **Button** object has a **click()** method that causes its **onclick** handler to execute (or to "fire," as many say). We can easily cause the click event defined in the previous two examples to fire:

L 9-17
```
document.myForm.myButton.click();
```

There is obviously much more to event handlers than we have described here. Both major browsers implement sophisticated event models that permit applications an extensive 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, by using a tag's name or determining its position, it is fairly easy to reference an occurrence of an HTML element that is exposed in the JavaScript object model. For example, given:

L 9-18
```
<form name="myForm" id="myForm">
<input type="text" name="userName" id="userName">
</form>
```

we would use:

L 9-19
```
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 (X)HTML

element's attributes are exposed as JavaScript object properties. So given that a text field in XHTML has the basic syntax of:

L 9-20

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

then given our last example, *document.myForm.userName.type* references the input field's **type** attribute value, in this case text, while *document.myForm.userName.size* references its displayed screen size in characters, *document.myForm.userName.value* represents the value typed in, and so on. The following simple example puts everything together and shows how the contents of a form field are accessed and displayed dynamically in an alert window by referencing the fields by name.

L 9-21

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Meet and Greet</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script 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" action="#" method="get">
<strong>What's your name?</strong>
<input type="text" name="userName" id="userName"  size="20" />
<br /><br />
<input type="button" value="Greet" onclick="sayHello();" />
</form>
</body>
</html>
```

Not only can we read the contents of page elements, particularly form fields, but we can update their contents using JavaScript. Using form fields that are the most obvious candidates for this, we modify the previous example to write our response to the user in another form field.

L 9-22

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Meet and Greet 2</title>
```

```
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script type="text/javascript">
<!--
function sayHello()
{
 var theirname = document.myForm.userName.value;
 if (theirname != "")
  document.myForm.theResponse.value="Hello "+theirname+"!";
 else
  document.myForm.theResponse.value="Don't be shy.";
}
//-->
</script>
</head>
<body>
<form name="myForm" id="myForm" action="#" method="get">
<b>What's your name?</b>
<input type="text" name="userName" id="userName"  size="20" />
<br /><br />
<b>Greeting:</b>
<input type="text" name="theResponse" id="theResponse" size="40" />
<br /><br />
<input type="button" value="Greet" onclick="sayHello();" />
</form>
</body>
</html>
```


The previous examples show how to access elements using the most traditional object model following the containment hierarchy of **window.document.*collectionname*** where ***collectionname*** is an array of JavaScript objects such as **forms[]**, **anchors[]**, **links[]** and so on that correspond to (X)HTML markup elements. However, under modern browsers that support the W3C DOM, we don't necessarily have to follow this hierarchical style of access. For example, given a tag like:

L 9-23

```
<p id="para1">Test paragraph</p>
```

we can use **document.getElementById("para1")** to access the **<p>** tag with **id** value of "para1" directly rather than accessing it through some non-existent *document.paragraphs[]* collection. Once we have accessed the tag we might set its attribute values as we did with the <**input**> tag previously. For example:

L 9-24

```
var theTag;
theTag = document.getElementById("para1");
theTag.align="right";
```

would set the **align** attribute of the paragraph to a value of "right". We could, of course, set any attribute the paragraph tag supports and even change its CSS properties via its style attribute.

While direct access seems far superior to the hierarchical method, it hasn't always been available. So before concluding this chapter and jumping into the DOM in Chapter 10, we briefly present the various Browser Object Models and how they have evolved over the years. However, do not skip these sections or dismiss them as historical notes; these object

models and approach to JavaScript are still the coding style used by many JavaScript developers, particularly those looking for backwards compatibility. Furthermore, the object models presented (particularly Netscape 3) serve as the foundation of the DOM Level 0 specification, so they will live on far into the future.

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 upon.

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, non-standard means of carrying out various tasks became a part of both Internet Explorer and Netscape. This means that the brand-new DHTML code a developer writes using the Netscape object model probably will not work in Internet Explorer (and vice versa). Fortunately, as the use of older browsers continues to dwindle and modern browsers improve their support for DOM standards, we won't have to know these differences forever and will be free to focus solely on the ideas of Chapter 10. However, for now readers are encouraged to understand the object models, and particular attention should be paid to the later Internet Explorer models, since many developers favor it over DOM standards for better or worse.

Early Netscape Browsers

The object model of the first JavaScript browser, 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 were limited. The main use of JavaScript in this browser because of its limited object model is form validation and very simple page manipulation, such as printing the last date of modification. Netscape 3's **Document** object opened the door for the first primitive DHTML-like applications. It exposes more of document content to scripts by providing the ability to access embedded objects, applets, plug-ins, 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.

Property	Description
applets[]	Array of applets (<**applet**> tags) in the document
embeds[]	Array of embedded objects (<**embed**> tags) in the document
images[]	Array of images (<**img**> tags) in the document
plugins[]	Array of plug-ins in the document

TABLE 9-4 New **Document Properties in Netscape 3**

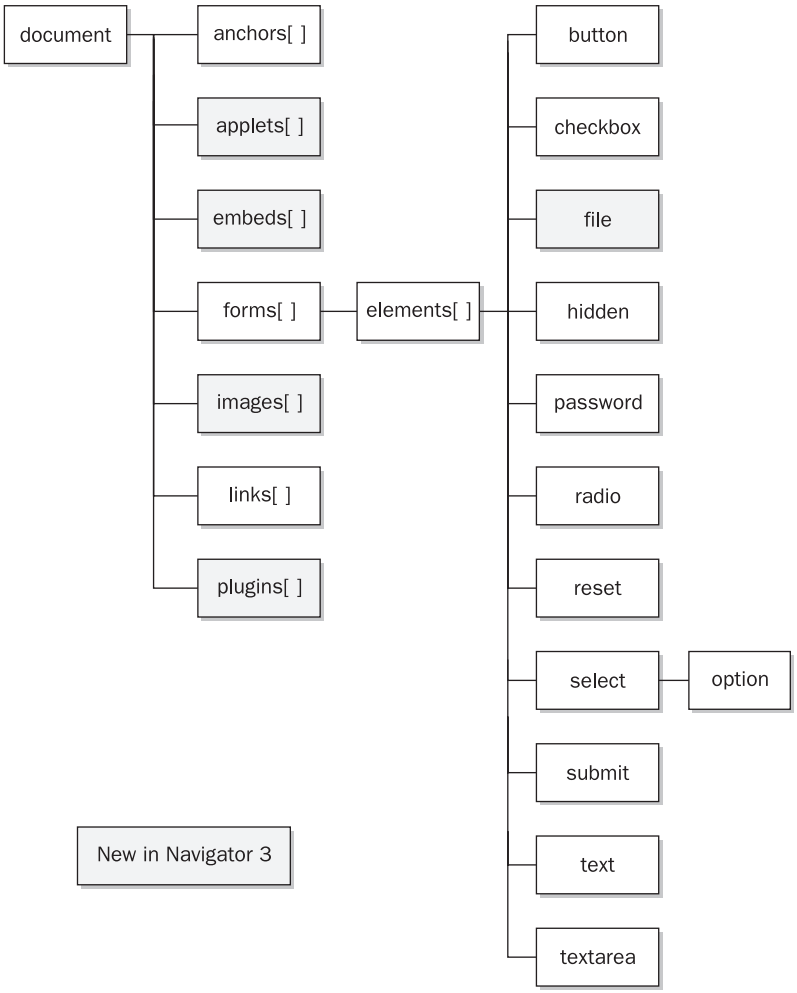

FIGURE 9-5 Netscape 3 object model

NOTE *The Netscape 3 object model without the **embeds[]** and **plugins[]** collections is the core of the DOM Level 0 standard and thus is quite important to know.*

Arguably, for many Web developers the most important addition to the **Document** object made by Netscape 3 was the inclusion of the **images[]** collection, which allowed for the now ubiquitous rollover button discussed in Chapter 15.

Netscape 4's *DHTML-Oriented Object Model*

The Document Object Model of version 4 browsers marks the point at which support for so-called Dynamic HTML (DHTML) begins. 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 and the most interesting new properties of the **Document** object are listed in Table 9-5.

While most of the aspects of the Netscape 4 object model are regulated to mere historical footnotes in Web development, one aspect of this generation of browsers that continues to plague developers is the proprietary **Layer** object.

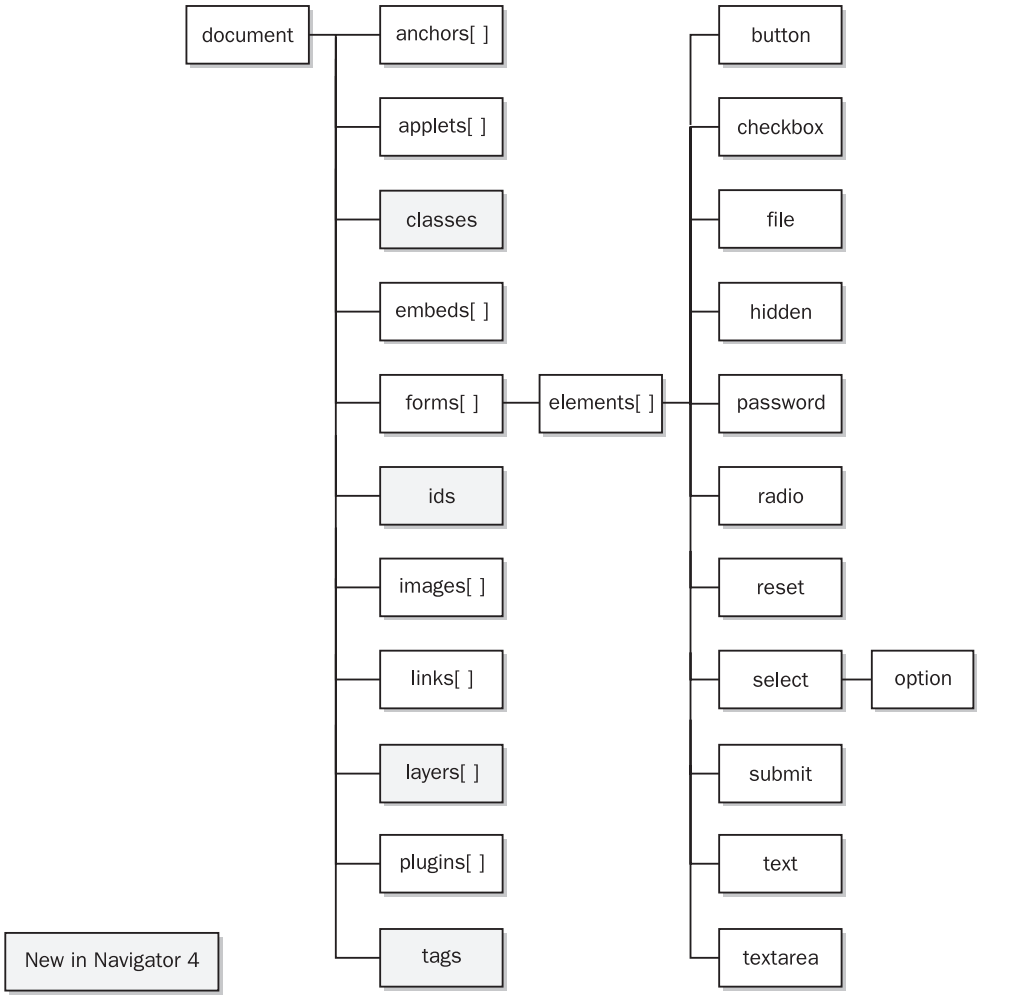

FIGURE 9-6 Netscape 4 object model

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 <**div**> 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

Netscape's document.layers[]

Netscape 4 introduced a proprietary HTML tag, **<layer>**, which allowed developers to define content areas that can be precisely positioned, moved, overlapped, and rendered hidden, visible, or even transparent. It would seem that **<layer>** should be ignored since it never made it into any W3C's HTML standard, was never included by any competing browser vendors, and it was quickly abandoned in the 6.*x* generation of Netscape. Yet its legacy lives on for JavaScript developers who need to use the **document.layers[]** collection to access CSS positioned ** ** regions in Netscape 4. To this day many DHTML libraries and applications support **document.layers[]** for better or for worse. As a quick example of Netscape 4's **Layer** object we present an example of hiding and revealing a CSS positioned region.

L 9-25

```
<!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 action="#" method="get">
<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>
```

One wrinkle with this collection is that 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, 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:

L 9-26
```
var nestedLayer = document.layers[0].document.layers[0].document;
```

Although the use of the **Layer** object hopefully will be gone forever in the near future, for backwards compatibility to Netscape 4.*x* generation browsers, they are required. Chapter 15 presents a cross-browser DHTML library that will help address such problems for interested readers.

Netscape 6, 7, and Mozilla

The release of Netscape 6 marked an exciting, but short era for Netscape browsers. While ultimately the Netscape browser itself died off, the engine and browser it was based upon, Mozilla, continues to live on in many forms. The main emphasis of this browser family 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 following chapter.

Support for nearly all of the proprietary extensions supported by older browsers like Netscape 4, most notably the **<layer>** tag and corresponding JavaScript object, have been dropped since Netscape 6. This breaks the paradigm that allowed developers to program for older browser versions knowing that such code will be supported by newer versions. Like many aspects of document models, this is both good and bad. Older code may not work in Netscape/Mozilla-based browsers, but future code written towards this browser will have a solid standards foundation. Readers unfamiliar with the Mozilla (**www.mozilla.org**) family of browsers are encouraged to take a look as they may find new and exciting changes as well as the opportunity to safely test many of the emerging W3C markup, CSS, and DOM standards discussed in Chapter 10.

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 9-7.

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.

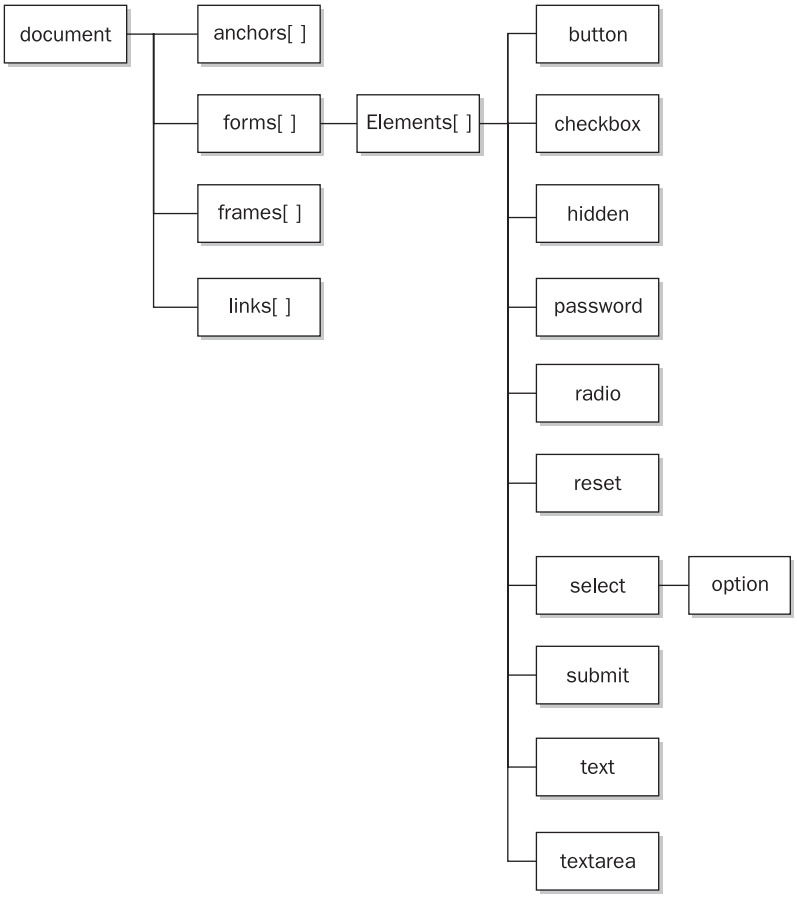

FIGURE 9-7 Internet Explorer 3 object model basically mimics Netscape 2.

Internet Explorer 4's *DHTML Object Model*

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 9-8.

Inspection of Figure 9-8 reveals that IE4 supports the basic object model of Netscape 2 and IE3, plus most of the features of Netscape 3 as well many of its own features. 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.

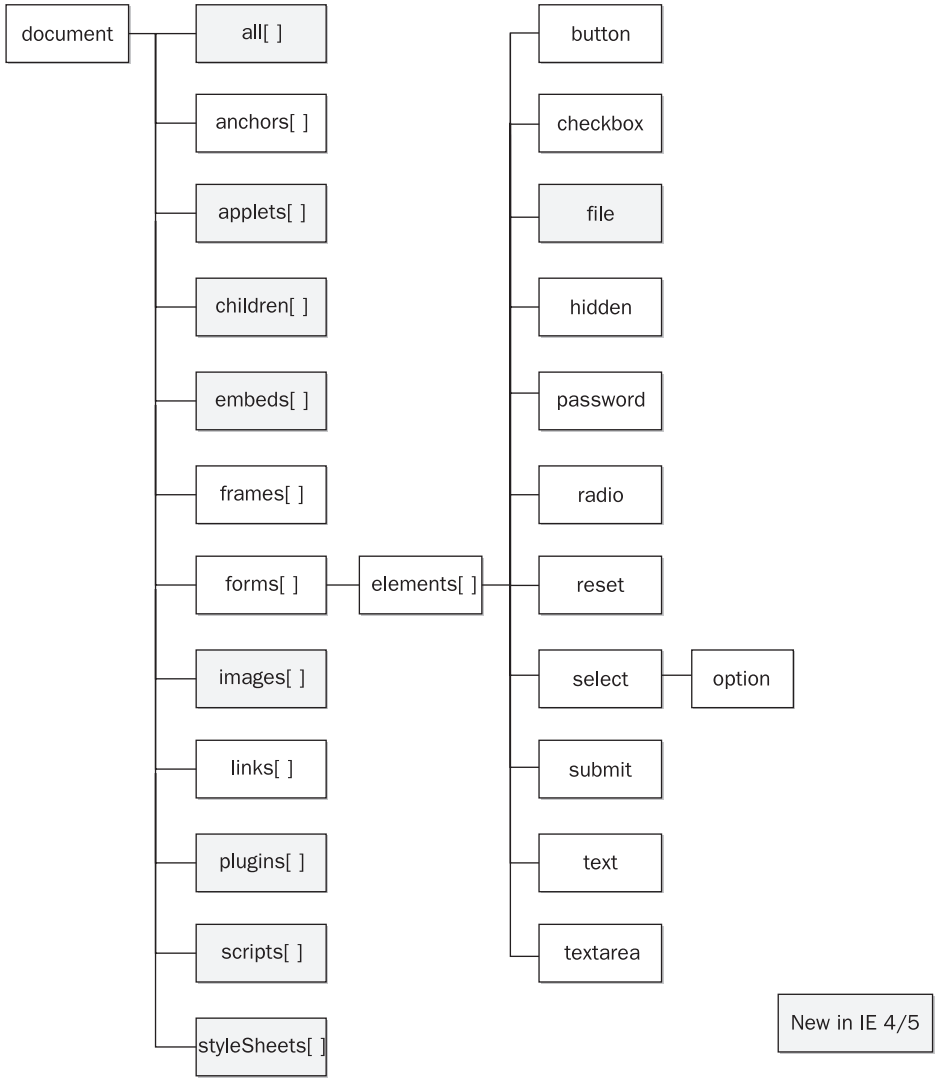

FIGURE 9-8 Internet Explorer 4 object model

IE's document.all[]

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:

L 9-27

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

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**

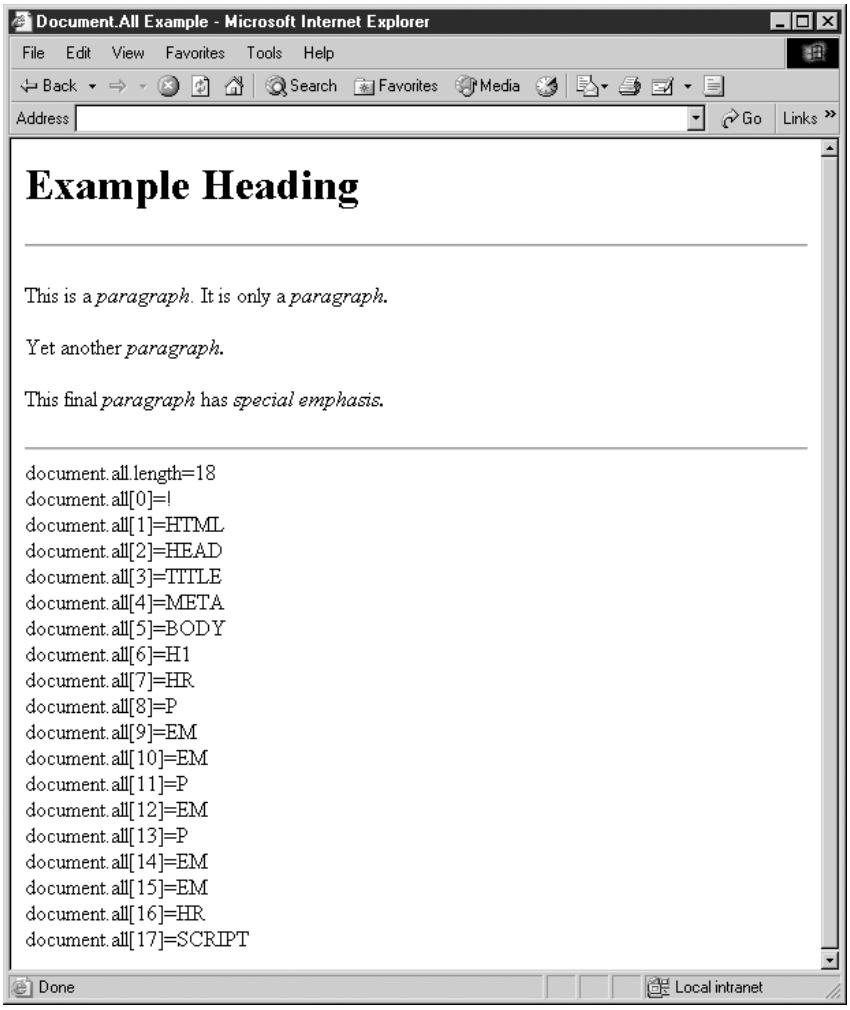

FIGURE 9-9 IE's document.all collection exposes all document elements.

```
// 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");
```

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 9-9.

L 9-28

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Document.All Example</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
</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 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!*

'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

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.

FIGURE 9-10 Some New Properties for Document Model Objects in IE4

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. We'll cover the DOM-related properties IE supported in the next chapter as well as a few of the more useful proprietary features. The balance will be covered in Chapter 21 and Appendix B.

However, even brief examination of the features available in Internet Explorer should reveal that this is the first browser where real dynamic HTML is possible, providing the means to manipulate style dynamically and to insert, modify, and delete arbitrary markup

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

FIGURE 9-4 Some New Methods for Document Model Objects in IE4

and text. For the first time JavaScript can manipulate the structure of the document, changing content and presentation of all aspects of the page at will. The following example illustrates this idea using Internet Explorer–specific syntax.

L 9-29

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Document.All Example #2</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
</head>
<body>
<!-- Works in Internet Explorer and compatible -->
<h1 id="heading1" align="center" style="font-size: larger;">DHTML Fun!!!</h1>

<form name="testform" id="testform" action="#" method="get">
<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='xx-large';" />
<input type="button" value="Smaller"
 onclick="document.all['heading1'].style.fontSize='xx-small';" />
<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;" />
</form>
</body>
</html>
```

The previous examples given here barely scratch the surface of IE's powerful Document Object Model that started first with Internet Explorer 4 and only increased in capability in later releases.

Internet Explorer 5, 5.5, and 6

The Document Object Model of Internet Explorer 5.*x* and 6.*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. Internet Explorer 5.5 continued the

trend of new features and by Internet Explorer 6 we see that IE implements significant portions of the W3C DOM. However, oftentimes developers may find that to make IE6 more standards-compliant they must be careful to "switch on" the standards mode by including a valid DOCTYPE. Yet even when enabled, the IE5/5.5/6 implementation is simply not a 100 percent complete implementation of the W3C DOM and there are numerous proprietary objects, properties, and methods that are built around the existing IE4 object model. Furthermore, given the browser's dominant position, many of its ideas like **document.all** and **innerHTML** seem to be more accepted by developers than standard's proponents would care to admit.

Opera, Safari, 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. Most provide support for the traditional JavaScript object model and embrace the fact that Internet Explorer–style JavaScript is commonplace on the Web. However, at their heart the alternative browsers focus their development efforts on the W3C standards. If the demographic for your Web site includes users likely to use less common browsers, such as Linux aficionados, it might be a good idea to avoid IE-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 only modest 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 separate scripts, one for Internet Explorer 4+ and one for other browsers like Netscape. Now with the rise of the W3C DOM standard you will often see three different code forks for full compatibility. It should be clear that the situation with competing object models is less than optimal. For those unconvinced, take a look at Chapter 15 and see what it takes to perform simple visual effects across browsers. The Web development community is ripe for change and the W3C Document Object Model provides the 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. If browser vendors continue to improve their support for the W3C DOM, there might be a point in the future where Web developers have access to a powerful, robust, and standardized interface for the manipulation of structured documents, but for now the platform lessons of this chapter are ignored at the reader's peril.

Summary

This chapter gives a basic introduction to the traditional Document Object Models. The traditional **Document** object is structured as a containment hierarchy and accessed by "navigating" through general objects to those that are more specific. 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 chapter also introduced the specific Document Object Models of the major browsers. The early browsers such as Netscape 2/3 and Internet Explorer 3 implemented the object model that is the basis of the DOM Level 0. However, the following 4.*x* generation browsers introduced some powerful "DHTML" features that were highly incompatible and have led some Web developers to embrace proprietary features. While the chapter clearly illustrated the divergent and incompatible nature of different Browser Object Models, it should not suggest this is the way things should be. Instead the W3C DOM should be embraced as it provides the way out of the cross-browser mess that plagues JavaScript developers. The next chapter explains the details of the W3C DOM and why it should revolutionize the way scripts manipulate documents.