Functions and Custom Objects

CHAPTER 34

In This Chapter

Creating function blocks

Passing parameters to functions

Creating your own objects

By now, you've seen dozens of JavaScript functions in action and probably have a pretty good feel for the way they work. This chapter provides the function object specification and delves into the fun prospect of creating objects in your JavaScript code. That includes objects that have properties and methods, just like the big boys.

Function Object

Properties	*Methods*	*Event Handlers*
arguments	(None)	(None)
arity		
caller		
prototype		

Syntax

Creating a function object:

```
function functionName([arg1,...[,argN]]) {
       statement(s)
}

var funcName = new
Function(["argName1",...[,"argNameN"],
"statement1;...[;statementN]"])

object.eventHandlerName =
function([arg1,...[,argN]]) {statement(s)}
```

Accessing function properties:

```
functionObject.property
```

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	(✔)	✔	✔	(✔)	✔	✔

About this object

JavaScript accommodates what other languages might call procedures, subroutines, and functions all in one type of structure: the *custom function*. A function may return a value (if programmed to do so with the `return` keyword), but it does not have to return any value. Except for JavaScript code that executes as the document loads, all deferred processing takes place in functions.

While you can create functions that are hundreds of lines long, it is advantageous to break up longer processes into shorter functions. Among the reasons for doing so: smaller chunks are easier to write and debug; building blocks make it easier to visualize the entire script; you can make functions generalizable and reusable for other scripts; and other parts of the script or other open frames may be able to use the functions.

Learning how to write good, reusable functions takes time and experience, but the earlier you understand the importance of this concept, the more you will be on the lookout for good examples in other people's scripts on the Web.

Creating functions

The standard way of defining a function in your script means following a simple pattern and then filling in the details. The formal syntax definition for a function is

```
function functionName( [arg1] ... [, argN]) {
       statement(s)
}
```

The task of assigning a function name helps you determine the precise scope of activity of the function. If you find that the planned task for the function can't be reduced to a simple one- to three-word name (which is then condensed into one contiguous sequence of characters for the *functionName*), perhaps you're asking the function to do too much. A better idea may be to break the job into two or more functions. As you start to design a function, you should also be on the lookout for functions that you can call from the one you're writing. If you find yourself copying and pasting lines of code from one part of a function to another because you're performing the same operation in different spots within the function, it may be time to break that segment out into its own function.

Starting with Navigator 3 (and Internet Explorer 3 with JScript.dll Version2), you can also create what is called an *anonymous function* using the `new Function()` constructor. It may be called anonymous, but in fact you assign a name to the function, as follows:

```
var funcName = new Function(["argName1",...[,"argNameN"],
"statement1;...[;statementN]"])
```

It is another way of building a function and is particularly helpful when your scripts need to create a function after a document loads. All the components of a

function are present in this definition. Each function parameter name is supplied as a string value, separated from each other by commas. The final parameter string consists of the statements that execute whenever the function is called. Separate each JavaScript statement with a semicolon, and enclose the entire sequence of statements inside quotes, as in the following:

```
var willItFit = new Function("width","height","var sx =
screen.availWidth; var sy = screen.availHeight; return (sx >= width &&
sy >= height)")
```

The `willItFit()` function takes two parameters; the body of the function defines two local variables (`sx` and `sy`) and then returns a Boolean value if the incoming parameters are smaller than the local variables. In traditional form, this function would be defined as follows:

```
function willItFit(width, height) {
        var sx = screen.availWidth
        var sy = screen.availHeight
        return (sx >= width && sy >= height)
}
```

Once this function exists in the browser's memory, you can invoke it like any other function:

```
if (willItFit(400,500)) {
        statements to load image
}
```

One last function creation format is available in Navigator 4 when you enclose the creation statement in a `<SCRIPT LANGUAGE="JavaScript1.2">` tag set. The advanced technique is called a *lambda expression* and provides a shortcut for creating a reference to an anonymous function (truly anonymous, since the function has no name that can be referenced later). The common application of this technique is to assign function references to event handlers when the event object must also be passed:

```
document.forms[0].age.onchange = function(event)
{isNumber(document.forms[0].age)}
```

Nesting functions

Navigator 4 introduced the ability to nest functions inside one another. In all prior scripting, each function definition is defined at the global level, whereby every function is exposed and available to all other scripting. With nested functions, you can encapsulate the exposure of a function inside another and make that nested function private to the enclosing function. In other words, although it is a form I don't recommend, you could create nested functions with the same name inside multiple global level functions, as the following skeletal structure shows:

```
function outerA() {
        statements
        function innerA() {
           statements
        }
```

```
        statements
}
function outerB() {
        statements
        function innerA() {
            statements
        }
        function innerB() {
            statements
        }
        statements
}
```

A nested function can be accessed only from statements in its containing function. Moreover, all variables defined in the outer function (including parameter variables) are accessible to the inner function; but variables defined in an inner function are not accessible to the outer function. See "Variable Scope: Globals and Locals" later in this chapter for details on how variables are visible to various components of a script.

Function parameters

The function definition requires a set of parentheses after the *functionName*. If the function does not rely on any information arriving with it when invoked, the parentheses can be empty. But when some kind of data will be coming with a call to the function, you need to assign names to each parameter. Virtually any kind of value can be a parameter: strings, numbers, Booleans, and even complete object references, such as a form or form element. Choose names for these variables that help you remember the content of those values; also avoid reusing existing object names as variable names, because it's easy to get confused when objects and variables with the same name appear in the same statements. You must avoid using JavaScript keywords (including the reserved words listed in Appendix B) and any global variable name defined elsewhere in your script (see more about global variables in following sections).

JavaScript is forgiving about matching the number of parameters in the function definition with the number of parameters passed along from the calling statement. If you define a function with three parameters and the calling statement only specifies two, the third parameter variable value in that function is assigned a null value. For example:

```
function oneFunction(a, b, c) {
        statements
}
oneFunction("George","Gracie")
```

In the preceding example, the values of `a` and `b` inside the function are "George" and "Gracie," respectively; the value of `c` is null.

At the opposite end of the spectrum, JavaScript also won't balk if you send more parameters from the calling statement than the number of parameter variables specified in the function definition. In fact, the language includes a mechanism — the `arguments` property — that you can add to your function to gather any extraneous parameters that should read your function.

Properties

arguments

Value: Array of arguments **Gettable:** Yes **Settable:** No

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility		✔	✔		✔	✔

When a function receives parameter values from the statement that invokes the function, those parameter values are silently assigned to the `arguments` property of the function object. The property is an array of the values, with each parameter value assigned to a zero-based index entry in the array—whether or not parameters are defined for it (and in Navigator 4, the property is a first-class object). You can find out how many parameters were sent by extracting `functionName.arguments.length`. For example, if four parameters were passed, `functionName.arguments.length` returns 4. Then use array notation (`functionName.arguments[i]`) to extract the values of any parameter(s) you want.

Theoretically, you never have to define parameter variables for your functions, extracting the desired `arguments` array entry instead. Well-chosen parameter variable names, however, are much more readable, so I recommend them over the `arguments` property for most cases. But you may run into situations in which a single function definition needs to handle multiple calls to the function when each call may have a different number of parameters. The function knows how to handle any arguments over and above the ones given names as parameter variables.

See Listings 34-1 and 34-2 for a demonstration of both the `arguments` and `caller` properties.

arity

Value: Integer **Gettable:** Yes **Settable:** No

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✔			

As the `arguments` property of a function proves, JavaScript is very forgiving about matching the number of parameters passed to a function with the number of parameter variables defined for the function. But a script can examine the `arity` property of a function to see precisely how many parameter variables are defined for a function. A reference to the property starts with the function name representing the object. For example, consider the following function definition shell:

```
function identify(name, rank, serialNum) {
        ...
}
```

A script statement anywhere outside of the function can read the number of parameters with the reference

```
identify.arity
```

The value of the property in the preceding example is 3.

caller

Value: Function **Gettable:** Yes **Settable:** No

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility		✔	✔		✔	✔

When one function invokes another, a chain is established between the two, primarily so that a returned value knows where to go. Therefore, a function invoked by another maintains a reference back to the function that called it. Such information is automatically stored in a function object as the `caller` property. This relationship reminds me a bit of a subwindow's `opener` property, which points back to the window or frame responsible for the subwindow's creation. The value is valid only while the called function is running at the request of another function; when a function isn't running, its `caller` property is null.

Since the value of the `caller` property is a function object, you can inspect its `arguments` and `caller` properties (in case it was called by yet another function). Thus, a function can look back at a calling function to see what values it was passed.

The `functionName.caller` property reveals the contents of an entire function definition if the current function was called from another function (including an event handler). If the call for a function comes from a regular JavaScript statement (such as in the Body as the document loads), the `functionName.caller` property is null.

To help you grasp all that these two properties yield, study Listing 34-1.

Listing 34-1: **A Function's arguments and caller Properties**

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function hansel(x,y) {
        var args = hansel.arguments
        document.write("hansel.caller is " + hansel.caller + "<BR>")
        document.write("hansel.arguments.length is " +
hansel.arguments.length + "<BR>")
        document.write("formal x is " + hansel.x + "<BR>")
        for (var i = 0; i < args.length; i++) {
           document.write("argument " + i + " is " + args[i] + "<BR>")
        }
        document.write("<P>")
```

```
}

function gretel(x,y,z) {
        today = new Date()
        thisYear = today.getYear()
        hansel(x,y,z,thisYear)
}
</SCRIPT>
</HEAD>
<BODY>
<SCRIPT LANGUAGE="JavaScript">
hansel(1, "two", 3);
gretel(4, "five", 6, "seven");
</SCRIPT>
</BODY>
</HTML>
```

When you load this page, the following results appear in the browser window:

```
hansel.caller is null
hansel.arguments.length is 3
formal x is 1
argument 0 is 1
argument 1 is two
argument 2 is 3

hansel.caller is function gretel(x, y, z) { today = new Date();
thisYear = today.getYear(); hansel(x, y, z, thisYear); }

hansel.arguments.length is 4
formal x is 4
argument 0 is 4
argument 1 is five
argument 2 is 6
argument 3 is 97 (or whatever the current year is)
```

As the document loads, the `hansel()` function is called directly in the Body script. It passes three arguments, even though the `hansel()` function defines only two. The `hansel.arguments` property picks up all three arguments, just the same. The main Body script then invokes the `gretel()` function, which, in turn, calls `hansel()` again. But when `gretel()` makes the call, it passes four parameters. The `gretel()` function picks up only three of the four arguments sent by the calling statement. It also inserts another value from its own calculations as an extra parameter to be sent to `hansel()`. The `hansel.caller` property reveals the entire content of the `gretel()` function, whereas `hansel.arguments` picks up all four parameters, including the year value introduced by the `gretel()` function.

If you have Navigator 4, you should also try Listing 34-2, which better demonstrates the chain of `caller` properties through a sequence of invoked functions. A click of the button in the page invokes a simple function named `first()`. The passed parameter is the button object reference. The `first()` function in turn invokes the `middle()` function, passing a string identifying its source as the

first function. Finally, the `middle()` function invokes the `last()` function, passing along the parameter it received from `first()`, plus two other string parameters. The `last()` function defines parameter variables for only two of the incoming parameters.

An examination of the properties for the arguments object in `last()` reveals a total of three elements — the three parameters. The index values for the first two consist of the parameter variable names, while the third parameter is assigned to the slot indexed with 2 (the third slot in the zero-based counting system). From within the `last()` function, a statement grabs the `arguments` property of the `caller` (the `middle()` function), whose only entry is the one incoming parameter to that function (`firstMsg`). And finally, an examination of the first function in the chain (via the `caller.caller` reference) finds that its `arguments` property consists of the one entry of the button reference passed from the event handler.

Listing 34-2: **Examining Arguments through Three Generations**

```
<HTML>
<HEAD>
<TITLE>Event.which Properties</TITLE>
<SCRIPT LANGUAGE="JavaScript1.2">
function showProps(objName,obj) {
        var msg = ""
        for (var i in obj) {
           msg += objName + "." + i + "=" + obj[i] + "\n"
        }
        return msg
}
function first(btn) {
        middle("1st Function Parameter")
}
function middle(firstMsg) {
        last(firstMsg, "2nd Function Parameter", "Bonus Param")
}
function last(firstMsg, secondMsg) {
        var thirdMsg = "Var in 3rd Function"
        var form = document.output
        form.lastFuncArgs.value = showProps("last.arguments",
last.arguments)
        form.midFuncArgs.value = showProps("caller.arguments",
caller.arguments)
        form.firstFuncArgs.value = showProps("caller.caller.arguments",
caller.caller.arguments)
}
</SCRIPT>
</HEAD>
<BODY>
<B>Function Properties</B>
<HR>
Click on the button to trigger a three-function ripple. The effects
are shown in the fields below</P>
<FORM NAME="output">
```

```
<INPUT TYPE="button" VALUE="Trigger and Show" onClick="first(this)"><BR>
last.arguments:<BR>
<TEXTAREA NAME="lastFuncArgs" COLS=70 ROWS=3></TEXTAREA><BR>
middle.arguments:<BR>
<TEXTAREA NAME="midFuncArgs" COLS=70 ROWS=2></TEXTAREA><BR>
first.arguments:<BR>
<TEXTAREA NAME="firstFuncArgs" COLS=70 ROWS=2
WRAP="virtual"></TEXTAREA><BR>
</FORM>
</BODY>
</HTML>
```

These are powerful and useful properties of functions, but I recommend that you not rely on them for your normal script operations unless you fully understand their inner workings. You should be defining functions that take into account all the possible parameters that could be sent by other calling functions. I do, however, use these properties as debugging aids when working on complex scripts that have many calls to the same function.

prototype

Value: String or Function **Gettable:** Yes **Settable:** Yes

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility		✔	✔		✔	✔

Like a number of JavaScript objects, the function object has a `prototype` property, which enables you to apply new properties and methods to every function object that is created in the current page. You can see examples of how this works in discussions of the `prototype` property for string and array objects (Chapters 26 and 29, respectively).

Function Application Notes

Understanding the ins and outs of JavaScript functions is key to successful scripting, especially for complex applications. Additional topics to be covered in this chapter include the ways to invoke functions, variable scope in and around functions, recursion, and designing reusable functions.

Invoking Functions

A function doesn't perform any work until a script calls it by name. Scripts invoke functions (that is, get functions doing something) via three routes: JavaScript object event handlers; javaScript statements; and `HREF=` attributes pointing to a `javascript:` URL.

Because you've seen dozens of examples of the first two methods throughout this book so far, let me say a few words about the last item.

Several HTML tags have HREF attributes that normally point to Internet URLs for either navigating to another page or loading a MIME file that requires a helper application or plug-in. These HTML tags are usually tags for clickable objects, such as links and client-side image map areas.

A JavaScript-enabled browser has a special built-in URL pseudo-protocol — javascript: — that lets the HREF attribute point to a JavaScript function or method, rather than to a URL out on the Net. For example, I use the javascript: URL when I want a link to change the contents of two other frames. Because the HREF attribute enables me to specify only a single URL, I'd be out of luck without a convenient way to put multiframe navigation into my hands. I do that by writing a function that sets the location properties of the two frames; then I invoke that function from the HREF attribute. The following example shows what the script may look like:

```
function loadPages() {
        parent.frames[1].location = "page2.html"
        parent.frames[2].location = "instrux2.html"
}
...
<A HREF="javascript:loadPages()">Next</A>
```

These kinds of function invocations can include parameters, and the functions can do anything you want. One potential side effect to watch out for occurs when the function returns a value (perhaps the function is also invoked from other script locations where a returned value is expected). Because the HREF attribute sets the TARGET window to whatever the attribute evaluates to, the returned value will be assigned to the TARGET window — probably not what you want.

To prevent the assignment of a returned value to the HREF attribute, prefix the function call with the void operator (you can also surround the function call with void()). The placement of this operator is critical. The following are two examples of how to use void:

```
<A HREF="javascript:void loadPages()">
<A HREF="javascript:void(loadPages())">
```

Experienced programmers of many other languages will recognize this operator as a way of indicating that no values are returned from a function or procedure. The operator has precisely that functionality here, but in a nontraditional location.

Variable Scope: Globals and Locals

A variable can have two scopes in JavaScript. As you'd expect, any variable initialized within the main flow of a script (not inside a function) is a *global variable*, in that any statement in the same document's script can access it by name. You can, however, also initialize variables inside a function (in a var statement) so the variable name applies only to statements inside that function. By limiting the scope of the variable to a single function, you can reuse the same

variable name in multiple functions, enabling the variables to carry very different information in each function. To demonstrate the various possibilities, I present the script in Listing 34-3.

Listing 34-3: **Variable Scope Workbench Page**

```
<HTML>
<HEAD>
<TITLE>Variable Scope Trials</TITLE>
<SCRIPT LANGUAGE="JavaScript">
var headGlobal = "Gumby"
function doNothing() {
        var headLocal = "Pokey"
        return headLocal
}
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT LANGUAGE="JavaScript">
// two global variables
var aBoy = "Charlie Brown"
var hisDog = "Snoopy"
function testValues() {
        var hisDog = "Gromit"  // initializes local version of "hisDog"
        var page = ""
        page += "headGlobal is: " + headGlobal + "<BR>"
        // page += "headLocal is: " + headLocal + "<BR>" // won't run:
headLocal not defined
        page += "headLocal value returned from head function is: " +
doNothing() + "<P>"
        page += " aBoy is: " + aBoy + "<BR>" // picks up global
        page += "local version of hisDog is: " + hisDog + "<P>" //
"sees" only local version
        document.write(page)
}
testValues()
document.write("global version of hisDog is intact: " + hisDog)
</SCRIPT>
</BODY>
</HTML>
```

In this page, you define a number of variables — some global, others local — that are spread out in the document's Head and Body sections. When you load this page, it runs the `testValues()` function, which accounts for the current values of all the variable names. The script then follows up with one more value extraction that was masked in the function. The results of the page look like this:

```
headGlobal is: Gumby
headLocal value returned from head function is: Pokey
```

```
aBoy is: Charlie Brown
local version of hisDog is: Gromit

global version of hisDog is intact: Snoopy
```

Examine the variable initialization throughout this script. In the Head, you define the first variable, `headGlobal`, as a global style — outside of any function definition. The `var` keyword for the global variable is optional but often helpful for enabling you to see at a glance where you initialize your variables. You then create a short function, which defines a variable (`headLocal`) that only statements in the function can use.

In the Body, you define two more global variables, `aBoy` and `hisDog`. Inside the Body's function, I intentionally (for purposes of demonstration) have you reuse the `hisDog` variable name. By initializing `hisDog` with the `var` statement inside the function, you tell JavaScript to create a separate variable whose scope is only within the function. This initialization does not disturb the global variable of the same name. It can, however, make things confusing for you as script author.

Statements in the script attempt to collect the values of variables scattered around this script. Even from within this script, JavaScript has no problem extracting global variables directly, including the one defined in the Head. But it cannot get the local variable defined in the other function — that `headLocal` variable is private to its own function. Trying to run a script that gets that variable value results in an error message saying that the variable name is not defined. In the eyes of everyone else outside of the `doNothing()` function, that's true. If you really need that value, you can have that function return the value to a calling statement, as you do in the `testValues()` function.

Near the end of the function, you get the `aBoy` global value without a hitch. But because you initialized a separate version of `hisDog` inside that function, only the localized version is available to the function. If you reassign a global variable name inside a function, you cannot access the global version from inside that function.

As proof that the global variable, whose name was reused inside the `testValues()` function, remains untouched, the script writes that value to the end of the page for all to see. Charlie Brown and his dog are reunited.

A benefit of this variable-scoping scheme is that you can reuse "throw-away" variable names in any function you like. For instance, you are free to use, say, the `i` loop counting variable in every function that uses loops (in fact, you can reuse it in multiple `for` loops of the same function, because the `for` loop reinitializes the value at the start of the loop). If you pass parameters to a function, you can assign those parameters the same names to aid in consistency. For example, a common practice is to pass an entire form object reference as a parameter to a function (using a `this.form` parameter in the event handler). For every function that catches one of these objects, you can use the variable name `form` in the parameter, as in

```
function doSomething(form) {
        statements
}
...
<INPUT TYPE="button" VALUE="Do Something"
onClick="doSomething(this.form)">
```

If five buttons on your page pass their form objects as parameters to five different functions, each function can assign `form` (or whatever you want to use) to that parameter value.

I recommend reusing variable names only for these "throw-away" variables. In this case, the variables are all local to functions, so the possibility of a mix-up with global variables does not exist. But the thought of reusing a global variable name as, say, a special case inside a function sends shivers up my spine. Such a tactic is doomed to cause confusion and error.

Some programmers devise naming conventions for themselves to avoid reusing global variables as local variables. A popular scheme puts a lowercase "g" in front of any global variable name. In the example from Listing 34-3, the global variables would have been named

```
gHeadGlobal
gABoy
gHisDog
```

Then if you define local variables, don't use the leading "g." Any scheme you use to prevent the reuse of variable names in different scopes is fine as long as it does the job.

In a multiple-frame or multiple-window environment, your scripts can also access global variables from any other document currently loaded into the browser. For details about this level of access, see Chapter 14.

Variable scoping rules apply equally to nested functions in Navigator 4. Any variables defined in an outer function (including parameter variables) are exposed to all functions nested inside. But if you define a new local variable inside a nested function, that variable is not available to the outer function. Instead, you can return a value from the nested function to the statement in the outer function that invokes the nested function.

Parameter variables

When a function receives data in the form of parameters, remember that the values may be merely copies of the data (in the case of run-of-the-mill data values) or references to real objects (such as a form object). In the latter case, you can change the object's modifiable properties in the function when the function receives the object as a parameter, as shown in the following example:

```
function validateCountry (form) {
        if (form.country.value == "") {
            form.country.value = "USA"
        }
}
```

JavaScript knows all about the form object passed to the `validateCountry()` function. Therefore, whenever you pass an object as a function parameter, be aware that the changes you make to that object in its "passed" form affect the real object.

As a matter of style, if my function needs to extract properties or results of methods from passed data (such as object properties or string substrings), I like to do that at the start of the function. I initialize as many variables as needed for each piece of data used later in the function. This task enables me to assign meaningful

names to the data chunks, rather than having to rely on potentially long references within the working part of the function (such as using a variable like `inputStr` instead of `form.entry.value`).

Recursion in functions

Functions can call themselves — a process known as *recursion*. The classic example of programmed recursion is the calculation of the factorial (the factorial for a value of 4 is 4 * 3 * 2 * 1), shown in Listing 34-4 (not on the CD-ROM).

In the third line of this function, the statement calls itself, passing along a parameter of the next lower value of `n`. As this function executes, diving ever deeper into itself, JavaScript watches intermediate values and performs the final evaluations of the nested expressions. Be sure to test any recursive function carefully. In particular, make sure that the recursion is finite: That a limit exists for the number of times it can recurse. In the case of Listing 34-4, that limit is the initial value of `n`. Failure to watch out for this limit may cause the recursion to overpower the limits of the browser's memory and even lead to a crash.

Listing 34-4: **A JavaScript Function Utilizing Recursion**

```
function factorial(n) {
        if (n > 0) {
            return n * (factorial(n-1))
        } else {
            return 1
        }
}
```

Turning functions into libraries

As you start writing functions for your scripts, be on the lookout for ways to make functions generalizable (written so that you can reuse the function in other instances, regardless of the object structure of the page). The likeliest candidates for this kind of treatment are functions that perform specific kinds of validation checks (see examples in Chapter 37), data conversions, or iterative math problems.

To make a function generalizable, don't let it make any references to specific objects by name. Object names will probably change from document to document. Instead, write the function so that it accepts a named object as a parameter. For example, if you write a function that accepts a text object as its parameter, the function can extract the object's data or invoke its methods without knowing anything about its enclosing form or name. Look again, for example, at the `factorial()` function in Listing 34-5 — but now as part of an entire document.

Listing 34-5: **Calling a Generalizable Function**

```
<HTML>
<HEAD>
<TITLE>Variable Scope Trials</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function factorial(n) {
        if (n > 0) {
           return n * (factorial(n - 1))
        } else {
           return 1
        }
}
</SCRIPT>
</HEAD>

<BODY>
<FORM>
Enter an input value: <INPUT TYPE="text" NAME="input" VALUE=0><P>
<INPUT TYPE="button" VALUE="Calc Factorial"
        onClick="this.form.output.value =
           factorial(this.form.input.value)"><P>
Results: <INPUT TYPE="text" NAME="output">
</FORM>
</BODY>
</HTML>
```

The function was designed to be generalizable, accepting only the input value (`n`) as a parameter. In the form, the `onClick=` event handler of the button extracts the input value from one of the form's fields, sending that value to the `factorial()` function. The returned value is assigned to the output field of the form. The `factorial()` function is totally ignorant about forms, fields, or buttons in this document. If I need this function in another script, I can copy and paste it into that script, knowing that it has been pretested. Any generalizable function becomes part of my personal library of scripts — from which I can borrow — and saves me time in future scripting tasks.

You will not always be able to generalize a function. Somewhere along the line in your scripts, you must have references to JavaScript or custom objects. But if you find that you're frequently writing functions that perform the same kind of actions, it's time to see how you can generalize the code and put the results in your library of ready-made functions. And if your audience is using browsers from Navigator 3 onward (and later versions of Internet Explorer 3 onward), consider placing these library functions in an external .js library file. See Chapter 13 for details on this convenient way to share utility functions among many documents.

Custom Objects

In all the previous chapters of this book, you've seen how conveniently the JavaScript document object model organizes all the information about the browser window and its document. What may not be obvious from the scripting you've done so far is that JavaScript enables you to create your own objects in memory — objects with properties and methods. These objects are not user interface elements per se on the page, but rather the kinds of objects that may contain data and script functions (behaving as methods), whose results the user can see displayed in the browser window.

You actually had a preview of this power in Chapter 29's discussion about arrays. An array, you recall, is an ordered collection of data. You can create a JavaScript array in which entries are labeled just like properties that you access via the now-familiar dot syntax (`arrayName[index].propertyName`). An object typically contains different kinds of data. It doesn't have to be an ordered collection of data — although your scripts can use objects in constructions that strongly resemble arrays. Moreover, you can attach any number of custom functions as methods for that object. You are in total control of the object's structure, data, and behavior.

An example — planetary objects

Building on your familiarity with the planetary data array created in Chapter 29, in this chapter I have you use the same information to build objects. The application goals are the same: Present a pop-up list of the nine planets of the solar system and display data about the selected planet. From a user interface perspective (and for more exposure to multiframe environments), the only difference is that the resulting data displays in a separate frame of a two-frame window rather than in a textarea object. This means your object method will be building HTML on the fly and plugging it into the display frame — a pretty typical task in these multiframe, Web-browsing days.

To recap the array style from Chapter 29: You created a two-dimensional array — a nine-row, five-column table of data about the planets. Each row was an entry in the `solarSys[]` array. For a function to extract and display data about a given planet, it needed the index value of the `solarSys[]` array passed as a parameter, so that it could get whatever property it needed for that entry (such as `solarSys[3]. name`).

In this chapter, instead of building arrays, you build objects — one object for each planet. The design of your object has five properties and one method. The properties are the same ones you used in the array version: name, diameter, distance from the sun, year length, and day length. To give these objects more intelligence, you give each of them the capability to display their data in the lower frame of the window. You can conveniently define one function that knows how to behave with any of these planet objects, rather than having to define nine separate functions.

Listing 34-6 shows the source code for the document that creates the frameset for your planetary explorations; Listing 34-7 shows the entire HTML page for the object-oriented planet document, which appears in the top frame.

Listing 34-6: Framesetting Document for a Two-Frame Window

```
<HTML>
<HEAD>
<TITLE>Solar System Viewer</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function blank() {
        return "<HTML><BODY></BODY></HTML>"
}
</SCRIPT>
</HEAD>
<FRAMESET ROWS="50%,50%">
        <FRAME NAME="Frame1" SRC="lst34-07.htm">
        <FRAME NAME="Frame2" SRC="javascript:parent.blank()">
</FRAMESET>
</HTML>
```

One item to point out in Listing 34-6 is that because the lower frame doesn't get filled until the upper frame's document loads, you need to assign some kind of URL for the `SRC=` attribute of the second frame. Rather than add the extra transaction and file burden of a blank HTML document, here you use the `javascript:` URL to invoke a function. In this instance, I want the value returned from the function (a blank HTML page) to be reflected into the target frame (no `void` operator here). This method is the most efficient way of creating a blank frame in a frameset.

Listing 34-7: Object-Oriented Planetary Data Presentation

```
<HTML>
<HEAD>
<TITLE>Our Solar System</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!-- start script
// method definition
function showPlanet() {
        var result = "<HTML><BODY><CENTER><TABLE BORDER=2>"
        result += "<CAPTION ALIGN=TOP>Planetary data for: <B>" +
this.name + "</B></CAPTION>"
        result += "<TR><TD ALIGN=RIGHT>Diameter:</TD><TD>" +
this.diameter + "</TD></TR>"
        result += "<TR><TD ALIGN=RIGHT>Distance from Sun:</TD><TD>" +
this.distance + "</TD></TR>"
        result += "<TR><TD ALIGN=RIGHT>One Orbit Around Sun:</TD><TD>"
+ this.year + "</TD></TR>"
        result += "<TR><TD ALIGN=RIGHT>One Revolution (Earth
Time):</TD><TD>" + this.day + "</TD></TR>"
        result += "</TABLE></CENTER></BODY></HTML>"
        // display results in a second frame of the window
```

(continued)

Listing 34-7 *(continued)*

```
        parent.Frame2.document.write(result)
        parent.Frame2.document.close()
}

// definition of planet object type;
// 'new' will create a new instance and stuff parameter data into
object
function planet(name, diameter, distance, year, day) {
        this.name = name
        this.diameter = diameter
        this.distance = distance
        this.year = year
        this.day = day
        this.showPlanet = showPlanet  // make showPlanet() function a
method of
        // planet
}

// create new planet objects, and store in a series of variables
Mercury = new planet("Mercury","3100 miles", "36 million miles", "88
days", "59 days")
Venus = new planet("Venus", "7700 miles", "67 million miles", "225
days", "244 days")
Earth = new planet("Earth", "7920 miles", "93 million miles", "365.25
days","24 hours")
Mars = new planet("Mars", "4200 miles", "141 million miles", "687
days", "24 hours, 24 minutes")
Jupiter = new planet("Jupiter","88,640 miles","483 million miles",
"11.9 years", "9 hours, 50 minutes")
Saturn = new planet("Saturn", "74,500 miles","886 million miles", "29.5
years", "10 hours, 39 minutes")
Uranus = new planet("Uranus", "32,000 miles","1.782 billion miles","84
years", "23 hours")
Neptune = new planet("Neptune","31,000 miles","2.793 billion
miles","165 years", "15 hours, 48 minutes")
Pluto = new planet("Pluto", "1500 miles", "3.67 billion miles", "248
years", "6 days, 7 hours")

// end script -->
</SCRIPT>
<BODY>
<H1>The Daily Planet</H1>
<HR>
<FORM>
<SCRIPT LANGUAGE = "JavaScript">
<!-- start script again
var page = "" // start assembling next part of page and form
page += "Select a planet to view its planetary data: "
// build popup list from planet object names
page += "<SELECT NAME='planets' onChange='doDisplay(this)'> "
```

```
page += "<OPTION>Mercury"
page += "<OPTION>Venus"
page += "<OPTION SELECTED>Earth"
page += "<OPTION>Mars"
page += "<OPTION>Jupiter"
page += "<OPTION>Saturn"
page += "<OPTION>Uranus"
page += "<OPTION>Neptune"
page += "<OPTION>Pluto"
page += "</SELECT><P>"  // close selection item tag
document.write(page)    // lay out this part of the page

// called from push button to invoke planet object method
function doDisplay(popup) {
        i = popup.selectedIndex
        eval(popup.options[i].text + ".showPlanet()")
}
doDisplay(document.forms[0].planets)
// really end script -->
</SCRIPT>
</FORM>
</BODY>
</HTML>
```

The first task in the Head is to define the function that becomes a method in each of the objects. You must do this task before scripting any other code that adopts the function as its method. Failure to define the function ahead of time results in an error — the function name is not defined. If you compare the data extraction methodology to the function in the array version, you will notice that not only is the parameter for the index value gone, but the reference to each property begins with `this`. I come back to the custom method after giving you a look at the rest of the Head code.

Next comes the object constructor function, which performs several important tasks. For one, everything in this function establishes the structure of your custom object: the properties available for data storage and retrieval and any methods that the object can invoke. The name of the function is the name you will use later to create new instances of the object. Therefore, choosing a name that truly reflects the nature of the object is important. And, because you will probably want to stuff some data into the function's properties to get one or more instances of the object loaded and ready for the page's user, the function definition includes parameters for each of the properties defined in this object definition.

Inside the function, you use the `this` keyword to assign data that comes in as parameters to labeled properties. For this example, I've decided to use the same names for both the incoming parameter variables and the properties. That's primarily for convenience, but you can assign any variable and property names you want and connect them any way you like. In the `planet()` constructor function, five property slots are reserved for every instance of the object whether or not any data actually gets in every property (if not, its value is null).

The last entry in the `planet()` constructor function is a reference to the `showPlanet()` function defined earlier. Notice that the assignment statement doesn't refer to the function with its parentheses — just to the function name. When JavaScript sees this assignment statement, it looks back through existing definitions (those functions defined ahead of the current location in the script) for a match. If it finds a function (as it does here), it knows to assign the function to the identifier on the left side of the assignment statement. In doing this task with a function, JavaScript automatically sets up the identifier as a method name for this object. As you do in every JavaScript method you've encountered, you must invoke a method by using a reference to the object, a period, and the method name followed by a set of parentheses. You see that syntax in a minute.

The next long block of statements creates the individual objects according to the definition established in the `planet()` constructor. Notice that like an array, an object is created by an assignment statement and the keyword `new`. I've assigned names that are not only the real names of planets (the Mercury object name is the Mercury planet object), but that will also come in handy later when extracting names from the pop-up list in search of a particular object's data.

The act of creating a new object sets aside space in memory (associated with the current document) for this object and its properties. In this script, you're creating nine object spaces, each with a different set of properties. Notice that no parameter is being sent (or expected at the function) that corresponds to the `showPlanet()` method. Omitting that parameter here is fine, because the specification of that method in the object definition means that script automatically attaches the method to every version (instance) of the planet object that the script creates.

In the Body portion of the document and after the page's headline, you use JavaScript to create the rest of the user interface for the top frame of the browser window. I've replaced the array version's `for` loop for the pop-up list content with a hard-wired approach. The task could have been accomplished in fewer statements, but I set it up so that if I modify or borrow this code for another purpose, the hard-wired strings will be easier to locate, select, and change.

After the HTML for the top frame is assembled, it is written for the first time (`document.write()`) to let the user see what's going on. Another function is included (`doDisplay()`) as an intermediary between the select object event handler and the `showPlanet()` method for the selected item.

The `onChange=` event handler in the select list passes a copy of the form's selection object to the `doDisplay()` function. In that function, the select object is assigned to a variable called `popup` to help you visualize that the object is the pop-up list. The first statement extracts the index value of the selected item. Using that index value, the script extracts the text. But things get a little tricky because you need to use that text string as a variable name — the name of the planet — and append it to the call to the `showPlanet()` method. To make the disparate data types come together, you use the `eval()` function. Inside the parentheses, you extract the string for the planet name and concatenate a string that completes the reference to the object's `showPlanet()` method. The `eval()` function evaluates that string, which turns it into a valid method call. Therefore, if the user selects Jupiter from the pop-up list, the method call becomes `Jupiter.showPlanet()`.

Now it's time to look back to the `showPlanet()` function/method definition at the top of the script. By the time this method starts working, JavaScript has

already been to the object whose name is selected in the pop-up list — the Jupiter object, for example. That Jupiter object has the `showPlanet()` method as part of its definition. When that method runs, its only scope is of the Jupiter object. Therefore, all references to `this.propertyName` in `showPlanet()` refer to Jupiter only. The only possibility for `this.name` in the Jupiter object is the value assigned to the `name` property for Jupiter. The same goes for the rest of the properties extracted in the function/method.

One final note about user interface for this admittedly minimal application (see Figure 34-1). At the bottom of the Body's script is the following statement:

```
doDisplay(document.forms[0].planets)
```

The purpose of this statement is to draw the lower-frame table of data to match the preselected item in the select object when the page initially loads. Without taking this extra step, the user would be faced with a selection showing and no data appearing in the lower frame. If the user selected the preselected item, no change event would register with the select object, and no data would appear in the lower frame. The user may think that the page or script was broken. To avoid this problem, a good practice is to look at your pages with the fresh eye of a new user when scripting new concepts.

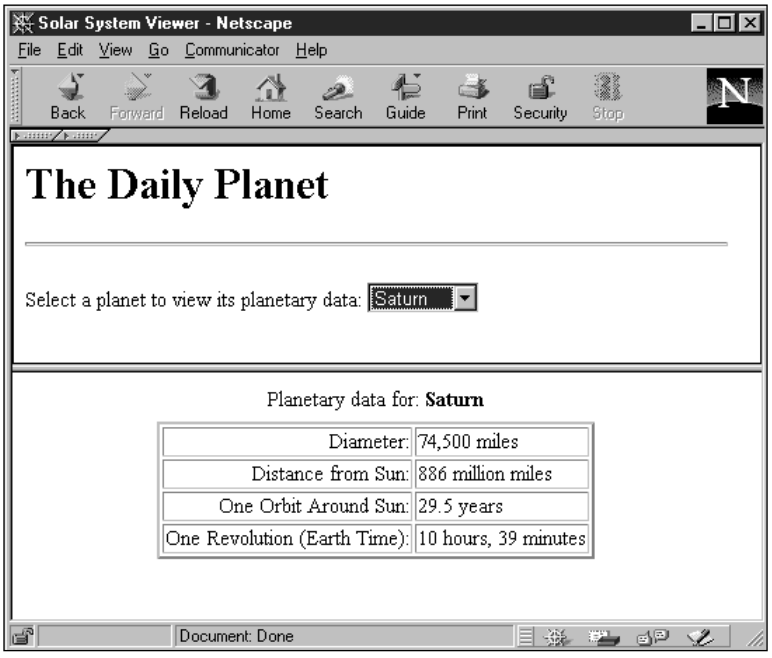

Figure 34-1: An external and internal face-lift for an earlier application

Adding a custom method

You're getting to quite advanced subject matter at this point, so I merely mention and briefly demonstrate an additional power of defining and using custom objects. A custom object can have another custom object as a property. Let's extend the planet example to help you understand the implications.

Say that you want to beef up the planet page with an image of each planet. Each image has a URL for the image file plus other information, such as the copyright notice and a reference number, both of which display on the page for the user. One way to handle this additional information is to create a separate object definition for an image database. Such a definition may look like this:

```
function image(name, URL, copyright, refNum) {
        this.name = name
        this.URL = URL
        this.copyright = copyright
        this.refNum = refNum
}
```

You then need to create individual image objects for each picture. One such definition may look like this:

```
mercuryImage = new image("Planet Mercury", "/images/merc44.gif",
"(c)1990 NASA", 28372)
```

Attaching an image object to a planet object requires modifying the planet constructor function to accommodate one more property. The new planet constructor looks like this:

```
function planet(name, diameter, distance, year, day, image) {
        this.name = name
        this.diameter = diameter
        this.distance = distance
        this.year = year
        this.day = day
        this.showPlanet = showPlanet
        this.image = image // add image property
}
```

Once the image objects are created, you can then create the planet objects, passing one more parameter — an image object you want associated with that object:

```
// create new planet objects, and store in a series of variables
Mercury = new planet("Mercury","3100 miles",  "36 million miles",
"88 days",    "59 days",  mercuryImage)
```

To access a property of an image object, your scripts then have to assemble a reference that works its way through the connection with the planet object:

```
copyrightData = Mercury.image.copyright
```

The potential of custom objects of this type is enormous. For example, you could embed all the copy elements and URL images for an online catalog in a single

document. As the user selects items to view (or cycles through them in sequence), a new JavaScript-written page displays the information in an instant, only requiring the image to be downloaded (unless the image was precached, as described in the `document.image` object discussion in Chapter 18, in which case everything works instantaneously — no waiting for page after page of catalog).

If, by now, you think you see a resemblance between this object-within-an-object construction and a relational database, give yourself a gold star. Nothing prevents multiple objects from having the same subobject as their properties — like multiple business contacts having the same company object property.

More ways to create objects

The examples in Listings 34-6 and 34-7 show a way of creating objects that works with all scriptable browsers. If your audience is limited to users with more modern browsers, additional ways of creating custom objects exist.

From Navigator 3 onward and Internet Explorer 4 onward, you can use the `new Object()` constructor to generate a blank object. From that point on, you can define property and method names by simple assignment, as in the following:

```
var Earth = new Object()
Earth.diameter = "7920 miles"
Earth.distance = "93 million miles"
Earth.year = "365.25"
Earth.day = "24 hours"
Earth.showPlanet = showPlanet  // function reference
```

When you are creating a lot of like-structured objects, the custom object constructor shown in Listing 34-7 is more efficient. But for single objects, the `new Object()` constructor is more efficient.

Navigator 4 users can also benefit from a shortcut literal syntax for creating a new object. Pairs of property names and their values can be set inside a set of curly braces, and the whole construction assigned to a variable that becomes the object name. The following script shows how to organize this kind of object constructor:

```
var Earth = {diameter:"7920 miles", distance:"93 million miles",
year:"365.25", day:"24 hours", showPlanet:showPlanet}
```

Name-value pairs are linked together with colons, and multiple name-value pairs are separated by commas. All in all, this is a very compact construction, well suited for single object construction.

Object watcher methods

Navigator 4 includes two special functions for objects that were designed primarily for use with external debugging tools (such as Netscape's JavaScript Debugger, described in Chapter 46). The methods are `watch()` and `unwatch()`. The `watch()` method instructs JavaScript to keep an eye on a particular property in an object (any JavaScript-accessible object) and execute a function when the value of the property changes by assignment (that is, not by user interaction).

You can see how this works in the simplified example of Listing 34-8. Three buttons set the `value` property of a text box. You can turn on the `watch()` method, in which case it calls a handler and passes the name of the property, the old value, and the new value. An alert in the function in the listing demonstrates what those values contain.

Listing 34-8: **Object Watching in Navigator 4**

```
<HTML>
<HEAD>
<TITLE>Our Solar System</TITLE>
<SCRIPT LANGUAGE="JavaScript1.2">
function setIt(msg) {
        document.forms[0].entry.value = msg
}
function watchIt(on) {
        var obj = document.forms[0].entry
        if (on) {
           obj.watch("value",report)
        } else {
           obj.unwatch("value")
        }
}
function report(id, oldval, newval) {
        alert("The field's " + id + " property on its way from \n'" +
oldval + "'\n to \n'" + newval + "'.")
        return newval
}
</SCRIPT>
<BODY>
<B>Watching Over You</B>
<HR>
<FORM>
Enter text here:
<INPUT TYPE="text" NAME="entry" SIZE=50 VALUE="Default Value"><P>
<INPUT TYPE="button" VALUE="Set to Phrase I" onClick="setIt('Four score
and seven years ago...')"><BR>
<INPUT TYPE="button" VALUE="Set to Phrase 2" onClick="setIt('When in
the course of human events...')"><BR>
<INPUT TYPE="reset" onClick="setIt('Default Value')"><P>
<INPUT TYPE="button" VALUE="Watch It" onClick="watchIt(true)">
<INPUT TYPE="button" VALUE="Don't Watch It" onClick="watchIt(false)">
</FORM>
</BODY>
</HTML>
```

Better ways exist to intercept and preprocess user input, but the `watch()` function can be a helpful debugging tool when you want to monitor the hidden workings of scripts.

Using custom objects

There is no magic to knowing when to use a custom object instead of an array in your application. The more you work with and understand the way custom objects work, the more likely you will think about your data-carrying scripts in these terms, especially if an object can benefit from having one or more methods associated with it. This avenue is certainly not one for beginners, but I recommend that you give custom objects more than a casual perusal once you gain some JavaScripting experience.

JavaScript Components

The level 4 browsers from Netscape and Microsoft introduce another twist on the notion of custom objects: external components written in JavaScript. Each company is headed in a different (and mutually exclusive) direction on the authoring and deployment of these components. Netscape is following through on its commitment to the Java environment by fashioning these components after Java Beans, calling them *JavaScript Beans* (or JSBs for short). Microsoft has defined a new term for its components — *scriptlets* — and commonly calls these items *controls*, just as ActiveX components are controls.

What I find most interesting about both approaches is that their goals are identical. In each case, a component author defines in an external file an object that has properties, methods, and event handlers accessible by scripts in a main document. The differences between the two approaches center on how these objects are defined and how each browser handles the objects. For now, neither browser knows what to do with the other browser's JavaScript components.

JavaScript Beans

A JavaScript Bean (JSB) is a separate file (extension .jsb) that defines a new object for inclusion in an HTML document. No HTML content appears in the JSB file, but rather it has an extensive set of tags that let you define constructors, properties, methods, and event handlers for the object. Functions defined inside a JSB file may, however, use `document.write()` to generate content that is displayed in the main document courtesy of the JSB.

If you know Java, you can readily see the Java influence in Netscape's approach to script components. JSB files are treated as classes in packages. A package is nothing more than a directory structure for storing one or more files. The structure is relative to the main HTML document that loads the component. In the Visual JavaScript tool, for example, JavaScript Beans are automatically stored in a directory named peas, which is itself inside a directory named netscape. In Java, such a package would be described as

```
netscape.peas
```

But in JavaScript, which actually accesses these components, the periods are replaced with underscore characters. Therefore, to generate an instance of a JavaScript Bean in a document, you use the `new` keyword in the following manner:

```
var myJSB = new netscape_peas_BeanObject([params])
```

Parameters to a constructor must be passed in the form of a native JavaScript object. The object defines property names (as named index values to the object properties) and values, as in the following fragment:

```
var params = new Object()
params.name = "Fred"
params.color = "red"
params.birthYear = 1982
var oneMember = new netscape_peas_member(params)
```

Thereafter, you can refer to the object via the variable that holds a reference to its instance. Property and method references are in the same format you use for other objects:

```
JSBRef.propertyName | methodName()
```

If a JavaScript Bean file relies on an external .js or other kind of file, the JSB and supporting files must be encapsulated in a JAR archive file. The archive does not have to be signed (see Chapter 40), but you must use the JAR Packager tool (from Netscape) to join these files together as one entity. It turns out to be a convenience for server management, since there are fewer itsy-bitsy pieces to worry about.

For further details on creating JavaScript Bean components, download the Component Developer's Kit from `http://developer.netscape.com`. Most of Netscape's discussion about JSBs is in relation to the Visual JavaScript tool (see Chapter 46), but you can deploy JSBs without Visual JavaScript if you understand how they work inside your HTML documents (both server- and client-side).

Scriptlets

Technically speaking, scriptlets are not just about JavaScript. In Microsoft's development world, scripting languages are interchangeable. Therefore, you can create scriptlets with JavaScript or VBScript as you like.

A scriptlet definition lives in a file with a standard HTML file extension because this file can also contain HTML content to be displayed within a rectangular space in a main page (much like an ActiveX control does). Inside the scriptlet file are numerous script statements that define the object and its behavior. Many of these scripted items must follow a very specific naming convention and format that allows properties and methods to be exposed to a document that loads the scriptlet.

To load the object into the HTML of the current page requires the `<OBJECT>` tag with a few attributes unique to Internet Explorer (Version 4 or later). For example, the following code loads a calendar scriptlet into a 400 × 300 pixel rectangle:

```
<OBJECT ID="cal" TYPE="text/x-scriptlet" DATA="calendar.html"
HEIGHT="300" WIDTH="400">
```

With an object loaded in the document, you use normal object references to access its values. For example, when the calendar object has its `Value` property retrieved by script, it internally invokes one of its own methods to fetch that value. The document's script to retrieve that value from the calendar object would be

```
var calendarDate = document.cal.Value
```

Scriptlets can also be easily programmed to interact with events on the page. Without much difficulty, you can make the user interaction with the content of a

scriptlet and the main document elements completely seamless to the user. For more details on writing scriptlets, download the Component Development section of Microsoft's Internet Client SDK at `http://www.microsoft.com/msdn/sdk/inetsdk/asetup/`.

Deployment

Such divergent approaches to component objects leave developers who desire cross-browser solutions stuck in the middle. Unless you wish to implement the same solution two different ways to support a cross-browser audience, I suggest using scripting components only when the application will be deployed to a single-browser audience, such as on an intranet you control. It is unclear whether the nascent document object model standard will address this aspect of scripted objects. In my examinations and experiments with both component models so far, I don't see a clear advantage for either one over the other. Therefore, until an approach is supported as a standard (if it ever will be), you will have to pick sides in this battle of the browser wars.

✦ ✦ ✦