**Chapter 12. JavaScript in Web Browsers**

**Contents:**

[The Web Browser Environment](http://docstore.mik.ua/orelly/webprog/jscript/ch12_01.htm#jscript4-CHP-12-SECT-1)  
[Embedding JavaScript in HTML](http://docstore.mik.ua/orelly/webprog/jscript/ch12_02.htm)  
[Execution of JavaScript Programs](http://docstore.mik.ua/orelly/webprog/jscript/ch12_03.htm)

The first part of this book described the core JavaScript language. Now we move on to JavaScript as used within web browsers, commonly called client-side JavaScript.[[43]](http://docstore.mik.ua/orelly/webprog/jscript/ch12_01.htm" \l "FOOTNOTE-43) Most of the examples we've seen so far, while legal JavaScript code, had no particular context; they were JavaScript fragments that ran in no specified environment. This chapter provides that context. It begins with a conceptual introduction to the web browser programming environment and basic client-side JavaScript concepts. Next, it discusses how we actually embed JavaScript code within HTML documents so it can run in a web browser. Finally, the chapter goes into detail about how JavaScript programs are executed in a web browser.

[43]The term "client-side JavaScript" is left over from the days when JavaScript was used in only two places: web browsers (clients) and web servers. As JavaScript is adopted as a scripting language in more and more environments, the term client-side makes less and less sense because it doesn't specify the client side of *what*. Nevertheless, we'll continue to use the term in this book.

**12.1. The Web Browser Environment**

To understand client-side JavaScript, you must understand the conceptual framework of the programming environment provided by a web browser. The following sections introduce three important features of that programming environment:

* The Window object that serves as the global object and global execution context for client-side JavaScript code
* The client-side object hierarchy and the document object model that forms a part of it
* The event-driven programming model

**12.1.1. The Window as Global Execution Context**

The primary task of a web browser is to display HTML documents in a window. In client-side JavaScript, the Document object represents an HTML document, and the Window object represents the window (or frame) that displays the document. While the Document and Window objects are both important to client-side JavaScript, the Window object is more important, for one substantial reason: the Window object is the global object in client-side programming.

Recall from [Chapter 4](http://docstore.mik.ua/orelly/webprog/jscript/ch04_01.htm) that in every implementation of JavaScript there is always a global object at the head of the scope chain; the properties of this global object are global variables. In client-side JavaScript, the Window object is the global object. The Window object defines a number of properties and methods that allow us to manipulate the web browser window. It also defines properties that refer to other important objects, such as the document property for the Document object. Finally, the Window object has two self-referential properties, window andself. You can use either of these global variables to refer directly to the Window object.

Since the Window object is the global object in client-side JavaScript, all global variables are defined as properties of the window. For example, the following two lines of code perform essentially the same function:

var answer = 42; // Declare and initialize a global variable

window.answer = 42; // Create a new property of the Window object

The Window object represents a web browser window or a frame within a window. To client-side JavaScript, top-level windows and frames are essentially equivalent. It is common to write JavaScript applications that use multiple frames, and it is possible, if less common, to write applications that use multiple windows. Each window or frame involved in an application has a unique Window object and defines a unique execution context for client-side JavaScript code. In other words, a global variable declared by JavaScript code in one frame is not a global variable within a second frame. However, the second frame *can* access a global variable of the first frame; we'll see how when we consider these issues in more detail in [Chapter 13](http://docstore.mik.ua/orelly/webprog/jscript/ch13_01.htm).

**12.1.2. The Client-Side Object Hierarchy and the Document Object Model**

We've seen that the Window object is the key object in client-side JavaScript. All other client-side objects are connected to this object. For example, every Window object contains a document property that refers to the Document object associated with the window and a locationproperty that refers to the Location object associated with the window. A Window object also contains a frames[] array that refers to the Window objects that represent the frames of the original window. Thus, document represents the Document object of the current window, andframes[1].document refers to the Document object of the second child frame of the current window.

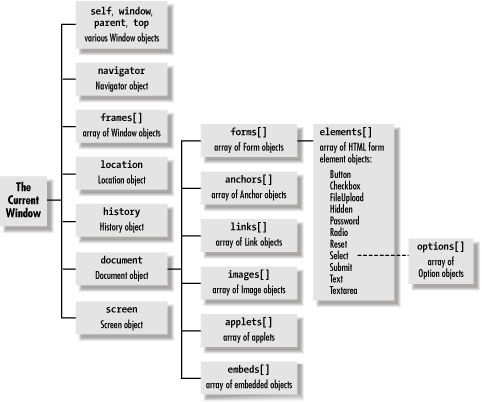
An object referenced through the current window or through some other Window object may itself refer to other objects. For example, every Document object has a forms[] array containing Form objects that represent any HTML forms appearing in the document. To refer to one of these forms, you might write:

window.document.forms[0]

To continue with the same example, each Form object has an elements[] array containing objects that represent the various HTML form elements (input fields, buttons, etc.) that appear within the form. In extreme cases, you can write code that refers to an object at the end of a whole chain of objects, ending up with expressions as complex as this one:

parent.frames[0].document.forms[0].elements[3].options[2].text

We've seen that the Window object is the global object at the head of the scope chain and that all client-side objects in JavaScript are accessible as properties of other objects. This means that there is a hierarchy of JavaScript objects, with the Window object at its root. [Figure 12-1](http://docstore.mik.ua/orelly/webprog/jscript/ch12_01.htm" \l "jscript4-CHP-12-FIG-1) shows this hierarchy. Study this figure carefully; understanding the hierarchy and the objects it contains is crucial to successful client-side JavaScript programming. Most of the remaining chapters of this book are devoted to fleshing out the details of the objects shown in this figure.



**Figure 12-1. The client-side object hierarchy and Level 0 DOM**

Note that [Figure 12-1](http://docstore.mik.ua/orelly/webprog/jscript/ch12_01.htm" \l "jscript4-CHP-12-FIG-1) shows just the object properties that refer to other objects. Most of the objects shown in the diagram have quite a few more properties than those shown.

Many of the objects pictured in [Figure 12-1](http://docstore.mik.ua/orelly/webprog/jscript/ch12_01.htm#jscript4-CHP-12-FIG-1) descend from the Document object. This subtree of the larger client-side object hierarchy is known as the document object model (DOM), which is interesting because it has been the focus of a standardization effort. [Figure 12-1](http://docstore.mik.ua/orelly/webprog/jscript/ch12_01.htm" \l "jscript4-CHP-12-FIG-1) illustrates the Document objects that have become de facto standards because they are consistently implemented by all major browsers. Collectively, they are known as the Level 0 DOM, because they form a base level of document functionality that JavaScript programmers can rely on in all browsers. These basic Document objects are the subject of [Chapter 14](http://docstore.mik.ua/orelly/webprog/jscript/ch14_01.htm) and [Chapter 15](http://docstore.mik.ua/orelly/webprog/jscript/ch15_01.htm). A more advanced document object model that has been standardized by the W3C is the subject of [Chapter 17](http://docstore.mik.ua/orelly/webprog/jscript/ch17_01.htm) and [Chapter 18](http://docstore.mik.ua/orelly/webprog/jscript/ch18_01.htm).

**12.1.3. The Event-Driven Programming Model**

In the old days, computer programs often ran in batch mode -- they read in a batch of data, did some computation on that data, and then wrote out the results. Later, with time-sharing and text-based terminals, limited kinds of interactivity became possible -- the program could ask the user for input, and the user could type in data. The computer could then process the data and display the results on screen.

Nowadays, with graphical displays and pointing devices like mice, the situation is different. Programs are generally event driven; they respond to asynchronous user input in the form of mouse-clicks and keystrokes in a way that depends on the position of the mouse pointer. A web browser is just such a graphical environment. An HTML document contains an embedded graphical user interface (GUI), so client-side JavaScript uses the event-driven programming model.

It is perfectly possible to write a static JavaScript program that does not accept user input and does exactly the same thing every time. Sometimes this sort of program is useful. More often, however, we want to write dynamic programs that interact with the user. To do this, we must be able to respond to user input.

In client-side JavaScript, the web browser notifies programs of user input by generating *events*. There are various types of events, such as keystroke events, mouse motion events, and so on. When an event occurs, the web browser attempts to invoke an appropriate *event handler*function to respond to the event. Thus, to write dynamic, interactive client-side JavaScript programs, we must define appropriate event handlers and register them with the system, so that the browser can invoke them at appropriate times.

If you are not already accustomed to the event-driven programming model, it can take a little getting used to. In the old model, you wrote a single, monolithic block of code that followed some well-defined flow of control and ran to completion from beginning to end. Event-driven programming stands this model on its head. In event-driven programming, you write a number of independent (but mutually interacting) event handlers. You do not invoke these handlers directly, but allow the system to invoke them at the appropriate times. Since they are triggered by the user's input, the handlers will be invoked at unpredictable, asynchronous times. Much of the time, your program is not running at all but merely sitting waiting for the system to invoke one of its event handlers.

The next section explains how JavaScript code is embedded within HTML files. It shows how we can define both static blocks of code that run synchronously from start to finish and event handlers that are invoked asynchronously by the system. We'll also discuss events and event handling in much greater detail in [Chapter 19](http://docstore.mik.ua/orelly/webprog/jscript/ch19_01.htm).

**12.2. Embedding JavaScript in HTML**

Client-side JavaScript code is embedded within HTML documents in a number of ways:

* Between a pair of <script> and </script> tags
* From an external file specified by the src attribute of a <script> tag
* In an event handler, specified as the value of an HTML attribute such as onclick or onmouseover
* As the body of a URL that uses the special javascript: protocol

The following sections document each of these JavaScript embedding techniques in more detail. Together, they explain all the ways to include JavaScript in web pages -- that is, they explain the allowed structure of JavaScript programs on the client side.

**12.2.1. The <script> Tag**

Client-side JavaScript scripts are part of an HTML file and are coded within <script> and </script> tags. You may place any number of JavaScript statements between these tags; they are executed in order of appearance, as part of the document loading process. <script> tags may appear in either the <head> or <body> of an HTML document.

A single HTML document may contain any number of nonoverlapping pairs of <script> and </script> tags. These multiple, separate scripts are executed in the order in which they appear within the document. While separate scripts within a single file are executed at different times during the loading and parsing of the HTML file, they constitute part of the same JavaScript program: functions and variables defined in one script are available to all scripts that follow in the same file. For example, you can have the following script somewhere in an HTML page:

<script>var x = 1;</script>

Later on in the same HTML page, you can refer to x, even though it's in a different script block. The context that matters is the HTML page, not the script block:

<script>document.write(x);</script>

The document.write( ) method is an important and commonly used one. When used as shown here, it inserts its output into the document at the location of the script. When the script finishes executing, the HTML parser resumes parsing the document and starts by parsing any text produced with document.write( ).

[Example 12-1](http://docstore.mik.ua/orelly/webprog/jscript/ch12_02.htm" \l "jscript4-CHP-12-EX-1) shows a sample HTML file that includes a simple JavaScript program. Note the difference between this example and many of the code fragments shown earlier in this book: this one is integrated with an HTML file and has a clear context in which it runs. Note also the use of a language attribute in the <script>tag. This is explained in the next section.

**Example 12-1. A simple JavaScript program in an HTML file**

<html>

<head>

<title>Today's Date</title>

<script language="JavaScript">

// Define a function for later use

function print\_todays\_date( ) {

var d = new Date( ); // Get today's date and time

document.write(d.toLocaleString( )); // Insert it into the document

}

</script>

</head>

<body>

The date and time are:<br>

<script language="JavaScript">

// Now call the function we defined above

print\_todays\_date( );

</script>

</body>

</html>

**12.2.1.1. The language and type attributes**

Although JavaScript is by far the most commonly used client-side scripting language, it is not the only one. In order to tell a web browser what language a script is written in, the <script> tag has an optional language attribute. Browsers that understand the specified scripting language run the script; browsers that do not know the language ignore it.

If you are writing JavaScript code, use the language attribute as follows:

<script language="JavaScript">

// JavaScript code goes here

</script>

If, for example, you are writing a script in Microsoft's Visual Basic Scripting Edition language,[[44]](http://docstore.mik.ua/orelly/webprog/jscript/ch12_02.htm" \l "FOOTNOTE-44) you would use the attribute like this:

[44]Also known as VBScript. The only browser that supports VBScript is Internet Explorer, so scripts written in this language are not portable. VBScript interfaces with HTML objects in the same way that JavaScript does, but the core language itself has a different syntax than JavaScript. VBScript is not documented in this book.

<script language="VBScript">

' VBScript code goes here (' is a comment character like // in JavaScript)

</script>

JavaScript is the default scripting language for the Web, and if you omit the language attribute, both Netscape and Internet Explorer will assume that your scripts are written in JavaScript.

The HTML 4 specification standardizes the <script> tag, but it deprecates the language attribute because there is no standard set of names for scripting languages. Instead, the specification prefers the use of a type attribute that specifies the scripting language as a MIME type. Thus, in theory, the preferred way to embed a JavaScript script is with a tag that looks like this:

<script type="text/javascript">

In practice, the language attribute is still better supported than this new type attribute.

The HTML 4 specification also defines a standard (and useful) way to specify the default scripting language for an entire HTML file. If you plan to use JavaScript as the only scripting language in a file, simply include the following line in the <head> of the document:

<meta http-equiv="Content-Script-Type" content="text/javascript">

If you do this, you can safely use JavaScript scripts without specifying the language or type attributes.

Since JavaScript is the default scripting language, those of us who program with it never really need to use the language attribute to specify the language in which a script is written. However, there is an important secondary purpose for this attribute: it can also be used to specify what version of JavaScript is required to interpret a script. When you specify the language="JavaScript" attribute for a script, any JavaScript-enabled browser will run the script. Suppose, however, that you have written a script that uses the exception-handling features of JavaScript 1.5. To avoid syntax errors in browsers that do not support this version of the language, you could embed your script with this tag:

<script language="JavaScript1.5">

If you do this, only browsers that support JavaScript 1.5 (and its exception-handling features) will run the script; any others will ignore it.

The use of the string "JavaScript1.2" in the language attribute deserves special mention. When Netscape 4 was being prepared for release, it appeared that the emerging ECMA-262 standard would require some incompatible changes to certain features of the language. To prevent these incompatible changes from breaking existing scripts, the designers of JavaScript at Netscape took the sensible precaution of implementing the changes only when "JavaScript1.2" was explicitly specified in the language attribute. Unfortunately, the ECMA standard was not finalized before Netscape 4 was released, and after the release, the proposed incompatible changes to the language were removed from the standard. Thus, specifying language="JavaScript1.2" makes Netscape 4 behave in ways that are not compatible with previous browsers or with the ECMA specification. (See [Section 11.6](http://docstore.mik.ua/orelly/webprog/jscript/ch11_06.htm" \l "jscript4-CHP-11-SECT-6), for complete details on these incompatibilities.) For this reason, you may want to avoid specifying "JavaScript1.2" as a value for the language attribute.

**12.2.1.2. The </script> tag**

You may at some point find yourself writing a script that uses the document.write( ) method to output a script into some other browser window or frame. If you do this, you'll need to write out a </script> tag to terminate the script you are writing. You must be careful, though -- the HTML parser makes no attempt to understand your JavaScript code, and if it sees the string "</script>" in your code, even if it appears within quotes, it assumes that it has found the closing tag of the currently running script. To avoid this problem, simply break up the tag into pieces and write it out using an expression like "</" + "script>":

<script>

f1.document.write("<script>");

f1.document.write("document.write('<h2>This is the quoted script</h2>')");

f1.document.write("</" + "script>");

</script>

Alternatively, you can escape the / in </script> with a backslash:

f1.document.write("<\/script>");

**12.2.1.3. The defer attribute**

The HTML 4 standard defines an attribute of the <script> tag that is not yet in common use but is nonetheless important. As I mentioned briefly earlier, a script may call the document.write( ) method to dynamically add content to a document. Because of this, when the HTML parser encounters a script, it must stop parsing the document and wait for the script to execute.

If you write a script that does not produce any document output -- for example, a script that defines a function but never calls document.write( ) -- you may use thedefer attribute in the <script> tag as a hint to the browser that it is safe for it to continue parsing the HTML document and defer execution of the script until it encounters a script that cannot be deferred. Doing this may result in improved performance in browsers that take advantage of the defer attribute. Note that deferdoes not have a value; it simply must be present in the tag:

<script defer>

// Any JavaScript code that does not call document.write( )

</script>

**12.2.2. Including JavaScript Files**

As of JavaScript 1.1, the <script> tag supports a src attribute. The value of this attribute specifies the URL of a file containing JavaScript code. It is used like this:

<script src="../../javascript/util.js"></script>

A JavaScript file typically has a *.js* extension and contains pure JavaScript, without <script> tags or any other HTML.

A <script> tag with the src attribute specified behaves exactly as if the contents of the specified JavaScript file appeared directly between the <script> and</script> tags. Any code that does appear between these tags is ignored by browsers that support the src attribute (although it is still executed by browsers such as Netscape 2 that do not recognize the attribute). Note that the closing </script> tag is required even when the src attribute is specified and there is no JavaScript between the <script> and </script> tags.

There are a number of advantages to using the src tag:

* It simplifies your HTML files by allowing you to remove large blocks of JavaScript code from them.
* When you have a function or other JavaScript code used by several different HTML files, you can keep it in a single file and read it into each HTML file that needs it. This reduces disk usage and makes code maintenance much easier.
* When JavaScript functions are used by more than one page, placing them in a separate JavaScript file allows them to be cached by the browser, making them load more quickly. When JavaScript code is shared by multiple pages, the time savings of caching more than outweigh the small delay required for the browser to open a separate network connection to download the JavaScript file the first time it is requested.
* Because the src attribute takes an arbitrary URL as its value, a JavaScript program or web page from one web server can employ code (such as subroutine libraries) exported by other web servers.

**12.2.3. Event Handlers**

JavaScript code in a script is executed once, when the HTML file that contains it is read into the web browser. A program that uses only this sort of static script cannot dynamically respond to the user. More dynamic programs define event handlers that are automatically invoked by the web browser when certain events occur -- for example, when the user clicks on a button within a form. Because events in client-side JavaScript originate from HTML objects (such as buttons), event handlers are defined as attributes of those objects. For example, to define an event handler that is invoked when the user clicks on a checkbox in a form, you specify the handler code as an attribute of the HTML tag that defines the checkbox:

<input type="checkbox" name="opts" value="ignore-case"

onclick="ignore-case = this.checked;"

>

What's of interest to us here is the onclick attribute.[[45]](http://docstore.mik.ua/orelly/webprog/jscript/ch12_02.htm" \l "FOOTNOTE-45) The string value of the onclick attribute may contain one or more JavaScript statements. If there is more than one statement, the statements must be separated from each other with semicolons. When the specified event -- in this case, a click -- occurs on the checkbox, the JavaScript code within the string is executed.

[45]All HTML event handler attribute names begin with "on".

While you can include any number of JavaScript statements within an event handler definition, a common technique when more than one or two simple statements are required is to define the body of an event handler as a function between <script> and </script> tags. Then you can simply invoke this function from the event handler. This keeps most of your actual JavaScript code within scripts and reduces the need to mingle JavaScript and HTML.

We'll cover events and event handlers in much more detail in [Chapter 19](http://docstore.mik.ua/orelly/webprog/jscript/ch19_01.htm), but you'll see them used in a variety of examples before then. [Chapter 19](http://docstore.mik.ua/orelly/webprog/jscript/ch19_01.htm) includes a comprehensive list of event handlers, but these are the most common:

**onclick**

This handler is supported by all button-like form elements, as well as <a> and <area> tags. It is triggered when the user clicks on the element. If an onclickhandler returns false, the browser does not perform any default action associated with the button or link; for example, it doesn't follow a hyperlink (for an <a>tag) or submit a form (for a **Submit** button).

**onmousedown , onmouseup**

These two event handlers are a lot like onclick, but they are triggered separately when the user presses and releases a mouse button. Document elements that support onclick also support these handlers. In IE 4 and Netscape 6, these handlers are actually supported by just about all document elements.

**onmouseover , onmouseout**

These two event handlers are triggered when the mouse pointer moves over or out of a document element, respectively. They are used most frequently with<a> tags. If the onmouseover handler of an <a> tag returns true, it prevents the browser from displaying the URL of the link in the status line.

**onchange**

This event handler is supported by the <input> , <select>, and <textarea> elements. It is triggered when the user changes the value displayed by the element and then tabs or otherwise moves focus out of the element.

**onsubmit , onreset**

These event handlers are supported by the <form> tag and are triggered when the form is about to be submitted or reset. They can return false to cancel the submission or reset. The onsubmit handler is commonly used to perform client-side form validation.

For a realistic example of the use of event handlers, take another look at the interactive loan-payment script in [Example 1-3](http://docstore.mik.ua/orelly/webprog/jscript/ch01_08.htm" \l "jscript4-CHP-1-EX-3). The HTML form in this example contains a number of event handler attributes. The body of these handlers is simple: they simply call the calculate( ) function defined elsewhere within a <script>.

**12.2.4. JavaScript in URLs**

Another way that JavaScript code can be included on the client side is in a URL following the javascript: pseudoprotocol specifier. This special protocol type specifies that the body of the URL is arbitrary JavaScript code to be run by the JavaScript interpreter. If the JavaScript code in a javascript: URL contains multiple statements, the statements must be separated from one another by semicolons. Such a URL might look like this:

javascript:var now = new Date( ); "<h1>The time is:</h1>" + now;

When the browser loads one of these JavaScript URLs, it executes the JavaScript code contained in the URL and uses the string value of the last JavaScript statement as the contents of the new document to display. This string value may contain HTML tags and is formatted and displayed just like any other document loaded into the browser.

JavaScript URLs may also contain JavaScript statements that perform actions but return no value. For example:

javascript:alert("Hello World!")

When this sort of URL is loaded, the browser executes the JavaScript code, but because there is no value to display as the new document, it does not modify the currently displayed document.

Often, we want to use a javascript: URL to execute some JavaScript code without altering the currently displayed document. To do this, you must be sure that the last statement in the URL has no return value. One way to ensure this is to use the void operator to explicitly specify an undefined return value. Simply use the statement void 0; at the end of your javascript: URL. For example, here is a URL that opens a new, blank browser window without altering the contents of the current window:

javascript:window.open("about:blank"); void 0;

Without the void operator in this URL, the return value of the Window.open( ) method call would be converted to a string and displayed, and the current document would be overwritten by a document that appears something like this:

[object Window]

You can use a javascript: URL anywhere you'd use a regular URL. One important way to use this syntax is to type it directly into the **Location** field of your browser, where you can test arbitrary JavaScript code without having to open your editor and create an HTML file containing the code.

javascript: URLs can be used in bookmarks, where they form useful mini-JavaScript programs, or "bookmarklets," that can be easily launched from a menu or toolbar of bookmarks.

javascript: URLs can also be used as the href value of a hyperlink. When the user clicks on such a link, the specified JavaScript code is executed. Or, if you specify a javascript: URL as the value of the action attribute of a <form> tag, the JavaScript code in the URL is executed when the user submits the form. In these contexts, the javascript: URL is essentially a substitute for an event handler.

There are a few circumstances where a javascript: URL can be used with objects that do not support event handlers. For example, the <area> tag does not support an onclick event handler on Windows platforms in Netscape 3 (though it does in Netscape 4). So, if you want to execute JavaScript code when the user clicks on a client-side image map in Netscape 3, you must use a javascript: URL.

**12.2.5. JavaScript in Nonstandard Contexts**

Both Netscape and Microsoft have implemented proprietary extensions in their browsers, and you may occasionally see JavaScript code in a context other than those described here. For example, Internet Explorer allows you to define event handlers in a <script> tag that uses special for and event attributes. Netscape 4 allows you to use JavaScript as an alternative syntax for defining CSS style sheets within a <style> tag. Netscape 4 also extends the HTML entity syntax and allows JavaScript to appear within entities (but only within the values of HTML attributes). This can result in HTML that looks like this:

<table border="&{getBorderWidth( )};">

Finally, Netscape 4 also supports a form of conditional comment based on this JavaScript entity syntax. Note that Netscape 6 and the Mozilla browser on which it is based no longer support these nonstandard uses of JavaScript.

## 12.3. Execution of JavaScript Programs

The previous section discussed the mechanics of integrating JavaScript code into an HTML file. Now we move on to discuss exactly how that integrated JavaScript code is executed by the JavaScript interpreter. The following sections explain how different forms of JavaScript code are executed. While some of this material is fairly obvious, there are a number of important details that are not so obvious.

### 12.3.1. Scripts

JavaScript statements that appear between <script> and </script> tags are executed in order of appearance; when more than one script appears in a file, the scripts are executed in the order in which they appear. If a script calls document.write( ), any text passed to that method is inserted into the document immediately after the closing </script> tag and is parsed by the HTML parser when the script finishes running. The same rules apply to scripts included from separate files with the srcattribute.

The detail that is not so obvious, but is nevertheless important to remember, is that execution of scripts occurs as part of the web browser's HTML parsing process. Thus, if a script appears in the <head> section of an HTML document, none of the <body> section of the document has been defined yet. This means that the JavaScript objects that represent the contents of the document body, such as Form and Link, have not been created yet and cannot be manipulated by that code.

Your scripts should not attempt to manipulate objects that have not yet been created. For example, you can't write a script that manipulates the contents of an HTML form if the script appears before the form in the HTML file. Some other, similar rules apply on a case-by-case basis. For example, there are properties of the Document object that may be set only from a script in the <head> section of an HTML document, before the browser has begun to parse the document content in the<body> section. Any special rules of this sort are documented in the reference page for the affected object or property in the client-side reference.

Since scripts are executed while the HTML document that contains them is being parsed and displayed, they should not take too long to run. Because scripts can create dynamic document content with document.write( ), the HTML parser must stop parsing the document whenever the JavaScript interpreter is running a script. An HTML document cannot be fully displayed until all the scripts it contains have finished executing. If a script performs some computationally intensive task that takes a long time to run, the user may become frustrated waiting for the document to be displayed. Thus, if you need to perform a lot of computation with JavaScript, you should define a function to do the computation and invoke that function from an event handler when the user requests it, rather than doing the computation when the document is first loaded.

As I noted earlier, scripts that use the src attribute to read in external JavaScript files are executed just like scripts that include their code directly in the file. What this means is that the HTML parser and the JavaScript interpreter must both stop and wait for the external JavaScript file to be downloaded. (Unlike embedded images, scripts cannot be downloaded in the background while the HTML parser continues to run.) Downloading an external file of JavaScript code, even over a relatively fast modem connection, can cause noticeable delays in the loading and execution of a web page. Of course, once the JavaScript code is cached locally, this problem effectively disappears.

### 12.3.2. Functions

Remember that defining a function is not the same as executing it. It is perfectly safe to define a function that manipulates objects that have not yet been created. Just take care that the function is not executed or invoked until the necessary variables, objects, and so on all exist. I said earlier that you can't write a script to manipulate an HTML form if the script appears before the form in the HTML file. You can, however, write a script that defines a function to manipulate the form, regardless of the relative locations of the script and form. In fact, this is a common practice. Many JavaScript programs start off with a script in the <head> of the document that does nothing more than define functions that are used in the <body> of the HTML file.

It is also common to write JavaScript programs that use scripts simply to define functions that are later invoked through event handlers. As we'll see in the next section, you must take care in this case to ensure two things: that all functions are defined before any event handler attempts to invoke them, and that event handlers and the functions they invoke do not attempt to use objects that have not yet been defined.

### 12.3.3. Event Handlers

Defining an event handler as the value of an onclick or another HTML attribute is much like defining a JavaScript function: the code is not immediately executed. Event-handler execution is asynchronous. Since events generally occur when the user interacts with HTML objects, there is no way to predict when an event handler will be invoked.

Event handlers share an important restriction with scripts: they should not take a long time to execute. As we've seen, scripts should run quickly because the HTML parser cannot continue parsing until the script finishes executing. Event handlers, on the other hand, should not take long to run because the user cannot interact with your program until the program has finished handling the event. If an event handler performs some time-consuming operation, it may appear to the user that the program has hung, frozen, or crashed.

If for some reason you must perform a long operation in an event handler, be sure that the user has explicitly requested that operation, and then notify him that there will be a wait. As we'll see in [Chapter 13](http://docstore.mik.ua/orelly/webprog/jscript/ch13_01.htm), you can notify the user by posting an alert( ) dialog box or displaying text in the browser's status line. Also, if your program requires a lot of background processing, you can schedule a function to be called repeatedly during idle time with the setTimeout( ) method.

It is important to understand that event handlers may be invoked before a web page is fully loaded and parsed. This is easier to understand if you imagine a slow network connection -- even a half-loaded document may display hypertext links and form elements that the user can interact with, thereby causing event handlers to be invoked before the second half of the document is loaded.

The fact that event handlers can be invoked before a document is fully loaded has two important implications. First, if your event handler invokes a function, you must be sure that the function is already defined before the handler calls it. One way to guarantee this is to define all your functions in the <head> section of an HTML document. This section of a document is always completely parsed (and any functions in it defined) before the <body> section of the document is parsed. Since all objects that define event handlers must themselves be defined in the <body> section, functions in the <head> section are guaranteed to be defined before any event handlers are invoked.

The second implication is that you must be sure that your event handler does not attempt to manipulate HTML objects that have not yet been parsed and created. An event handler can always safely manipulate its own object, of course, and also any objects that are defined before it in the HTML file. One strategy is simply to define your web page's user interface in such a way that event handlers refer only to previously defined objects. For example, if you define a form that uses event handlers only on the **Submit** and **Reset** buttons, you just need to place these buttons at the bottom of the form (which is where good user-interface style says they should go anyway).

In more complex programs, you may not be able to ensure that event handlers manipulate only objects defined before them, so you need to take extra care with these programs. If an event handler manipulates only objects defined within the same form, it is pretty unlikely that you'll ever have problems. When you manipulate objects in other forms or other frames, however, this starts to be a real concern. One technique is to test for the existence of the object you want to manipulate before you manipulate it. You can do this simply by comparing it (and any parent objects) to null. For example:

<script>

function set\_name\_other\_frame(name)

{

if (parent.frames[1] == null) return; // Other frame not yet defined

if (!parent.frames[1].document) return; // Document not yet loaded in it

if (!parent.frames[1].document.myform) return; // Form not yet defined

if (!parent.frames[1].document.myform.name) return; // Field not yet defined

parent.frames[1].document.myform.name.value = name;

}

</script>

<input type="text" name="lastname"

onchange="set\_name\_other\_frame(this.value)";

>

In JavaScript 1.5 and later, you can omit the existence tests in the previous code if you instead use the try/catch statement to catch the exception that will be thrown if the function is invoked before the document is fully loaded.

Another technique that an event handler can use to ensure that all required objects are defined involves the onload event handler. This event handler is defined in the<body> or <frameset> tag of an HTML file and is invoked when the document or frameset is fully loaded. If you set a flag within the onload event handler, other event handlers can test this flag to see if they can safely run, with the knowledge that the document is fully loaded and all objects it contains are defined. For example:

<body onload="window.fullyLoaded = true;">

<form>

<input type="button" value="Do It!"

onclick="if (window.fullyLoaded) doit( );">

</form>

</body>

### 12.3.3.1. onload and onunload event handlers

The onload event handler and its partner onunload are worth a special mention in the context of the execution order of JavaScript programs. Both of these event handlers are defined in the <body> or <frameset> tag of an HTML file. (No HTML file can legally contain both of these tags.) The onload handler is executed when the document or frameset is fully loaded, which means that all images have been downloaded and displayed, all subframes have loaded, any Java applets have started running, and so on. Be aware that when you are working with multiple frames, there is no guarantee of the order in which the onload event handler is invoked for the various frames, except that the handler for the parent frame is invoked after the handlers of all its child frames.

The onunload handler is executed just before the page is unloaded, which occurs when the browser is about to move on to a new page. You can use it to undo the effects of your onload handler or other scripts in your web page. For example, if your web page opens up a secondary browser window, the onunload handler provides an opportunity to close that window when the user moves on to some other web page. The onunload handler should not run any kind of time-consuming operation, nor should it pop up a dialog box. It exists simply to perform a quick cleanup operation; running it should not slow down or impede the user's transition toa new page.

### 12.3.4. JavaScript URLs

JavaScript code in a javascript: URL is not executed when the document containing the URL is loaded. It is not interpreted until the browser tries to load the document to which the URL refers. This may be when a user types in a JavaScript URL or, more likely, when a user follows a link, clicks on a client-side image map, or submits a form. javascript: URLs are often used as an alternative to event handlers, and as with event handlers, the code in those URLs can be executed before a document is fully loaded. Thus, you must take the same precautions with javascript: URLs that you take with event handlers to ensure that they do not attempt to reference objects (or functions) that are not yet defined.

### 12.3.5. Window and Variable Lifetime

A final topic in our investigation of how client-side JavaScript programs run is the issue of variable lifetime. We've seen that the Window object is the global object for client-side JavaScript and that all global variables are properties of the Window object. What happens to Window objects and the variables they contain when the web browser moves from one web page to another?

Whenever a new document is loaded into a window or a frame, the Window object for that window or frame is restored to its default state: any properties and functions defined by a script in the previous document are deleted, and any of the standard system properties that may have been altered or overwritten are restored. Every document begins with a "clean slate." Your scripts can rely on this -- they will not inherit a corrupted environment from the previous document. Any variables and functions your scripts define persist only until the document is replaced with a new one.

The clean slate we're discussing here is the Window object that represents the window or frame into which the document is loaded. As we've discussed, this Window object is the global object for JavaScript code in that window or frame. However, if you're working with multiple frames or multiple windows, a script in one window may refer to the Window objects that represent other windows or frames. So in addition to considering the persistence of variables and functions defined in Window objects, we must also consider the persistence of the Window object itself.

A Window object that represents a top-level browser window exists as long as that window exists. A reference to the Window object remains valid regardless of how many web pages the window loads and unloads. The Window object is valid as long as the top-level window is open.[[46]](http://docstore.mik.ua/orelly/webprog/jscript/ch12_03.htm" \l "FOOTNOTE-46)

[46]A Window object may not actually be destroyed when its window is closed. If there are still references to the Window object from other windows, the object is not garbage collected. However, a reference to a window that has been closed is of very little practical use.

A Window object that represents a frame remains valid as long as that frame remains within the frame or window that contains it. For example, if frame A contains a script that has a reference to the Window object for frame B, and a new document is loaded into frame B, frame A's reference to the Window object remains valid. Any variables or functions defined in frame B's Window object will be deleted when the new document is loaded, but the Window object itself remains valid (until the containing frame or window loads a new document and overwrites both frame A and frame B).

This means that Window objects, whether they represent top-level windows or frames, are quite persistent. The lifetime of a Window object may be longer than that of the web pages it contains and displays and longer than the lifetime of the scripts contained in the web pages it displays.