CHAPTER 5

Scripts and HTML Documents

In This Chapter

Where to place scripts in HTML documents

What a JavaScript statement is

What makes a script run

The difference between scripting and programming

In this chapter's tutorial, you begin to see how scripts are embedded within HTML documents and what a script statement consists of. You also see how script statements can run when the document loads or in response to user action.

Where Scripts Go in Documents

In Chapter 4, not much was said about what scripts look like or how you add them to an HTML document. That's where this lesson picks up the story.

The <SCRIPT> tag

To assist the browser in recognizing lines of code in an HTML document as belonging to a script, you surround lines of script code with a `<SCRIPT>...</SCRIPT>` tag set. This is common usage in HTML, where start and end tags encapsulate content controlled by that tag, whether the tag set be for a form or a bold font.

Depending on the browser, the `<SCRIPT>` tag has a variety of attributes you can set that govern the script. One attribute shared by Navigator and Internet Explorer is the `LANGUAGE` attribute. This attribute is essential because each browser brand and version accepts a different set of scripting languages. One setting that all scriptable browsers accept is the JavaScript language, as in

```
<SCRIPT LANGUAGE="JavaScript">
```

Other possibilities include later versions of JavaScript (version numbers are part of the language name), Microsoft's JScript variant, and the separate VBScript language. You don't need to specify any of these other languages unless your script intends to take specific advantage of a particular language version to the exclusion of all others. Until you learn the differences among the language versions, you can safely specify plain JavaScript on all scriptable browsers.

Be sure to include the ending tag for the script. Lines of JavaScript code go between the two tags:

```
<SCRIPT LANGUAGE="JavaScript">
  one or more lines of JavaScript code here
</SCRIPT>
```

If you forget the closing script tag, not only may the script not run properly, but the HTML elsewhere in the page may look strange.

Although you won't be working with it in this tutorial, another attribute works with more recent browsers to blend the contents of an external script file into the current document. A `SRC` attribute (similar to the `SRC` attribute of an `<IMG>` tag) points to the file containing the script code. Such files must end with a .js extension, and the tag set looks like the following:

```
<SCRIPT LANGUAGE="JavaScript" SRC="myscript.js"></SCRIPT>
```

Since all script lines are in the external file, no script lines are included between the start and end script tags in the document.

Tag positions

Where do these tags go within a document? The answer is, Anywhere they're needed in the document. Sometimes it makes sense to include the tags nested within the `<HEAD>...</HEAD>` tag set; other times it is essential that the script be dropped into a very specific location in the `<BODY>...</BODY>` section.

In the following four listings, I demonstrate with the help of a skeletal HTML document some of the possibilities of `<SCRIPT>` tag placement. Later in this lesson, you will see why scripts may need to go in different places within a page depending on the scripting requirements.

Listing 5-1 shows the outline of what may be the most common positioning of a `<SCRIPT>` tag set in a document: in the `<HEAD>` tag section. Typically, the Head is a place for tags that influence noncontent settings for the page — items such as `<META>` tags and the document title. It turns out that this is also a convenient place to plant scripts that are called on in response to user action.

Listing 5-1: **Scripts in the Head**

```
<HTML>
<HEAD>
<TITLE>A Document</TITLE>
<SCRIPT LANGUAGE="JavaScript">
       //statements
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

On the other hand, if you need a script to run as the page loads so that the script generates content in the page, the script goes in the `<BODY>` portion of the document, as shown in Listing 5-2. If you check the code listing for your first script in Chapter 3, you see that the script tags are in the Body, because the script needed to fetch information about the browser and write the results to the page as the page loaded.

Listing 5-2: A Script in the Body

```
<HTML>
<HEAD>
<TITLE>A Document</TITLE>
</HEAD>
<BODY>
<SCRIPT LANGUAGE="JavaScript">
        //statements
</SCRIPT>
</BODY>
</HTML>
```

It's also good to know that you can place an unlimited number of `<SCRIPT>` tag sets in a document. For example, Listing 5-3 shows a script in both the Head and Body portions of a document. Perhaps this document needs the Body script to create some dynamic content as the page loads, but the document also contains a button that needs a script to run later. That script is stored in the Head portion.

Listing 5-3: Scripts in the Head and Body

```
<HTML>
<HEAD>
<TITLE>A Document</TITLE>
<SCRIPT LANGUAGE="JavaScript">
        //statements
</SCRIPT>
</HEAD>
<BODY>
<SCRIPT LANGUAGE="JavaScript">
        //statements
</SCRIPT>
</BODY>
</HTML>
```

You also are not limited to one `<SCRIPT>` tag set in either the Head or Body. You can include as many `<SCRIPT>` tag sets in a document as are needed to complete your application. In Listing 5-4, for example, two `<SCRIPT>` tag sets are located in the Body portion, with some other HTML between them.

Listing 5-4: Two Scripts in the Body

```
<HTML>
<HEAD>
<TITLE>A Document</TITLE>
</HEAD>
<BODY>
<SCRIPT LANGUAGE="JavaScript">
        //statements
</SCRIPT>
<MORE HTML>
<SCRIPT LANGUAGE="JavaScript">
        //statements
</SCRIPT>
</BODY>
</HTML>
```

Handling older browsers

Only browsers that include JavaScript in them know to interpret the lines of code between the `<SCRIPT>` and `</SCRIPT>` tag pair as script statements and not HTML text to be displayed in the browser. This means that a pre-JavaScript browser not only will ignore the tags, but will treat the JavaScript code as page content. As you saw at the end of Chapter 3 in an illustration of your first script running on an old browser, the results can be disastrous to a page.

You can reduce the risk of old browsers displaying the script lines by playing a trick. The trick is to enclose the script lines between HTML comment symbols, as shown in Listing 5-5. Most nonscriptable browsers completely ignore the content between the `<!--` and `-->` comment tags, whereas scriptable browsers ignore those comment symbols when they appear inside a `<SCRIPT>` tag set.

Listing 5-5: Hiding Scripts from Most Old Browsers

```
<SCRIPT LANGUAGE="JavaScript">
<!--
        statements
// -->
</SCRIPT>
```

The odd construction right before the ending script tag needs a brief explanation. The two forward slashes are a JavaScript comment symbol. This symbol is necessary, because JavaScript would try to interpret the components of the ending HTML symbol (`-->`). Therefore, the forward slashes tell JavaScript to skip the line entirely; a nonscriptable browser simply treats those slash characters as part of the entire HTML comment to be ignored.

Despite the fact that this technique is often called “hiding scripts,” it does not disguise the scripts entirely. All client-side JavaScript scripts are part of the HTML document and download to the browser just like all the other HTML. Do not be fooled into thinking that you can hide your scripts entirely from prying eyes.

JavaScript Statements

Every line of code that sits between a `<SCRIPT>` and `</SCRIPT>` tag is a JavaScript *statement*. To be compatible with habits of experienced programmers, JavaScript accepts a semicolon at the end of every statement. Fortunately for newcomers, this semicolon is optional. The carriage return at the end of a statement is enough for JavaScript to know the statement has ended. A statement must be in the script for a purpose. Therefore, every statement does “something” relevant to the script. The kinds of things that statements do are

- ✦ Defining or initializing a variable
- ✦ Assigning a value to a property or variable
- ✦ Changing the value of a property or variable
- ✦ Invoking an object’s method
- ✦ Invoking a function routine
- ✦ Making a decision

If you don’t yet know what all of these mean, don’t worry — you will by the end of the next lesson. The point I want to stress is that each statement contributes to the scripts you write. The only statement that doesn’t perform any explicit action is the comment. A pair of forward slashes (no space between them) is the most common way to include a comment in a script. You add comments to a script for your benefit. Comments usually explain in plain language what a statement or group of statements does. The purpose of including comments is to remind you six months from now how your script works.

When Script Statements Execute

Now that you know where scripts go in a document, it’s time to look at when they run. Depending on what you need a script to do, you have four choices for determining when a script runs: while a document loads; immediately after a document loads; in response to user action; when called upon by other script statements.

A determining factor is how the script statements are positioned in a document.

While a document loads — immediate execution

Your first script in Chapter 3 (reproduced in Listing 5-6) runs while the document loads into the browser. For this application, it is essential that a script inspect some properties of the navigator object and include those property values in the content being rendered for the page as it loads. It makes sense, therefore, to include the `<SCRIPT>` tags and statements in the Body portion of the document. I call the kind of statements that run as the page loads *immediate* statements.

Listing 5-6: **HTML Page with Immediate Script Statements**

```
<HTML>
<HEAD>
<TITLE>My First Script</TITLE>
</HEAD>

<BODY>
<H1>Let's Script...</H1>
<HR>
<SCRIPT LANGUAGE="JavaScript">
<!-- hide from old browsers
document.write("This browser is version " + navigator.appVersion)
document.write(" of <B>" + navigator.appName + "</B>.")
// end script hiding -->
</SCRIPT>
</BODY>
</HTML>
```

Deferred scripts

The other three ways that script statements run are grouped together as what I called *deferred* scripts. To demonstrate these deferred script situations, I must introduce you briefly to a concept covered in more depth in Chapter 7: the *function*. A function defines a block of script statements summoned to run some time after those statements load into the browser. Functions are clearly visible inside a `<SCRIPT>` tag because each function definition begins with the word `function`, followed by the function name (and parentheses). Once a function has loaded into the browser (commonly in the Head portion so it loads early), it stands ready to run whenever called upon.

One of the times a function is called upon to run is immediately after a page loads. The window object has an event handler called `onLoad=`. Unlike most event handlers, which are triggered in response to user action (for example, clicking a button), the `onLoad=` event handler fires the instant that all of the page's components (including images, Java applets, and embedded multimedia) have loaded into the browser. The `onLoad=` event handler goes in the `<BODY>` tag, as shown in Listing 5-7. If you recall from Chapter 4 (Listing 4-1), an event handler can run a script statement directly. But if the event handler must run several script statements, it is usually more convenient to put those statements in a function definition, and then have the event handler *invoke* that function. That's what is happening in Listing 5-7: When the page completes loading, the `onLoad=` event handler triggers the `done()` function. That function (simplified for this example) displays an alert dialog box.

Listing 5-7: **Running a Script from the onLoad= Event Handler**

```
<HTML>
<HEAD>
<TITLE>An onLoad= script</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--
function done() {
       alert("The page has finished loading.")
}
// -->
</SCRIPT>
</HEAD>
<BODY onLoad="done()">
Here is some body text.
</BODY>
</HTML>
```

Don't worry about the curly braces or other oddities in Listing 5-7 that cause you concern at this point. Focus instead on the structure of the document and the flow: The entire page loads without running any script statements (although the page loads the `done()` function in memory so that it is ready to run at a moment's notice); after the document loads, the browser fires the `onLoad=` event handler; that causes the `done()` function to run; the user sees the alert dialog box.

To get a script to execute in response to a user action is very similar to the example you just saw for running a deferred script right after the document loads. Commonly, a script function is defined in the Head portion, and an event handler in, say, a form element calls upon that function to run. Listing 5-8 includes a script that runs when a user clicks on a button.

Listing 5-8: **Running a Script from User Action**

```
<HTML>
<HEAD>
<TITLE>An onClick= script</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--
function alertUser() {
       alert("Ouch!")
}
// -->
</SCRIPT>
</HEAD>
<BODY>
Here is some body text.
<FORM>
```

(continued)

Listing 5-8 *Continued*

```
       <INPUT TYPE="text" NAME="entry">
       <INPUT TYPE="button" NAME="oneButton" VALUE="Press Me!"
onClick="alertUser()">
</FORM>
</BODY>
</HTML>
```

Not every object must have an event handler defined for it in the HTML, as shown in Listing 5-8 — only the ones for which scripting is needed. No script statements execute in Listing 5-8 until the user clicks on the button. The `alertUser()` function is defined as the page loads, and will wait to run as long as the page remains loaded in the browser. If it is never called upon to load, there's no harm done.

The last scenario for when script statements run also involves functions. In this case, a function is called upon to run by another script statement. Before you see how that works, it will help to have been through the next lesson (Chapter 6). Therefore, I will hold off on this example until later in the tutorial.

Scripting versus Programming

It is easy to get the impression that scripting must be somehow easier than programming. "Scripting" simply sounds easier or more friendly than "programming." And in many respects this is true. One of my favorite analogies is the difference between a hobbyist who builds model airplanes from scratch and a hobbyist who builds model airplanes from commercial kits. The "from scratch" hobbyist carefully cuts and shapes each piece of wood and metal according to very detailed plans before the model starts to take shape. The commercial kit builder starts with many prefabricated parts and assembles them into the finished product. When both builders are finished, you may not be able to tell which airplane was built from scratch and which one came out of a box of components. In the end, both builders used many of the same techniques to complete the assembly, and each can take pride in the result.

As you've seen with the document object model, the browser gives scripters many prefabricated components to work with. A programmer might have to write code that builds all of that infrastructure that scripters get for "free" from the browser. In the end, both authors have working applications, each of which looks as professional as the other.

Beyond the document object model, however, "real programming" nibbles its way into the scripting world. That's because scripts (and programs) work with more than just objects. When I said earlier in this lesson that each statement of a JavaScript script does something, that "something" involves *data* of some kind. Data is the information associated with objects or other pieces of information that a script pushes around from place to place with each statement.

Data takes many forms. In JavaScript, the common incarnations of data are as numbers; text (called *strings*), objects (both from the object model and others you can create with scripts); true and false (called *Boolean* values).

Each programming or scripting language determines numerous structures and limits for each kind of data. Fortunately for newcomers to JavaScript, the universe of knowledge necessary for working with data is smaller than in a language such as Java. At the same time, what you learn about data in JavaScript is immediately applicable to future learning you may undertake in any other programming language — don't believe for an instant that your efforts in learning scripting will be wasted.

Because deep down scripting is programming, you need to have a basic knowledge of fundamental programming concepts to consider yourself a good JavaScript scripter. In the next two lessons, I set aside most discussion about the document object model and focus on the programming principles that will serve you well in JavaScript and future programming endeavors.

Exercises

1. Write the complete script tag set for a script whose lone statement is

```
document.write("Hello, world.")
```

2. Build an HTML document and include the answer to the previous question such that the page executes the script as it loads. Open the document in your browser.

3. Add a comment to the script in the previous answer that explains what the script does.

4. Create an HTML document that displays an alert dialog box immediately after the page loads and displays a different alert dialog box when the user clicks on a form button.

5. Carefully study the document in Listing 5-9. Without entering and loading the document, predict

a. What the page looks like

b. How users interact with the page

c. What the script does

Then type the listing into a text editor exactly as shown (observe all capitalization and punctuation). Do not type a carriage return after the "-" sign in the upperMe function statement: Let the line word-wrap as it does in the listing below. Save the document as an HTML file, and load the file into your browser to see how well you did.

Listing 5-9: **How Does This Page Work?**

```
<HTML>
<HEAD>
<TITLE>Text Object Value</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--
function upperMe() {
       document.converter.output.value =
document.converter.input.value.toUpperCase()
}
// -->
</SCRIPT>
</HEAD>

<BODY>
Enter lowercase letters for conversion to uppercase:<BR>
<FORM NAME="converter">
       <INPUT TYPE="text" NAME="input" VALUE="sample"
onChange="upperMe()"><BR>
       <INPUT TYPE="text" NAME="output" VALUE="">
</FORM>
</BODY>
</HTML>
```

✥ ✥ ✥