Chapter 19

Despite the similarity in name, JavaScript is very different from Java. JavaScript is a scripting language that is embedded in Web pages and interpreted as the page is loaded. Java is a general-purpose programming language that can be used for applets that execute after the page loads. JavaScript can discover a lot of information about the HTML document it is in, and manipulate a variety of HTML elements. Java, if used in a Web page at all, is relatively isolated from the Web page in which it is embedded. JavaScript has no graphics library, explicit threads, or networking. Java has the AWT, an extensive threading library, and networking options that include sockets, RMI, and JDBC.

There are currently four major browsers that support JavaScript, and, unfortunately, four different versions of JavaScript. JavaScript 1.0 is supported by Netscape 2, JavaScript 1.1 by Netscape 3, and JavaScript 1.2 by Netscape 4. Internet Explorer supports JScript, a version of JavaScript that is about halfway between JavaScript 1.0 and 1.1. I'll discuss all four versions in this chapter and the next, but will give the most attention to the Netscape versions.

A topic not covered here is server-side JavaScript. Netscape servers have a package called LiveWire that lets you use JavaScript for CGI-like server applications. This is not yet in widespread use, however. For more information on this and on alternatives, see Chapter 18, "CGI Programming and Beyond—The Server Side."
There are two basic ways that JavaScript can be used in your Web pages. The first is to build HTML dynamically as the Web page is loaded. The second is to monitor various user events and to take action when these events occur. These two syntactic styles are described in the first two sections, with the following section summarizing some other important syntax. These two styles can be combined in a variety of ways, and are used for seven general classes of applications: customizing Web pages, making pages more dynamic, validating CGI forms, manipulating cookies, interacting with frames, calling Java from JavaScript, and accessing JavaScript from Java. The remaining sections of this chapter describe each of these application areas, providing two or three examples of each. The following chapter gives details of the built-in JavaScript objects.

19.1 Generating HTML Dynamically

JavaScript code contained inside a SCRIPT element will be executed as the page is loaded, with any output the code generates being inserted into the document at the place the SCRIPT occurred. Listing 19.1 outlines the basic format. Don't worry about all the syntactic details for now; they will be explained at the end of this section. For now, just note the standard form.

Listing 19.1: Template for Generating HTML with JavaScript

```html
...<BODY>
 Regular HTML
 <SCRIPT LANGUAGE="JavaScript">
 <!--
 Build HTML Here
 // -->
 </SCRIPT>
 More Regular HTML
</BODY>
```

In Netscape 3 and 4 and recent copies of Internet Explorer 3, you can use the SRC attribute of SCRIPT to load remote JavaScript code. However, this will not work in Netscape 2, and requires the server to tag JavaScript code with the application/x-JavaScript MIME type, something most HTTP servers do not yet do automatically. For more information, see Section 16.5 (HTTP Response Headers), especially Table 16.1 (Common MIME Types).

The simplest way to build HTML is to use document.write, which places a single string in the current document. Listing 19.2 gives an example, with the result shown in Figure 19-1.

Listing 19.2: firstScript.html

```html
<DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
 <HEAD>
  <TITLE>First JavaScript Page</TITLE>
 </HEAD>
 <BODY>
  <H1>First JavaScript Page</H1>
  <SCRIPT LANGUAGE="JavaScript">
  <!--
  document.write("<HR>");
  document.write("Hello World Wide Web");
  document.write("<HR>");
  // -->
  </SCRIPT>
 </BODY>
</HTML>
```
Now, this script is not particularly useful because the JavaScript did not contribute anything that couldn’t have been done with static HTML. It is more common to use the JavaScript to build different HTML in different circumstances. Listing 19.3 gives an example, with Figures 19–2 and 19–3 showing the results in Netscape Navigator and Microsoft Internet Explorer, respectively. Note the use of the referringPage helper function and the "+" string concatenation operator. Building one long string for each chunk of HTML often yields code that is easier to read than it would be with a separate document.write for each line of HTML. Also note that this script outputs a linefeed after each line of HTML, using document.writeln instead of document.write, and adding "\n" to each line of text. This has no effect whatsoever on the resultant Web page. However, some browser versions (e.g., Netscape on Unix) show the script results when viewing the "source" of a page, and adding the extra newlines makes the result much easier to read. See Figure 19–4 for an example. This is a very useful debugging tool. If your browser shows you the source like this, you can cut and paste the results into a file, then check the syntax of this result using a standard HTML validator (see Section 1.3, "Publishing Your Document on the Web"). Highly recommended!

### Core Approach

If your browser shows you the text and markup resulting from your scripts, verify the syntax using a standard HTML validator.
Chapter 19  JavaScript: Adding Dynamic Content to Web Pages

19.1 Generating HTML Dynamically

Extracting Document Info with JavaScript

Document Info:
- URL: http://www.apl.jhu.edu/~hall/javascript/ShowInfo.html
- Modification Date: 08/24/97 11:52:37
- Title: Extracting Document Info with JavaScript
- Referring page: none

Browser Info:
- Name: Netscape
- Version: 3.01 (Win95, I)

Figure 19-2  ShowInfo result in Netscape 3.01 on Windows 95.

Extracting Document Info with JavaScript

Document Info:
- URL: http://www.apl.jhu.edu/~hall/javascript/ShowInfo.html
- Modification Date: Sun Aug 24 11:35:37 1997
- Title: Extracting Document Info with JavaScript
- Referring page: http://www.apl.jhu.edu/~hall/javascript/ShowInfo.html

Browser Info:
- Name: Microsoft Internet Explorer
- Version: 2.0 (compatible; MSIE 3.01, Windows 95)

Figure 19-3  ShowInfo result in Internet Explorer 3.01 on Windows 95.

Figure 19-4  Some browser versions will show script results when viewing document source.

Compatibility with Multiple Browsers

Note the use of the LANGUAGE attribute in Listings 19.1, 19.2, and 19.3. Although it is not strictly required, it is useful for differentiating among various JavaScript versions. Netscape 2.0 will load scripts indicated via LANGUAGE="JavaScript" but ignore those tagged with "JavaScript1.1" or "JavaScript1.2". Netscape 3 will load scripts tagged with "JavaScript1.1" but ignore those tagged with "JavaScript1.2". Netscape 4 will load scripts tagged any of the three ways. Internet Explorer 3 always honors scripts tagged with "JavaScript", always ignores those tagged with "JavaScript1.2", and is inconsistent with scripts tagged via "JavaScript1.1". On Windows 95/NT, Internet Explorer 3.01 loads "JavaScript1.1" scripts, while Internet Explorer 3.02 doesn't. On the other hand, Internet Explorer 3.02 for the Mac does honor such scripts. So it is often better to check the browser vendor and/or version than to rely on the LANGUAGE attribute.
Also note that the JavaScript code is enclosed inside an HTML comment. Again, this is not required, but it is a good standard practice. Because browsers ignore unrecognized tags, browsers that don't support JavaScript will automatically ignore the `<SCRIPT>` and `</SCRIPT>` tags, but will still see text in between. Using the HTML comment will hide the script contents as well. This works because JavaScript treats both `/*` and `<!--` as the beginning of a single-line comment. Now, this script-hiding strategy is not foolproof, because there are things inside the script that could fool older browsers if you are not careful to avoid them. For instance, in HTML 2.0, the official comment syntax is that comments must be inside pairs of `"--"`, which in turn must be between `"<"` and `">"`. Thus,

```
<!-- Foo -- Bar -->
```

is a legal comment, but

```
<!-- Foo -- Bar -->
```

is illegal. Consequently, both of the following are illegal comments in HTML 2.0.

```
<!-
var x = 3;
if (x==2) // Illegal
doSomething();
else
doAnotherThing();
// -->
<!-
var x = 3;
var y = x--; // Illegal
// -->
```

Similarly, if the script itself contains a string of the form `"</SCRIPT>"`, it will prematurely terminate the script, even in HTML 3.2. So be careful to avoid such traps.

Okay, the combination of HTML and JavaScript comments lets you hide the scripts from non-JavaScript browsers. But how do you insert meaningful alternate text in such a case? JavaScript 1.1 introduced the `NOScript` tag, whose contents are to be ignored by non-JavaScript browsers. But because Netscape 2.0 doesn’t understand this tag, it is not particularly useful. The alternative depends on whether or not you want to go to the effort of maintaining JavaScript and non-JavaScript versions of your pages. If you do, you can have users start with a non-JavaScript alternate page, but have it forward JavaScript users to the JavaScript version at the very beginning, as in Listing 19.4.

```
Listing 19.4: NondJavaScriptVersion.html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
  <head>
    <title>Some Topic</title>
  </head>
  <script language="JavaScript">
    <!--
    // Send 'real' browsers to the JavaScript version of
    // Some Topic
    location = "JavaScriptVersion.html"
    // -->
  </script>
  <body>
    <h1>Some Topic (Non JavaScript Version)</h1>
    Blah, blah, blah.
  </body>
</html>
```

If you don't plan on maintaining entirely separate pages, a better alternative is to include short comments at the end of lines starting with `"<!--"`. JavaScript will treat the entire line as a comment, while non-JavaScript browsers will treat anything on the rest of the line as regular HTML. Here is an example.

```
<script language="JavaScript">
<!--
  Sorry, this example requires JavaScript.
  // Real script here
  // -->
</script>
```
19.2 Monitoring User Events

In addition to being used to build HTML on the fly, JavaScript expressions can be attached to various HTML elements to be triggered when certain user actions are performed. You can monitor events such as clicking on a button or hypertext link, loading or unloading (exiting) a page, moving the mouse on or off a link, giving or taking away the input focus from a FORM element, submitting a CGI form, and getting an error when an image is loaded. Listing 19.5 gives an example where the dontClick method is attached to a button using the onClick attribute. Figures 19-5 and 19-6 show the result. The BUTTON input element was added to HTML just for JavaScript, but this type of event handler can be attached to a variety of standard HTML elements. Note that dontClick was defined in the HEAD instead of the BODY. This is common practice for functions that don’t directly generate HTML.

```
Listing 19.5: dontClick.html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
  <HEAD>
    <TITLE>Simple JavaScript Button</TITLE>
    <SCRIPT LANGUAGE="JavaScript">
      function dontClick() {
        alert("I told you not to click!");
      }
      // -->
    </SCRIPT>
  </HEAD>
  <BODY BGCOLOR="WHITE">
    <H1>Simple JavaScript Button</H1>
    <FORM TYPE="BUTTON"
      VALUE="Don't Click Me"
      onClick="dontClick()">
    </FORM>
  </BODY>
</HTML>
```

19.3 Basic JavaScript Syntax

The fundamental syntax of JavaScript looks a lot like Java or C. Most simple constructs will look familiar: if, “? :”, while, for, break and continue are used just as in Java. JavaScript 1.2 added a switch statement that looks very similar to Java’s switch, with the exception that the case values need not be integers. The operators + (addition and string concatenation), −, *, /, ++, --, &&, ||, and so forth are virtually identical. Trailing semicolons are optional, but I will use them throughout for the sake of familiarity. I want to outline several important features here; details are given in the next chapter “JavaScript Quick Reference.” For even more information, including updated features and new additions to the language, see the following URLs.

**JavaScript (Netscape)**

**JScript (Internet Explorer)**


Also note that Netscape provides a convenient interactive JavaScript input window (see Figure 19–7). To use it, simply open a URL of "JavaScript:" (nothing after the colon).

![Netscape interface](image)

**Figure 19–7** Netscape provides an interactive JavaScript "listener."

### Dynamic Typing

The most striking difference between Java and JavaScript is the lack of declared types in JavaScript. You don't declare types for local variables, instance variables (called "properties" in JavaScript lingo), or even return types for functions. A variable can even change type during its lifetime. So, for example, the following is perfectly legal.

```javascript
var x = 5; // int
x = 5.5; // float
x = "five point five"; // String
```

### Function Declarations

Functions are declared using the `function` reserved word. The return value is not declared, nor are the types of the arguments. Here are some examples.

```javascript
function square(x) {
    return(x * x);
}

function factorial(n) {
    if (n <= 0)
        return(1);
    else
        return(n * factorial(n - 1));
}

function printHeading(message) {
    document.writeln("<H1>" + message + "</H1>");
}
```

Functions can be passed around and assigned to variables, as follows:

```javascript
var fun = Math.sin;
alert("sin(pi/2) = " + fun(Math.PI/2));
```

Figure 19–8 shows the result.

![JavaScript function assignment](image)

**Figure 19–8** JavaScript lets you assign a function to a new name.

You can also reassign existing functions. In fact, you can even override system functions, as in the following example, although you almost always want to avoid this in real applications.

```javascript
Math.sin = Math.cos; // Don't do this at home
alert("Yikes! sin(pi/2) = " + Math.sin(Math.PI/2));
```

Figure 19–9 shows the result.
Chapter 19  JavaScript: Adding Dynamic Content to Web Pages

Objects and Classes

JavaScript's approach to object-oriented programming seems a bit haphazard compared to the strict and consistent approach of Java. Following are a few of the most unusual features.

Fields Can Be Added On-the-Fly

Adding a new property (field) is a simple matter of assigning a value to one. If the field doesn't already exist when you try to assign to it, JavaScript will create it automatically. For instance:

```javascript
var test = new Object();
test.field1 = "Value 1"; // Create field1 property
test.field2 = 7; // Create field2 property
```

Although this simplifies the addition of new properties, it also makes it difficult to catch typos, because misspelled property names will be happily accepted. Also, if you try to look up a property that doesn't exist, you will get the special undefined value. This value compares == to null.

You Can Use Literal Notation in JavaScript 1.2

In JavaScript 1.2, you can create objects using a shorthand “literal” notation of the form

```javascript
{ field1:val1, field2:val2, ... , fieldN:valN }
```

For example, the following gives equivalent values to object1 and object2 in JavaScript 1.2. It results in an error in JavaScript 1.0 or 1.1.

```javascript
var object1 = new Object();
object1.x = 3;
object1.y = 4;
object1.z = 5;
object2 = { x:3, y:4, z:5 };
```

19.3  Basic JavaScript Syntax

The "for/in" Statement Iterates Over Properties

JavaScript, unlike Java or C++, has a construct that lets you easily retrieve all of the fields of an object. The basic format is as follows:

```javascript
for (fieldName in object)
    doSomethingWith(fieldName);
```

Given a field name, you can access the field via `object["field"]` as well as via `object.field`. This is useful when iterating over fields in an object, as in Listing 19.6, which defines a general-purpose `makeObjectTable` function that will create an HTML table for a given object. Figure 19-10 gives the result in Internet Explorer 3.01.

Listing 19.6  For/in Loops

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head>
    <title>For/in Loops</title>
    <script language="JavaScript">
        function makeObjectTable(name, object) {
            document.writeln("<H2>" + name + "</H2>");
            document.writeln("<TABLE BORDER=1>
            ...
            for (fieldName in object)
                document.writeln("<TR><TD>Field</TD><TD>Value</TD>\n            document.writeln("<TR><TD>" + fieldName + "<TD>" + object[fieldname];
            document.writeln("</TABLE>");
        }
    </script>
</head>
<body bgcolor="#C0C0C0">
<h1>For/in Loops</h1>
```
Listing 19.6: ForIn.html (continued)

```html
<SCRIPT LANGUAGE="JavaScript">
<!--

var test = new Object();
test.field1 = "Field One";
test.field2 = "Field Two";
test.field3 = "Field Three";
makeObjectTable("test", test);

// -->
</SCRIPT>
</BODY>
</HTML>
```

A "Constructor" is Just a Function that Assigns to "this"

JavaScript does not have an exact equivalent to Java's class definition. The closest you get is when you define a function that assigns values to properties in the this reference. Calling this function using new binds this to a new object. For example, following is a simple constructor for a Ship class.

```javascript
function Ship(x, y, speed, direction) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.direction = direction;
}
```

Given the previous definition of makeObjectTable, putting the following in a script in the BODY of a document yields the result shown in Figure 19-11.

```javascript
var ship1 = new Ship(0, 0, 1, 90);
makeObjectTable("ship1", ship1);
```

Figure 19-10 The for/in statement iterates over the properties of an object.

Figure 19-11 Constructors are simply a shorthand way to define objects and assign properties.
Methods Are Function-Valued Properties

There is no special syntax for defining methods of objects. Instead, you simply assign a function to a property. For instance, here is a version of the Ship class that includes a move method.

```javascript
function degreesToRadians(degrees) {
    return degrees * Math.PI / 180.0;
}

function move() {
    var angle = degreesToRadians(this.direction);
    this.x = this.x + this.speed * Math.cos(angle);
    this.y = this.y + this.speed * Math.sin(angle);
}

function Ship(x, y, speed, direction) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.direction = direction;
    this.move = move;
}
```

Here is an example of its use, with the result shown in Figure 19–12.

```javascript
var ship1 = new Ship(0, 0, 1, 90);
makeObjectTable("ship1 (originally)", ship1);
ship1.move();
makeObjectTable("ship1 (after move)", ship1);
```

The "prototype" Property

In JavaScript 1.1, you can simplify the creation of methods and constant properties by use of the special prototype property. This is available in Netscape 3 and 4 and Internet Explorer, but not in Netscape 2. Once at least one object of a given class exists, assigning values to a field in the object stored in the prototype property of the class object (really the function object named for the class) gives a shared reference to this value to all members of the class that do not override it. For instance, here is a definition of Ship that adds a shared maxSpeed property intended to specify the highest speed any Ship can go:

```javascript
function Ship(x, y, speed, direction) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.direction = direction;
}
// JavaScript 1.1 only
new Ship(0, 0, 0, 0);
Ship.prototype.move = move;
Ship.prototype.maxSpeed = 50;
```

Arrays

For the most part, you can use arrays in JavaScript a lot like Java arrays. JavaScript 1.1 introduced the Array constructor to simplify building arrays. Here are a few examples.
var squares = new Array(5);
for(var i=0; i<squares.length; i++)
  vals[i] = i * i;
// Or, in one fell swoop:
var squares = new Array(0, 1, 4, 9, 16);

Using arrays this way is probably the simplest. Behind the scenes, however, JavaScript simply represents arrays as objects with numbered fields. You can access named fields using either object.field or object['field'], but numbered fields only via object[fieldNumber]. Here is an example, with Figure 19-13 showing the result:

var arrayObj = new Object();
arrayObj[0] = "Index zero";
arrayObj[10] = "Index ten";
arrayObj.field1 = "Field One";
arrayObj['field2'] = "Field Two";

makeObjectTable("arrayObj", arrayObj);

![Array/Object Duality](image)

**Figure 19-13** Arrays are really just objects with numbered fields.

In general, it is a good idea to avoid mixing array and object notation, and to treat them as two separate varieties of object. Occasionally, however, it is convenient to mix the notations, because the array notation is required when an existing string is to be used as a property name. For instance, the makeObjectTable (Listing 19.6) function relied upon this capability.

### 19.4 Using JavaScript to Customize Web Pages

**Literal Array Notation in JavaScript 1.2**

In JavaScript 1.2, you can create arrays by supplying comma-separated values inside square brackets. For example, the following assigns equivalent values to array1 and array2 in JavaScript 1.2. It results in an error in JavaScript 1.0 and 1.1.

```javascript
var array1 = new Array("fee", "fie", "fo", "fum");
var array2 = [ "fee", "fie", "fo", "fum" ];
```

### 19.4 Using JavaScript to Customize Web Pages

Because JavaScript can determine several characteristics of the current browser and document, you can use it to build different Web pages in different circumstances. For instance, you might want to use certain Microsoft-specific features on Internet Explorer and Netscape-specific features on Netscape Navigator. Or omit certain background information if users came to your page from another one of your pages, as opposed to an outside page. Or use smaller images if the user has a small screen. Or use embedded objects only if the browser has a plug in that supports them. Integration with Java (Section 19.9) will allow additional applications such as printing only a "Sorry, no Web page for you buddy!" page to users who visit from spam-tolerant ISPs, or removing the prominent link to your résumé when people access your Web page from your company domain. Following are three examples. The first shows how to customize the page based on whether the browser supports the BGCOLOR attribute of tables. The second shows how to customize the page based on the browser window size. The third illustrates how to tell if certain plug ins are available.

**Avoiding Incompatibility When Using Extensions**

In Section 2.4 (Tables), I pointed out that although applying the BGCOLOR attribute to table rows or cells can improve the appearance of some tables, careless use can cause catastrophic results in browsers that don’t support it, because your text might be invisible. Listing 19.7 shows a page that checks the browser release version, only using white on black headings if this version is three or higher. Figures 19–14 and 19–15 show the results in Netscape 3.01 and 2.02, respectively.
Core Approach

introduce all local variables with "var".

This use of JavaScript exchanges one type of incompatibility for another. Non-JavaScript browsers (e.g., Mosaic, Lynx) won't see the table at all, but users of Netscape 2 will get a better result than if the same white-on-black colors were used everywhere as in the example of Section 2.4. Although JavaScript can be used to tack on event handlers that can safely be ignored in incompatible browsers, it is difficult to use JavaScript to build HTML dynamically and still maintain compatibility with non-JavaScript browsers. In a corporate environment, you may well know which browsers will be using your page. For general Web applications, however, if you build HTML with JavaScript you will have to evaluate the benefit against the fact that you will lose some viewers.

Listing 19.7 DynamicTable.html

```html
<DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head>
  <title>Some Strawberry Varieties</title>
</head>
<body bgcolor="#000000">
  <h1>Some Strawberry Varieties</h1>

The following table summarizes a few strawberry varieties, giving their resistance to leafspot, size, relative ripening day (day 0 is about June 1 in zone 7), and fresh/frozen quality.

```

<script language="JavaScript">

```
Some Strawberry Varieties

The following table summarizes a few strawberry varieties, giving their resistance to leafspot, size, relative opening day (day 0 is about June 1 in zone 7), and fresh/frozen quality.

<table>
<thead>
<tr>
<th>Variety</th>
<th>Leafspot Resistance</th>
<th>Size</th>
<th>Opening Day</th>
<th>Quality Fresh</th>
<th>Quality Frozen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackmore</td>
<td>medium</td>
<td>small</td>
<td>0</td>
<td>fair</td>
<td>good</td>
</tr>
<tr>
<td>Cardinal</td>
<td>high</td>
<td>medium</td>
<td>5</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Spicele</td>
<td>low</td>
<td>small</td>
<td>9</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Surecrop</td>
<td>high</td>
<td>large</td>
<td>3</td>
<td>good</td>
<td>fair</td>
</tr>
<tr>
<td>Tenn. Beauty</td>
<td>medium</td>
<td>medium</td>
<td>9</td>
<td>good</td>
<td>good</td>
</tr>
</tbody>
</table>

Figure 19-14 In Netscape 3 or 4 or Internet Explorer 3, the heading background is black and the heading foreground is white.

Setting Attribute Values with JavaScript

If you are building HTML elements where only a few attributes change, you have a shorthand alternative in Netscape 3.0 and later. You can have the browser evaluate attribute values dynamically using the following syntax:

```html
<ELEMENT ATTRIBUTE="&{ JavaScript-Expression };"/>
```

For instance, the previous example could be simplified in Netscape 3 and 4 by setting the colors in the HEAD, omitting the SCRIPT in the BODY, and using

```html
<TABLE BORDER=1>

<TRbgcolor="#{headingCellColor}">
```

Of course, this would accomplish little in this particular example, because the whole goal is to support Netscape 2.0, but this is a useful capability for applications that only need to support more recent browsers.
Adjusting to the Browser Window Size

Netscape 4.0 introduced the `window.innerWidth` and `window.innerHeight` properties, which let you determine the usable size of the current browser window. Listing 19.8 uses this to shrink/stretch images to a fixed percent of the window width and to adjust the size of a heading font accordingly. Figures 19-16 through 19-18 show the results in a large, medium, and small browser window. To avoid incompatibility with earlier browsers, a fixed image and font size is used in browser versions 3 and earlier.

There are a couple of other stylistic notes to make about this example. First of all, notice that the function definitions are placed in the **HEAD**, even though they are used in the **BODY**. This is a standard practice that has three benefits. Number one: It can make the actual script easier to read. Number two: It allows a single function to be used multiple places. And number three: Because the **HEAD** is parsed before the **BODY**, it means that JavaScript routines defined in the **HEAD** will be available even if the user interrupts the loading or clicks on an image or cross reference before the page is done loading. This is particularly valuable for event-handling functions.

**Core Approach**

*Define JavaScript functions in the **HEAD**, especially if they will be used in event handlers.*

---

Also note the use of single quotes instead of double quotes for strings in this example. JavaScript allows either, so using single quotes for document.write makes it easier to embed double quotes inside the string, which are needed in this case for the “better berry” quotation.

**Core Note**

*Either single or double quotes can be used for JavaScript strings. Double quotes can be embedded inside strings created with single quotes; single quotes can be used inside strings made with double quotes.***
Strawberries are my favorite garden crop; a fresh strawberry picked five minutes ago makes the dry and woody grocery store variety seem like a <B>totally</B> different fruit. My favorite varieties are Surecrop and Cardinal.

"Doubtless God could have made a better berry, but doubtless He never did."
Determining if Plug Ins are Available

In Netscape 3 and 4 (but not Internet Explorer), the navigator.plugins array contains information about the available browser plug-ins. Each element in this array is a Plugin object that has name, description, filename, and length properties, and contains an array of MIME types objects. These properties give a short name to the plug-in, a textual description, the filename containing the plug-in, and the number of supported MIME types, respectively. Each MIME type object has properties type (MIME datatype such as "text/html"), description (descriptive text), enabledPlugin (the Plugin object supporting this type), and suffixes (a comma-separated list of file extensions associated with this type).

An interesting aspect of JavaScript's arrays is that you can reference them using a string "index" instead of an integer. In fact, as explained in Section 19.3 (Basic JavaScript Syntax) arrays in JavaScript are really just objects with numbered fields. Anyhow, you can use this shortcut to determine if a plug-in is installed, as in the following snippet.

```javascript
if (navigator.plugins["Cosmo Player 1.0"]) {
    document.write('<EMBED SRC="coolWorld.vml" .../>

else
    document.write('This example requires VRML.');
```

Note that this tells you if a particular plug-in is available. If you are more concerned about whether a certain MIME type is supported somehow (directly, through a plug-in, or via an external application), you can check the navigator.mimeTypes property. For example,

```javascript
if (navigator.mimeTypes["application/postscript"])
    addPostScriptLink();
```

For more information, see Section 20.19 (The MIME Type Object).

Compatibility Across Browser Versions

The navigator.plugins property is not available in Netscape 2. In many cases, rather than checking the browser name it is easier to use "JavaScript1.1" (no spaces, capitalized "J" and "S") as the value of the LANGUAGE attribute, as follows:

```html
<SCRIPT LANGUAGE="JavaScript1.1">
```

Scripts tagged this way will be ignored in Netscape 2 but honored in Netscape 3 and 4. If your script is specific to Netscape 4, use
<SCRIPT LANGUAGE="JavaScript1.1">  

Now, this brings up the issue of Internet Explorer (version 3). The variation of JavaScript it supports (JScript) is partway between JavaScript 1.0 and 1.1, and support for the JavaScript1.1 value is inconsistent. For instance, on Windows 95 and NT, Internet Explorer 3.01 honors scripts tagged that way, while Internet Explorer 3.02 ignores them! Internet Explorer 3.02 on the Mac honors them. Consequently, code that depends on 1.1 features not available in Internet Explorer should designate "JavaScript1.1" and test the browser version, as follows:

<SCRIPT LANGUAGE="JavaScript1.1">  

<!--
if (navigator.appName == "Netscape") {
  // script actions here
}
// -->
</SCRIPT>

Core Approach

When using capabilities specific to Netscape 3 or 4, use LANGUAGE="JavaScript1.1" and check navigator.appName. Alternatively, use LANGUAGE="JavaScript" and check navigator.appVersion.

Listing 19.9 uses this approach to build a table of available plug-ins and their supported MIME types. Figures 19–19 and 19–20 show results in Netscape 3 and 4, respectively.

```
<SCRIPT LANGUAGE="JavaScript1.1">  

function printRow(plugin) {
  document.write("<TR><TD>" + plugin.name + "n" +
  "<TD>" + plugin.description + "n" +
  "<TD>");
  document.write(plugin[0].type);
  for (var i=1; i<plugin.length; i++)
    document.writeln("<BR>" + plugin[i].type);
}

<!--
</SCRIPT>
</HEAD>

<BODY>

<HI>Plugins Supported</HI>

<SCRIPT LANGUAGE="JavaScript1.1">  

if (navigator.appName == "Netscape") {
  document.writeln("<TABLE BORDER=1>
  " + "<TR>" + "<TH>" + "<TH>" + "<TH>" + "<TH>");
  for (var i=0; i<navigator.plugins.length; i++)
    printRow(navigator.plugins[i]);
  document.writeln("</TABLE>");
}

<!--
</SCRIPT>

</BODY>
</HTML>
```
19.5 Using JavaScript to Make Pages Dynamic

In most of the previous examples, parts of the document were built dynamically (when the page was loaded), but the resultant document was normal HTML. JavaScript can also be used to create elements that are dynamic. For instance, one common application is to create images that change when the user moves the mouse over them. This can be used to implement toolbars with regions that “light up” to indicate hypertext links or custom buttons that show a grayed-out image when you press them. Alternatively, by using timers, JavaScript can animate images even without being triggered by user events. In Netscape 4, the possibilities are even greater. JavaScript can manipulate layers, scroll the document, and even move the browser window around on the screen. This opens up myriad possibilities for abuse; if you think users hate <BLINK>, wait until they try your page with bouncing regions. Run for cover! On the other hand, if used with restraint, these new features can add value to certain types of pages.

**Modifying Images Dynamically**

In JavaScript 1.1 (Netscape 3 and 4 but not Internet Explorer), the document.images property contains an array of Image objects corresponding to each IMG element in the current document. To display a new image, simply set the SRC property of an existing image to a string representing a different image file. For instance, the following function changes the first image in a document.

```javascript
function changeImage() {
    document.images[0].src = "images/new-image.gif";
}
```
This function could be invoked from an event handler (e.g. when the user clicks a button) or even executed automatically after a certain amount of time. Now, referring to images by number is not very flexible, because the addition of a new image in the middle of the document would require changing the references to all later images. Fortunately, JavaScript lets you name images using a new NAME attribute of the IMG element as follows:

```html
<IMG SRC="cool-image.jpg" NAME="cool"
     WIDTH=75 HEIGHT=25>
```

Because JavaScript lets you refer to array elements by name instead of number, you could then do the following:

```javascript
function improveImage() {
    document.images["cool"].src = 'way-cool.jpg';
}
```

However, be careful with this; images embedded in table cells appear twice in the `images` array in Netscape 3. This bug is fixed in Netscape 4.

### A Clickable Image Button

This idea can be used to create images that change when you click on them. For example, following is a `clickButton` function that temporarily changes an image, switching it back to the original version after 1/10 of a second. To do this, it uses the `setImage` and `setTimeout` function. The first of these is defined as follows, while `setTimeout` is a built-in routine which takes a string designating a JavaScript expression and a time in milliseconds. It returns immediately, but starts a background process that waits for the specified time then executes the code specified by the string.

```javascript
function setImage(name, image) {
    document.images[name].src = image;
}

function clickButton(name, grayImage) {
    var origImage = document.images[name].src;
    setImage(name, grayImage);
    var resetString = "setImage('" + name + "," + origImage + ")";
    setTimeout(resetString, 100);
}
```

To use this for a clickable image button, we need to do two more things: (1) attach the routine to a button or buttons, and (2) make sure that the images needed are already cached by the browser. The first step is straight-forward: simply use the `onClick` attribute of the `<A HREF...>` element, as shown below. Remember to put the `<A>` on the same line as the IMG element to prevent the Netscape bug that would otherwise display an underscore after the image. See Section 3.3 (Specifying Hypertext Links) for more information on this problem.

```html
<A HREF="location1.html"
    onClick="clickButton('Button1',
        'images/Button1-Down.gif')">
    <IMG SRC="images/Button1-Up.gif" NAME="Button1"
         WIDTH=150 HEIGHT=25></A>
```

```html
<A HREF="location2.html"
    onClick="clickButton('Button2',
        'images/Button2-Down.gif')">
    <IMG SRC="images/Button2-Up.gif" NAME="Button2"
         WIDTH=150 HEIGHT=25></A>
```

Finally, before trying to display an image, you should make sure it is already loaded. This will prevent long pauses when the button is pressed. You can do this by creating an `Image` object (Section 20.12), then setting its `src` property. Oddly, this `Image` object never actually gets used; its only purpose is to force the browser to load (and cache) the image. Here is an example:

```javascript
imageFiles = new Array("images/Button1-Up.gif",
                      "images/Button1-Down.gif",
                      "images/Button2-Up.gif",
                      "images/Button2-Down.gif");

imageObjects = new Array(imageFiles.length);

for(var i=0; i<imageFiles.length; i++) {
    imageObjects[i] = new Image(150, 25);
    imageObjects[i].src = imageFiles[i];
}
```

If you are handling a lot of images, you can simplify the process by having a consistent naming scheme for the images; I'll give an example later in this section.

Now, this process is sufficient for browsers that support image manipulation. However, if the browser understands `onClick` but not the `Image` object, the code will result in an error. This is a serious problem, since both Netscape 2 and Internet Explorer 3 (on Windows 95/NT) are in this category. Surprisingly, Internet Explorer 3.02 supports image manipulation on the Macintosh, but not on Windows 95/NT. So checking the JavaScript version or browser vendor is
Chapter 19  JavaScript: Adding Dynamic Content to Web Pages

not a sufficient test. A better approach is to explicitly test if images are supported by checking for the document.images property, as below:

```javascript
var imagesSupported = (document['images'] != null);
```

Then, you can wrap the image manipulation code inside:

```javascript
if (imagesSupported) {
  ...
}
```

This will prevent errors on incompatible browsers, while still allowing the image highlighting on as many browsers as possible. If this seems like an awful lot of work to go through to maintain portability, you are absolutely right! It takes quite a bit of effort to create JavaScript documents that support as many browsers as possible while still working reasonably on less-capable systems. Listing 19.10 shows the whole process put together.

Core Approach

Make image manipulation code portable by checking for the existence of document.images.

Listing 19.10 ImageButton.html (continued)

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
  <head>
    <title>JavaScript Image Buttons</title>
    <script LANGUAGE="JavaScript">
      <!--

      // Netscape 2.0 and Windows versions of Internet
      // Explorer understand onmouseover but not
      // the image stuff, so we need to have an empty
      // placeholder for the clickButton method.

      function clickButton(name, grayImage) {}

      // -->
    </script>
    </head>
    <body>
      <h1>JavaScript Image Buttons</h1>
      <a href="location1.html"
        onClick="clickButton('Button1',
          'images/Button1-Up.gif');"
        onclick="clickButton('Button1',
          'images/Button1-Down.gif');"
        onmouseover="setImage('Button1',
          'images/Button1-Over.gif');"
        onmouseout="setImage('Button1',
          'images/Button1-Idle.gif');">
        Button1
      </a>
    </body>
  </html>
```

19.5  Using JavaScript to Make Pages Dynamic

Listing 19.10 ImageButton.html (continued)

```html
<SCRIPT LANGUAGE="JavaScript">
<script LANGUAGE="JavaScript">
<!
```

var imagesSupported = (document['images'] != null);

```javascript
if (imagesSupported) {
  imageFiles = new Array('images/Button1-Up.gif",
    'images/Button1-Down.gif",
    'images/Button1-Over.gif",
    'images/Button1-Idle.gif');

  imageObjects = new Array(imageFiles.length);

  for(var i=0; i<imageFiles.length; i++) {
    imageObjects[i] = new Image(150, 25);
    imageObjects[i].src = imageFiles[i];
    
    function setImage(name, image) {
      document.images[name].src = image;
    }

    function clickButton(name, grayImage) {
      if (imagesSupported) {
        var origImage = document.images[name].src;
        setImage(name, grayImage);
        var resetString =
          'setImage('+name+'","'+origImage+'")';
        setTimeout(resetString, 100);
      }
    }

    // -->
  </SCRIPT>
  </HEAD>
  <BODY>
    <h1>JavaScript Image Buttons</h1>
    <a href="location1.html"
      onClick="clickButton('Button1',
        'images/Button1-Up.gif');"
      onclick="clickButton('Button1',
        'images/Button1-Down.gif');"
      onmouseover="setImage('Button1',
        'images/Button1-Over.gif');"
      onmouseout="setImage('Button1',
        'images/Button1-Idle.gif');">
      Button1
    </a>
  </BODY>
</html>
```

continued
Highlighting Images Under the Mouse

An even more common application of the image modification process is to create a series of images that change as the user moves the mouse over them, using the hypertext link's onMouseOver to display the highlighted image and onMouseOut to change the image back. This can make toolbars more appealing by providing visual cues as to which regions are clickable. However, when dealing with a large number of images, listing each explicitly when preloading them can be tedious. Listing 19.11 shows an approach that simplifies this process considerably, using consistent names. The normal image and highlighted image are both derived from the NAME of the IMG element (see regularImageFile and negativeImageFile), negating the need to list the full filenames in the array listing the images to be preloaded, or to pass the highlighted image name in the onMouseOver call. Toolbars of this type are most commonly used with frames; Listing 19.12 and Listing 19.13 show the rest of the frame structure, while Figures 19–21 and 19–22 show the results.

If you can't remember how to use frames, this would be a good time to review them (Chapter 4), because they are used quite frequently with JavaScript.
function cacheImages(index) {
    regularImageObjects[index] = new Image(150, 25);
    regularImageObjects[index].src =
        regularImageFile(imageNames[index]);
    negativeImageObjects[index] = new Image(150, 25);
    negativeImageObjects[index].src =
        negativeImageFile(imageNames[index]);
}

if (imagesSupported) {
    imageNameArrays = new Array("Home", "Tibet", "Nepal",
        "Austria", "Switzerland");
    regularImageObjects = new Array(imageNames.length);
    negativeImageObjects = new Array(imageNames.length);
    // Put images in cache for fast highlighting
    for (var i = 0; i < imageNameArrays.length; i++)
        cacheImages(i);
}

// This is attached to onmouseover -- change image
// under the mouse to negative (reverse video) version.

function highlight(imageName) {
    if (imagesSupported)
        document.images[imageName].src =
            negativeImageFile(imageName);

    // This is attached to onmouseout -- return image to
    // normal.

function unHighlight(imageName) {
    if (imagesSupported)
        document.images[imageName].src =
            regularImageFile(imageName);

    // -->
    </SCRIPT>
    </HEAD>

    <BODY BGCOLOR="WHITE">
we have package deals for beginner, experienced, and expert climbers, discount priced (*) for the budget-conscious traveller.

HPT is currently arranging trips to the following exciting locations:

- Tibet
- Nepal
- Austria
- Switzerland

Sign up today!

(*) No ropes or safety equipment provided on discount tours.
Moving Layers

Netscape 4.0 introduced "layers." These are HTML regions that can overlap and be positioned arbitrarily, and are discussed at length in Sections 5.13 (Specifying Layers Using the LAYER and ILAYER Elements) and 5.14 (Specifying Layers Using Style Sheets). JavaScript 1.2 lets you access layers via the document.layers array, each element of which is a Layer object with properties corresponding to the attributes of the LAYER element. A named layer can be accessed via document.layers['layer name']
rather than by using an index, or simply by using `document.layers[1].visibility = "hidden";` to hide a layer. Layers can be accessed this way no matter how they are defined in HTML: by the `layer` element, by the `ILAYER` element, or via `style`.

Listing 19.14 presents an example with two layers that are initially visible (Figure 19-23). When a certain button is pressed, the first layer is made invisible near the upper-left corner, then moves down over the top of the page to its final location, where it annotates an image (Figure 19-24). Clicking a second button hides the first layer and displays a second, which then drifts to its final location (Figure 19-25). The properties and methods of the `Layer` object are described in Section 20.15, but for this example the only ones of interest are `visibility` ("show" or "hidden") and position (absolute location in window). The methods used are `moveToAbsolute` (position layer at absolute location) and `moveBy` (move layer relative to previous position).

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head>
<title>Camps on K-3</title>

<script language="JavaScript1.2">
function hideCamps() {
    // Or document.baseCamp.visibility = "hidden";
    document.layers["baseCamp"].visibility = "hidden";
    document.layers["highCamp"].visibility = "hidden";
}

function moveBaseCamp() {
    baseCamp.moveBy(1, 3);
    if (baseCamp.pageX < 130) {
        setTimeout("moveBaseCamp()");
    }
    // Hide camps, position base camp near top-left
    // corner, make it visible, then have it slowly
    // drift down to final position.
</script>
</head>
<body>
</body>
</html>
```
function showBaseCamp() {
  hideCamps();
  baseCamp = document.layers['baseCamp'];
  baseCamp.moveToAbsolute(0, 20);
  baseCamp.visibility = "show";
  moveBaseCamp();
}

function moveHighCamp() {  
  highCamp.moveBy(2, 1);
  if (highCamp.pageX < 110)
    setTimeout("moveHighCamp()", 10);
}

// Hide camps, position high camp near top-left
// corner, make it visible, then have it slowly
// drift down to final position.

function showHighCamp() {
  hideCamps();
  highCamp = document.layers['highCamp'];
  highCamp.moveToAbsolute(0, 65);
  highCamp.visibility = "show";
  moveHighCamp();
}

// -->
</SCRIPT>
</HEAD>

<BODY>

<IMG SRC="images/peak4.gif" WIDTH=511 HEIGHT=500
     ALIGN="LEFT">
<H1>Camps on K-3</H1>
The High Peaks Tours trip to the summit:
<UL>
  <LI>Day 1: Travel to Base Camp
  <LI>Day 2: Climb to High Camp
  <LI>Day 3: Ascend summit, return to High Camp
  <LI>Day 4: Descend to Base Camp
  <LI>Day 5: Return Home
</UL>

continued
Listing 19.14: Camps.html (continued)

```html
<layer id="highCamp" pageX=50 pageY=100
    visibility="hidden">
  <table>
    <tr>
      <th bgcolor="#ffffff" width=50>
        <font size="+2">High Camp</font>
      </th>
    </tr>
    <td><img src="images/Arrow-Right.gif"></td>
  </table>
</layer>

<layer id="baseCamp" pageX=50 pageY=100
    visibility="hidden">
  <table>
    <tr>
      <th bgcolor="#ffffff" width=50>
        <font size="+3">Base Camp</font>
      </th>
    </tr>
    <td><img src="images/Arrow-Right.gif"></td>
  </table>
</layer>

<form>
  <input type="button" value="Show Base Camp"
         onclick="showBaseCamp()">
  <input type="button" value="Show High Camp"
         onclick="showHighCamp()">
  <input type="button" value="Hide Camps"
         onclick="hideCamps()">
</form>
</body>
</html>
```

Figure 19-23 When the page is first loaded, both layers are hidden.
19.6 Using JavaScript to Validate CGI Forms

Another important application of JavaScript is to check the format of CGI forms before the form is submitted to the server. Contacting the server can be expensive, especially over a slow connection, and simple tasks like checking that all required fields are filled out or making sure textfields that should
contain numbers don't have strings should be performed on the client if at all possible. The document.forms property contains an array of Form entries contained in the document. As usual in JavaScript, named entries can be accessed via name instead of by number, plus named forms are automatically inserted as properties in the document object, so any of the following formats would be legal to access forms:

```javascript
var firstForm = document.forms[0];
// Assumes <FORM NAME="orders" ...>
var orderForm = document.forms["orders"]; // Assumes <FORM NAME="register" ...>
var registrationForm = document.register;
```

The Form object contains an elements property that holds an array of Element objects. You can retrieve form elements by number, by name from the array, or via the property name:

```javascript
var firstElement = firstForm.elements[0]; // Assumes <INPUT ... NAME="quantity">
var quantityField = orderForm.elements["quantity"]; // Assumes <INPUT ... NAME="submitSchedule">
var submitButton = register.submitSchedule;
```

Different elements can be manipulated different ways. Some generally important capabilities include the ability to execute code before a form is submitted (via the onsubmit attribute of FORM), look up and change form values (via the element's value property), to recognize when keyboard focus is gained or lost (via onFocus and onBlur), and to notice changed values automatically (via onChange). The following examples illustrate two major ways form entries are checked: individually (each time one changes) and en masse (only when the form is submitted). For more details, see the Form and Element objects (Sections 20.6 and 20.8).

**Checking Values Individually**

Listing 19.15 gives a very simple input form containing a singletextfield and a SUBMIT button. JavaScript is used in two ways. First, when the textfield gets the input focus, text describing the expected value is printed in the status line. The status line is reset when the textfield loses the focus. Secondly, if the user changes the textfield value to something illegal, a warning is issued when they leave the textfield. Then the textfield value is reset (changing the value via JavaScript does not trigger onChange) and it is given the input focus for the user to enter a correction. Figures 19-26 and 19-27 show the results.

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head>
<title>On-Line Training</title>
</head>
<script language="JavaScript">
//
// Print a description of the legal text in the
// status line

function describeLanguage()
  status = "Enter an important Web language";
}

// Clear status line

function clearStatus()
  status = "";
}

// When the user changes and leaves textfield, check
// that a valid choice was entered. If not, alert
// user, clear field, and set focus back there.

function checkLanguage()
  // or document.forms["LangForm"].elements["langField"]
  var field = document.langForm.langField;
  var lang = field.value;
  var prefix = lang.substring(0, 4); toUpperCase();
  if (prefix != "JAVA") {
    alert("Sorry, " + lang + " is not valid.\n    Please try again.");
    field.value = ""; // Erase old value
    field.focus(); // Give keyboard focus
  }
}

// -->
</script>
</head>
<body bgcolor="WHITE">
<h1>On-Line Training</h1>

continued
Checking Values When Form Is Submitted

Sometimes it is more convenient to check the entire form in one fell swoop. Some people feel that correcting the user after every mistake is too intrusive, since they may enter values temporarily but correct them before submission. Other times it is simply easier to check a bunch of values than to create a separate function for each of several dozen input elements. The key idea is that the function invoked by the form onSubmit attribute prevents submission of the form if it returns false.

Checking numeric values is one of the most common validation tasks, but the value property of textfields and text areas is a string, not a number. Fortunately, JavaScript provides two built-in functions to assist in this: parseInt and parseFloat. These take a string as input and return either an integer or a floating point number. In JavaScript 1.1, if no prefix of the string is a valid number, they return the special value NaN which can be recognized with the built-in isNaN function (not by ==, because NaNs return false for all comparisons). Unfortunately, in JavaScript 1.0 these functions return 0 for illegal numbers, and the isNaN function is not defined, making it difficult to recognize incorrect numeric values in scripts that must run on both platforms. Surprisingly, JavaScript does not have a property corresponding to “the current language version,” so you have to create your own as in the following example.

```javascript
<SCRIPT LANGUAGE="JavaScript">
<!--
languageVersion = 1.0;
// -->
</SCRIPT>
<SCRIPT LANGUAGE="JavaScript1.1">
<!--
languageVersion = 1.1;
// -->
</SCRIPT>
<SCRIPT LANGUAGE="JavaScript1.2">
<!--
languageVersion = 1.2;
// -->
</SCRIPT>
```

This idea is used to make the following portable version of parseInt, except that Internet Explorer is classified as JavaScript 1.0, since its treatment of parseInt is like Netscape 2, not Netscape 3.
function isInt(numString) {
    if (languageVersion == 1.0) {
        var val = parseInt(numString);
        return((numString.substring(0,1) == "0") ||
                (val != 0));
    } else {
        return(!isNaN(parseInt(numString)));
    }
}

Listing 19.16 illustrates this function, with the results in Netscape 2.02, Netscape 3.01, Netscape 4.01, and Internet Explorer 3.02 shown in Figures 19-28 through 19-31.

Listing 19.16 Numbers.html (continued)

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head>
    <title>Testing Numbers</title>
    <script language="JavaScript">
        <!--
        languageVersion = 1.0;
        // -->
    </script>
    <script language="JavaScript1.1">
        <!--
        if (navigator.appName == "Netscape")
            languageVersion = 1.1;
        else
            languageVersion = 1.0;
        // -->
    </script>
    <script language="JavaScript1.2">
        <!--
        languageVersion = 1.2;
        // -->
    </script>
</head>
<body bgcolor="white">
<script language="JavaScript"> function testInt(numString) {
    return("<TR><TD>" + numString + "<TD>
    + parseInt(numString) + "<TD>
    + isNaN(parseInt(numString)) + "\n    ;
}
</script>
</body>
</html>
Chapter 19 JavaScript: Adding Dynamic Content to Web Pages

19.6 Using JavaScript to Validate CGI Forms

Figure 19-28 The parseInt method returns 0 for illegal numbers in Netscape 2.02.

Figure 19-29 The parseInt method returns NaN for illegal numbers in Netscape 3.01.

Figure 19-30 The parseInt method returns NaN for illegal numbers in Netscape 4.01.

Figure 19-31 The parseInt method returns 0 for illegal numbers in Internet Explorer 3.01.
If a textfield value is supposed to be greater than zero, you can simplify the test considerably by relying on the fact that NaN returns false when compared to any other number. Here is a variation of `isNaN` that uses this idea. It works in Internet Explorer as well as all versions of Netscape.

```javascript
function isInt(string) {
    var val = parseInt(string);
    return (val > 0);
}
```

Listing 19.17 uses this to create a simple input form with three textfields. The first and third should contain numbers and the second should contain a string. Rather than correcting values as they are entered, the only action taken during data entry is to print a descriptive message whenever a textfield has the input focus. However, when the form is submitted, the `checkRegistration` function is invoked. This function verifies that the first and third entries are integers and that the second is neither an integer nor is missing. Results are shown in Figures 19–32 and 19–33.

```html
<!-- DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head>
    <title>Camp Registration</title>
    <script language="JavaScript">
        // -->
        function clearStatus() { status = ""; }
        function promptAge() { status = "Age (no fractions)"; }
        function promptRank() { status = "Rank Name"; }
        function promptSerial() { status = "Serial Number"; }

        // In JavaScript 1.1, parseInt returns NaN (recognizable via isNaN()) for nonintegers. But JavaScript 1.0 returns 0 and doesn't have an isNaN routine. Since comparisons to NaN always fail, the > 0 test works on either version.

        function checkRegistration() {
            var ageField = document.registerForm.ageField;
            if (!isInt(ageField.value)) {
                alert("Age must be an integer.");
                return(false);
            }
            var rankField = document.registerForm.rankField;
            if (!isInt(rankField.value)) {
                alert("Use rank name, not rank number.");
                return(false);
            }
            if (rankField.value == ")") {
                alert("Missing rank.");
                return(false);
            }
            var serialField = document.registerForm.serialField;
            if (!isInt(serialField.value)) {
                alert("Serial number must be an integer.");
                return(false);
            }
            // Format looks OK. Submit form.
            return(true);
        }
    </script>
</head>
<body bgcolor="WHITE">
<html>Camp Registration</html>
```

continued
19.7 Using JavaScript to Store and Examine Cookies

A "cookie" is a small amount of textual information about a page that is stored by the browser on the client system. The structure and syntax of cookies are explained in Section 16.6, and Section 18.12 gives examples of server-side CGI programs manipulating them. They can also be manipulated entirely on the client through the use of the `document.cookie` property. This property behaves in a very unusual fashion. If you look up the value of `document.cookie`, you will get a single big string containing all the cookie values, as sent by the browser via the `Cookie` HTTP request header. For example, if the current page has three cookies `name1`, `name2`, and `name3`, the value of `document.cookie` would be something like

\[
\text{"name1=val1; name2=val2; name3=val3"}
\]

However, you do not assign values to `document.cookie` using a single large string like this. Instead, you specify a single cookie at a time using the same format as would be used in the `Set-Cookie` HTTP response header. See Section 16.6 (Cookies: Storing Persistent Data on the Client) for a complete description of the syntax, but here are a couple of examples.

```javascript
document.cookie = "name1=val1;";
document.cookie = "name2=val2; expires=" + someDate;
document.cookie = "name3=val3; path=/; domain=test.com";
```

Each time `document.cookie` is set, the cookie is stored by the browser. The cookies persist as long as the browser remains open, and if an expiration date is specified, the cookie is reloaded in a later session. This works in JavaScript 1.0, 1.1, and 1.2, but it only works in Internet Explorer when the page is loaded from the network. Local files cannot set cookies, making page