CHAPTER 35

Global Functions and Statements

✦ ✦ ✦ ✦

In This Chapter

Converting strings into object references

Converting strings to numbers for math

Adding comments to scripts

✦ ✦ ✦ ✦

In addition to all the objects and other language constructs described in the preceding chapters of this reference part of the book, several language items need to be treated on a global scale. These items apply to no particular objects (or any object) and can be used anywhere in a script. If you've read earlier chapters, you have been introduced to many of these functions and statements. This chapter is presented as a convenient place to highlight these all-important items that are otherwise easily forgotten. This chapter covers the following functions and statements:

Functions	*Statements*
`eval()`	`// and /*...*/` (comment)
`escape()`	`var`
`isNaN()`	
`Number()`	
`parseFloat()`	
`parseInt()`	
`String()`	
`unescape()`	
`unwatch()`	
`watch()`	

Very often, the discussions point to examples in earlier chapters that demonstrate the item in context.

Functions

Global functions are not tied to the document object model. Instead they typically let you convert data from one type to another type.

`eval(`*`string`*`)`

Returns: Object reference.

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

Expression evaluation, as you are probably well aware by now, is an important concept to grasp in scripting with JavaScript (and programming in general). An expression evaluates to some value. But occasionally you need to force an additional evaluation on an expression to receive the desired results. The `eval()` function acts on a string value to force an evaluation of that string expression.

Perhaps the most common application of the `eval()` function is to convert a string version of an object reference to a genuine object reference. For example, in an effort to create a Dynamic HTML script that accommodates the different ways that Microsoft and Netscape reference positionable objects (see Chapter 41), one technique is to assemble references out of the comparable pieces of references. In the following function, the name of a positionable object is passed as a parameter. The example assumes that global variable flags have been set elsewhere for `isNav4` and `isIE4`. The job of the function is to create a valid reference to the object depending on the browser being run by the user:

```
function getReference(objName) {
        if (navigator.appVersion.charAt(0) == "4") {
           if (navigator.appName == "Netscape") {
              var range = ""
              var styleObj = ""
           } else {
              var range = ".all"
              var styleObj = ".style"
           }
           var theObj = eval("document" + range + "." + objName +
styleObj)
           return theObj
        }
        return null
}
```

In the Netscape branch of the preceding example, the variables `range` and `styleObj` are assigned empty strings; for the Microsoft branch, each variable assumes the components that must be inserted into an object reference for the Microsoft syntax. If the components are concatenated without the `eval()` function, the result would simply be a concatenated string, which is not the same as the object reference. By forcing an additional evaluation with the `eval()` function, the script invokes JavaScript to see if one more level of evaluation is needed. If JavaScript finds that the evaluation of that string is a valid object reference, it returns the reference as the result; otherwise the function returns `undefined`.

Any JavaScript statement or expression stored as a string can be evaluated with the `eval()` function. This includes string equivalents of arithmetic expressions, object value assignments, and object method invocation. You may not always need the `eval()` function. For example, if your script loops through a series of objects whose names include serial numbers, you can use the object names as array indices, rather than using `eval()` to assemble the object references. The inefficient way to set the value of a series of fields named `data0`, `data1`, and so on, is as follows:

```
function fillFields() {
       var theObj
       for (var i = 0; i < 10; i++) {
          theObj = eval("document.forms[0].data" + i)
          theObj.value = i
       }
}
```

A more efficient way is to perform the concatenation within the index brackets for the object reference:

```
function fillFields() {
       for (var i = 0; i < 10; i++) {
          document.forms[0].elements["data" + i].value = i
       }
}
```

Note

A slight change to the `eval()` function occurred between Navigator 2 and 3. In Navigator 2, the function was considered a built-in function. But in Navigator 3 and onward, `eval()` was defined internally as a method of all objects. This does not change its basic functionality, but does provide known behavior for all types of objects, including those that would not normally use this method.

escape("string" [,1])
unescape("string")

Returns: String.

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

If you watch the content of the Location field in your browser, you may occasionally see URLs that include a lot of % symbols plus some numbers. The format you see is URL encoding that allows even multiple word strings and nonalphanumeric characters to be sent as one contiguous string of a very low common-denominator character set. This encoding turns a character such as a space into its hexadecimal equivalent value, preceded by a percent symbol. For example, the space character (ASCII value 32) is hexadecimal 20, so the encoded version of a space is %20.

All characters, including tabs and carriage returns, can be encoded in this way and sent as a simple string that can be decoded on the receiving end for reconstruction. This encoding can also be used to preprocess multiple lines of text that must be stored as a character string in databases. To convert a plain-language string to its encoded version, use the `escape()` method. This function returns a string consisting of the encoded version. For example

```
var theCode = escape("Hello there")
       // result: "Hello%20there"
```

Most, but not all, nonalphanumeric characters are converted to escaped versions with the `escape()` function. One exception is the plus sign, which URLs use to separate components of search strings. If you must encode the plus symbol, too, then add the optional second parameter to the function, and the plus symbol is converted to its hexadecimal equivalent, 2B:

```
var a = escape("Adding 2+2")
       // result: "Adding%202+2
var a = escape("Adding 2+2",1)
       // result: "Adding%202%2B2
```

To convert an escaped string back into plain language, use the `unescape()` function. This function returns a string, and converts all URL-encoded strings, including those encoded with the optional parameter.

isNaN(expression)

Returns: Boolean.

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

For those instances in which a calculation relies on data coming from a text field or other string-oriented source, you frequently need to be sure that the value is a number. If the value is not a number, the calculation may result in a script error.

Use the `isNaN()` function to test whether a value is a number prior to passing the value onto the operation. The most common usage of this function is to test the result of a `parseInt()` or `parseFloat()` function. If the strings submitted for conversion to those functions cannot be converted to a number, the resulting value is `NaN` (a special symbol indicating "not a number"). The `isNaN()` function returns true if the value is not a number.

A convenient way to use this function is to intercept improper data before it can do damage, as follows:

```
function calc(form) {
       var inputValue = parseInt(form.entry.value)
       if (isNaN(inputValue)) {
          alert("You must enter a number to continue.")
       } else {
            statements for calculation
        }
}
```

Probably the biggest mistake scripters make with this function is failing to observe the case of all the letters in the function name. The trailing uppercase "N" is easy to miss.

Note

The `isNaN()` function works in Navigator 2 only on UNIX platforms. It is available on all platforms in Navigator 3 onward and on Internet Explorer 3 and onward.

Number("string")
parseFloat("string")
parseInt("string" [,radix])

Returns: Number.

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

All three of these functions convert string values into a numeric value. The `parseInt()` and `parseFloat()` functions are compatible across all versions of all browsers; the `Number()` function is new with Navigator 4.

Use the `Number()` function when your script is not concerned with the precision of the value and prefers to let the source string govern whether the returned value is a floating-point number or an integer. The function takes a single parameter, a string to convert to a number value.

The `parseFloat()` functions also let the string source value determine whether the returned value is a floating-point number or integer. If the source string includes any non-zero value to the right of the decimal, the result is a floating-point number. But if the string value were, say, `"3.00"`, the returned value would be an integer value.

An extra, optional parameter for `parseInt()` lets you define the number base to be used in the conversion. If you don't specify a radix parameter, JavaScript tries to look out for you, but in so doing may cause some difficulty for you. The primary problem comes when the string parameter for `parseInt()` starts with a zero, which a text box entry or database field might do. In JavaScript, numbers starting with zero are treated as octal (base 8) numbers. Therefore, `parseInt("010")` yields the decimal value 8.

When you apply the `parseInt()` function, you should always specify the radix of 10 if you are working in base 10 numbers. You can, however, specify any radix value from 2 through 36. For example, to convert a binary number string to its decimal equivalent, you would assign a radix of two, as follows:

```
var n = parseInt("011",2)
        // result: 3
```

Similarly, you can convert a hexadecimal string to its decimal equivalent by specifying a radix of 16:

```
var n = parseInt("4F",16)
        // result: 79
```

String(objectOrValue) toString([radix])

Returns: String.

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

Every JavaScript language object and document object has a `toString()` method associated with it. The method is designed to render the contents of the object in as meaningful a way as possible. Table 35-1 shows the result of applying the `toString()` method on each of the convertible core language object types.

Table 35-1
Object toString() Results

Object Type	*Result*
String	The same string
Number	String equivalent (but numeric literals cannot be converted)
Boolean	“true” or “false”
Array	Comma-delimited list of array contents (with no spaces after commas)
Function	Decompiled string version of the function definition

Many document objects can also be converted to a string. For example, a location object returns its URL. But when an object has nothing suitable to return for its content as a string, it usually returns a string in the following format:

```
[object objectType]
```

The `toString()` function is available on all versions of all browsers. New in Navigator 4 is the `String()` function, which operates like `toString()`. However, a convenient improvement to `toString()` for Navigator 3 is the optional radix parameter. By setting this parameter between 2 and 16, you can convert numbers to string equivalents in different number bases. Listing 35-1 calculates and draws a conversion table for decimal, hexadecimal, and binary numbers between 0 and

20. In this case, the values being converted to strings are the index counter of the for loop.

Listing 35-1: **Using toString() with Radix Values**

```
<HTML>
<HEAD>
<TITLE>Number Conversion Table</TITLE>
</HEAD>
<BODY>
<B>Using toString() to convert to other number bases:</B>
<HR>
<TABLE BORDER=1>
<TR>
<TH>Decimal</TH><TH>Hexadecimal</TH><TH>Binary</TH></TR>
<SCRIPT LANGUAGE="JavaScript">
var content = ""
for (var i = 0; i <= 20; i++) {
       content += "<TR>"
       content += "<TD>" + i.toString(10) + "</TD>"
       content += "<TD>" + i.toString(16) + "</TD>"
       content += "<TD>" + i.toString(2) + "</TD></TR>"
}
document.write(content)
</SCRIPT>
</TABLE>
</BODY>
</HTML>
```

User-defined objects do not have a `toString()` method assigned to them, but you can create your own. For example, if you wanted to make your custom object's `toString()` method behave like an array's method, then you define the action of the method and assign that function to a property of the object, as shown in Listing 35-2.

Listing 35-2: **Creating a Custom toString() Function**

```
<HTML>
<HEAD>
<TITLE>Custom toString()</TITLE>
<SCRIPT LANGUAGE="JavaScript1.2">
function customToString() {
       var dataArray = new Array()
       var count = 0
       for (var i in this) {
          dataArray[count++] = this[i]
          if (count > 2) {
             break
          }
```

(continued)

Listing 35-2 *(continued)*

```
        }
        return dataArray.join(",")
}
var book = {title:"The Aeneid", author:"Virgil", pageCount:543}
book.toString = customToString
</SCRIPT>
</HEAD>
<BODY>
<B>A user-defined toString() result:</B>
<HR>
<SCRIPT LANGUAGE="JavaScript">
document.write(book.toString())
</SCRIPT>
</BODY>
</HTML>
```

When you run Listing 35-2, you can see how the custom object's `toString()` handler extracts the values of all elements of the object except for the last one, which is the function handler reference. You can customize how the data should be labeled and/or formatted.

unwatch(property)
watch(property, handler)

Returns: Nothing.

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

To supply the right kind of information to its own external debugger, JavaScript in Navigator 4 implements two new global functions that belong to every object, including user-defined objects. The `watch()` function keeps an eye on a desired object and property. If that property is set by assignment, the function invokes another user-defined function that receives information about the property name, its old value, and its new value. The `unwatch()` function turns off the watch functionality for a particular property. See Listing 34-8 near the end of Chapter 34 for an example of how to use these functions that can be assigned to any object.

Statements

The final two statements covered here are for setting off comments and defining variables — statements used a lot in JavaScript.

//

/*...*/

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

Comments are statements that are ignored by the JavaScript interpreter (or server-side compiler) but allow authors to leave notes about how things work in their scripts. While lavish comments are useful to authors during a script's creation and maintenance, the full content of a client-side comment is downloaded with the document. Every byte of nonoperational content of the page takes a bit more time to download. Still, I recommend lots of comments, particularly as you create a script.

JavaScript offers two styles of comments. One style consists of two forward slashes (no spaces between them). JavaScript ignores any characters to the right of those slashes, even if they appear in the middle of a line. You can stack as many lines of these single-line comments as is necessary to convey your thoughts. My style typically places a space between the second slash and the beginning of my comment. The following are examples of valid one-line comment formats:

```
// this is a comment line usually about what's to come
var a = "Fred"  // a comment about this line
// You may want to capitalize the first word of a comment
// sentence if it runs across multiple lines.
//
// And you can leave a completely blank line, like the one above.
```

For longer comments, it is usually more convenient to enclose the section in the other style of comment. This one opens with a forward slash and asterisk (/*) and ends with a start and forward slash (*/). All statements in between — including multiple lines — are ignored by JavaScript. If you want to briefly comment out a large segment of your script, it is easiest to bracket the segment with these comment symbols. To make these comment blocks easier to find, I generally place these symbols on their own lines, as follows:

```
/*
some
  commented-out
    statements
*/
```

If you are developing rather complex documents, you might find using comments a convenient way to help you organize segments of your scripts and make each segment easier to find. For example, you could define a comment block above each function and describe what the function is about:

```
/*-----------------------------------------------
   calculate()
   Performs a mortgage calculation based on
   parameters blah, blah, blah.  Called by blah
   blah blah.
-----------------------------------------------*/
function calculate(form) {
       statements
}
```

var

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

Before using any variable, you should declare it (and optionally initialize it with a value) via the `var` statement. If you omit the `var` keyword, the variable is automatically assigned as a global variable within the current document. To keep a variable local to a function, you must declare or initialize the variable with the `var` keyword inside the function's braces.

If you assign no value to a variable, it evaluates to null. Because a JavaScript variable is not limited to one variable type during its lifetime, you don't need to initialize a variable to an empty string or zero unless that initial value will help your scripting. For example, if you initialize a variable as an empty string, you can then use the add-by-value operator (+=) to append string values to that variable in a future statement in the document.

To save statement lines, you can declare and/or initialize multiple variables with a single var statement. Each varName=value pair must be separated by a comma, as in

```
var name, age, height  // declare as null
var color="green", temperature=85.6 // initialize
```

Variable names (also known as *identifiers*) must be one contiguous string of characters, but the first character must be a letter. Many punctuation symbols are also banned, but the underscore character is valid and is often used to separate multiple words in a long variable name. All variable names (like most identifiers in JavaScript) are case-sensitive, so your naming of a particular variable must be identical throughout the variable's scope.

✥ ✥ ✥