

# JavaScript Operators

JavaScript is rich in *operators*: words and symbols in expressions that perform operations on one or two values to arrive at another value. Any value on which an operator performs some action is called an *operand*. An expression may contain one operand and one operator (called a unary operator) or two operands separated by one operator (called a binary operator). Many of the same symbols are used in a variety of operators. The combination and order of those symbols are what distinguish their powers.

## Operator Categories

To help you grasp the range of JavaScript operators, I've grouped them into five categories. I have assigned a wholly untraditional name to the second group — but a name that I believe better identifies its purpose in the language. Table 32-1 shows the operator types.

Table 32-1  
JavaScript Operator Categories

Type	What It Does
Comparison	Compares the values of two operands, deriving a result of either true or false (used extensively in condition statements for <code>if...else</code> and for loop constructions)
Connubial	Joins together two operands to produce a single value that is a result of an arithmetical or other operation on the two
Assignment	Stuffs the value of the expression of the right-hand operand into a variable name on the left-hand side, sometimes with minor modification, as determined by the operator symbol
Boolean	Performs Boolean arithmetic on one or two Boolean operands
Bitwise	Performs arithmetic or column-shifting actions on the binary (base-2) representations of two operands

# 32

CHAPTER



### In This Chapter

Understanding operator categories

Role of operators in script statements



Any expression that contains an operator evaluates to a value of some kind. Sometimes the operator changes the value of one of the operands; other times the result is a new value. Even this simple expression

```
5 + 5
```

shows two integer operands joined by the addition operator. This expression evaluates to 10. The operator is what provides the instruction for JavaScript to follow in its never-ending drive to evaluate every expression in a script.

Doing an equality comparison on two operands that, on the surface, look very different is not at all uncommon. JavaScript doesn't care what the operands look like — only how they evaluate. Two very dissimilar-looking values can, in fact, be identical when they are evaluated. Thus, an expression that compares the equality of two values such as

```
fred == 25
```

does, in fact, evaluate to true if the variable `fred` has the number 25 stored in it from an earlier statement.

## Comparison Operators

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

Any time you compare two values in JavaScript, the result is a Boolean true or false value. You have a wide selection of comparison operators to choose from, depending on the kind of test you want to apply to the two operands. Table 32-2 lists all six comparison operators.

Table 32-2 JavaScript Comparison Operators			
<i>Syntax</i>	<i>Name</i>	<i>Operand Types</i>	<i>Results</i>
<code>==</code>	Equals	All	Boolean
<code>!=</code>	Does not equal	All	Boolean
<code>&gt;</code>	Is greater than	All	Boolean
<code>&gt;=</code>	Is greater than or equal to	All	Boolean
<code>&lt;</code>	Is less than	All	Boolean
<code>&lt;=</code>	Is less than or equal to	All	Boolean

For numeric values, the results are the same as those you'd expect from your high school algebra class. Some examples follow, including some that may not be obvious:

```

10 == 10    // true
10 == 10.0  // true
9 != 10     // true
9 > 10      // false
9.99 <= 9.98 // false

```

Strings can also be compared on all of these levels:

```

"Fred" == "Fred" // true
"Fred" == "fred" // false
"Fred" > "fred"  // false
"Fran" < "Fred" // true

```

To calculate string comparisons, JavaScript converts each character of a string to its ASCII value. Each letter, beginning with the first of the left-hand operator, is compared to the corresponding letter in the right-hand operator. With ASCII values for uppercase letters being less than their lowercase counterparts, an uppercase letter evaluates to being less than its lowercase equivalent. JavaScript takes case-sensitivity very seriously.

Values for comparison can also come from object properties or values passed to functions from event handlers or other functions. A common string comparison used in data-entry validation is the one that sees if the string has anything in it:

```
form.entry.value != "" // true if something is in the field
```

## Equality of Disparate Data Types

For all versions of JavaScript before 1.2, when your script tries to compare string values consisting of numerals and real numbers (for example, "123" == 123 or "123" != 123), JavaScript anticipates that you want to compare apples to apples. Internally it does some data type conversion that does not affect the data type of the original values (for example, if the values are in variables). But the entire situation is more complex, because other data types, such as objects, need to be dealt with. Therefore, prior to JavaScript 1.2, the rules of comparison are as shown in Table 32-3.

Table 32-3  
Equality Comparisons for JavaScript 1.0 and 1.1

<i>Operand A</i>	<i>Operand B</i>	<i>Internal Comparison Treatment</i>
Object reference	Object reference	Compare object reference evaluations
Any data type	Null	Convert nonnull to its object type and compare against null
Object reference	String	Convert object to string and compare strings
String	Number	Convert string to number and compare numbers

The logic to what goes on in equality comparisons from Table 32-3 requires a lot of forethought on the scripter's part, because you have to be very conscious of the particular way data types may or may not be converted for equality evaluation (even though the values themselves are not converted). In this situation, it is best to supply the proper conversion where necessary in the comparison statement. This ensures that what you want to compare — say, the string versions of two values or the number versions of two values — is compared, rather than leaving the conversion up to JavaScript.

Backward compatible conversion from a number to string entails concatenating an empty string to a number:

```
var a = "09"
var b = 9
a == "" + b // result: false, because "09" does not equal "9"
```

For converting strings to numbers, you have numerous possibilities. The simplest is subtracting zero from a numeric string:

```
var a = "09"
var b = 9
a-0 == b // result: true because number 9 equals number 9
```

You can also use the `parseInt()` and `parseFloat()` functions to convert strings to numbers:

```
var a = "09"
var b = 9
parseInt(a, 10) == b // result: true because number 9 equals number 9
```

To clear up the ambiguity of JavaScript's equality internal conversions, JavaScript 1.2 in Navigator 4 introduces a different way of evaluating equality. For all scripts encapsulated inside a `<SCRIPT LANGUAGE="JavaScript1.2"></SCRIPT>` tag pair, equality operators do *not* perform any automatic type conversion. Therefore no number will ever be automatically equal to a string version of that same number. Data and object types must match before their values are compared.

JavaScript 1.2 provides some convenient global functions for converting strings to numbers and vice versa: `String()` and `Number()`. To demonstrate these methods, the following examples use the `typeof` operator to show the data type of expressions using these functions:

```
typeof 9 // result: number
type of String(9) // result: string
type of "9" // result: string
type of Number("9") // result: number
```

Neither of these functions alters the data type of the value being converted. But the value of the function is what gets compared in an equality comparison:

```
var a = "09"
var b = 9
a == String(b) // result: false, because "09" does not equal "9"
typeof b // result: still a number
Number(a) == b // result: true, because 9 equals 9
typeof a // result: still a string
```

For forward and backward compatibility, you should always make your equality comparisons compare identical data types.

## Connubial Operators

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

Connubial operators is my terminology for those operators that join two operands to yield a value related to the operands. Table 32-4 lists the connubial operators in JavaScript.

Table 32-4  
JavaScript Connubial Operators

<i>Syntax</i>	<i>Name</i>	<i>Operand Types</i>	<i>Results</i>
+	Plus	Integer, float, string	Integer, float, string
-	Minus	Integer, float	Integer, float
*	Multiply	Integer, float	Integer, float
/	Divide	Integer, float	Integer, float
%	Modulo	Integer, float	Integer, float
++	Increment	Integer, float	Integer, float
--	Decrement	Integer, float	Integer, float
-val	Negation	Integer, float	Integer, float

The four basic arithmetic operators for numbers should be straightforward. The plus operator also works on strings to join them together, as in

```
"Howdy " + "Doody" // result = "Howdy Doody"
```

In object-oriented programming terminology, the plus sign is said to be *overloaded*, meaning that it performs a different action depending on its context. Remember, too, that string concatenation does not do anything on its own to monitor or insert spaces between words. In the preceding example, the space between the names is part of the first string.

Modulo arithmetic is helpful for those times when you want to know if one number divides evenly into another. You used it in an example in the last chapter to figure out if a particular year was a leap year. Although some other leap year considerations exist for the turn of each century, the math in the example simply checked whether the year was evenly divisible by four. The result of the modulo math is the remainder

of division of the two values: When the remainder is 0, one divides evenly into the other. Here are some samples of years evenly divisible by four:

```
1994 % 4    // result = 2
1995 % 4    // result = 3
1996 % 4    // result = 0 --Bingo! Leap and election year!
```

Thus, I used this operator in a condition statement of an `if . . . else` structure:

```
var howMany = 0
today = new Date()
var theYear = today.getFullYear()
if (theYear % 4 == 0) {
    howMany = 29
} else {
    howMany = 28
}
```

Some other languages offer an operator that results in the integer part of a division problem solution: integral division, or `div`. Although JavaScript does not have an explicit operator for this behavior, you can re-create it reliably if you know that your operands are always positive numbers. Use the `Math.floor()` or `Math.ceil()` methods with the division operator, as in

```
Math.floor(4/3) // result = 1
```

In this example, `Math.floor()` works only with values greater than or equal to 0; `Math.ceil()` works for values less than 0.

The increment operator (`++`) is a unary operator (only one operand) and displays two different behaviors, depending on the side of the operand on which the symbols lie. Both the increment and decrement (`--`) operators can be used in conjunction with assignment operators, which I cover next.

As its name implies, the increment operator increases the value of its operand by one. But in an assignment statement, you have to pay close attention to precisely when that increase takes place. An assignment statement stuffs the value of the right operand into a variable on the left. If the `++` operator is located in front of the right operand (prefix), the right operand is incremented before the value is assigned to the variable; if the `++` operator is located after the right operand (postfix), the previous value of the operand is sent to the variable before the value is incremented. Follow this sequence to get a feel for these two behaviors:

```
var a = 10 // initialize a to 10
var z = 0  // initialize z to zero
z = a      // a = 10, so z = 10
z = ++a    // a becomes 11 before assignment, so a = 11 and z becomes 11
z = a++    // a is still 11 before assignment, so z = 11; then a becomes 12
z = a++    // a is still 12 before assignment, so z = 12; then a becomes 13
```

The decrement operator behaves the same way, except that the value of the operand decreases by one. Increment and decrement operators are used most often with loop counters in `for` and `while` loops. The simpler `++` or `--` symbology

is more compact than reassigning a value by adding 1 to it (such as, `z = z + 1` or `z += 1`). Because these are unary operators, you can use the increment and decrement operators without an assignment statement to adjust the value of a counting variable within a loop:

```
function doNothing() {
    var i = 1
    while (i < 20) {
        ++i
    }
    alert(i) // breaks out at i = 20
}
```

The last connubial operator is the negation operator (`-val`). By placing a minus sign in front of any numeric value (no space between the symbol and the value), you instruct JavaScript to evaluate a positive value as its corresponding negative value, and vice versa. The operator does not change the actual value. The following example provides a sequence of statements to demonstrate:

```
x = 2
y = 8
-x // expression evaluates to -2, but x still equals 2
-(x + y // doesn't change variable values; evaluates to -10
-x + y // evaluates to 6, but x still equals 2
```

To negate a Boolean value, see the Not (!) operator in the discussion of Boolean operators.

## Assignment Operators

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

Assignment statements are among the most common statements you write in your JavaScript scripts. These statements are where you copy a value or the results of an expression into a variable for further manipulation of that value.

You assign values to variables for many reasons, even though you could probably use the original values or expressions several times throughout a script. Here is a sampling of reasons why you should assign values to variables:

- ♦ Variable names are usually shorter
- ♦ Variable names can be more descriptive
- ♦ You may need to preserve the original value for later in the script
- ♦ The original value is a property that cannot be changed
- ♦ Invoking the same method several times in a script is not efficient

Newcomers to scripting often overlook the last reason. For instance, if a script is writing HTML to a new document, it's more efficient to assemble the string of large chunks of the page into one variable before invoking the `document.writeln()` method to send that text to the document. This method is more efficient than literally sending out one line of HTML at a time with multiple `document.writeln()` method statements. Table 32-5 shows the range of assignment operators in JavaScript.

Table 32-5  
JavaScript Assignment Operators

<i>Syntax</i>	<i>Name</i>	<i>Example</i>	<i>Means</i>
<code>=</code>	Equals	<code>x = y</code>	<code>x = y</code>
<code>+=</code>	Add by value	<code>x += y</code>	<code>x = x + y</code>
<code>-=</code>	Subtract by value	<code>x -= y</code>	<code>x = x - y</code>
<code>*=</code>	Multiply by value	<code>x *= y</code>	<code>x = x * y</code>
<code>/=</code>	Divide by value	<code>x /= y</code>	<code>x = x / y</code>
<code>%=</code>	Modulo by value	<code>x %= y</code>	<code>x = x % y</code>
<code>&lt;&lt;=</code>	Left shift by value	<code>x &lt;&lt;= y</code>	<code>x = x &lt;&lt; y</code>
<code>&gt;=</code>	Right shift by value	<code>x &gt;= y</code>	<code>x = x &gt; y</code>
<code>&gt;&gt;=</code>	Zero fill by value	<code>x &gt;&gt;= y</code>	<code>x = x &gt;&gt; y</code>
<code>&amp;=</code>	Bitwise AND by value	<code>x &amp;= y</code>	<code>x = x &amp; y</code>
<code> =</code>	Bitwise OR by value	<code>x  = y</code>	<code>x = x   y</code>
<code>^=</code>	Bitwise XOR by value	<code>x ^= y</code>	<code>x = x ^ y</code>

As clearly demonstrated in the top group (see “Bitwise Operators” later in the chapter for information on the bottom group), assignment operators beyond the simple equal sign can save some characters in your typing, especially when you have a series of values that you’re trying to bring together in subsequent statements. You’ve seen plenty of examples in previous chapters, where you’ve used the add-by-value operator (`+=`) to work wonders with strings as you assemble a long string variable that you eventually send to a `document.write()` method. Look at this excerpt from Listing 29-4, where you use JavaScript to create the content of an HTML page on the fly:

```
var page = "" // start assembling next part of page and form
page += "Select a planet to view its planetary data: "
page += "<SELECT NAME='planets'> "
// build popup list from array planet names
for (var i = 0; i < solarSys.length; i++) {
    page += "<OPTION" // OPTION tags
    if (i == 1) { // pre-select first item in list
        page += " SELECTED"
```

```

    }
    page += ">" + solarSys[i].name
}
page += "</SELECT><P>" // close selection item tag
document.write(page)    // lay out this part of the page

```

The script segment starts with a plain equals assignment operator to initialize the `page` variable as an empty string. In many of the succeeding lines, you use the add-by-value operator to tack additional string values onto whatever is in the `page` variable at the time. Without the add-by-value operator, you'd be forced to use the plain equals assignment operator for each line of code to concatenate new string data to the existing string data. In that case, the first few lines of code would look like this:

```

var page = "" // start assembling next part of page and form
page = page + "Select a planet to view its planetary data: "
page = page + "<SELECT NAME='planets'> "

```

Within the `for` loop, the repetition of `page +` makes the code very difficult to read, trace, and maintain. These enhanced assignment operators are excellent shortcuts that you should use at every turn.

## Boolean Operators

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

Because a great deal of programming involves logic, it is no accident that the arithmetic of the logic world plays an important role. You've already seen dozens of instances where programs make all kinds of decisions based on whether a statement or expression is the Boolean value of true or false. What you haven't seen much of yet is how to combine multiple Boolean values and expressions — something that scripts with slightly above average complexity may need to have in them.

In the various condition expressions required throughout JavaScript (such as in an `if` construction), the condition that the program must test for may be more complicated than, say, whether a variable value is greater than a certain fixed value or whether a field is not empty. Look at the case of validating a text field entry for whether the entry contains all the numbers that your script may want. Without some magical JavaScript function to tell you whether or not a string consists of all numbers, you have to break apart the entry character by character and examine whether each character falls within the range of 0 through 9. But that examination actually comprises two tests: You can test for any character whose ASCII value is less than 0 or greater than 9. Alternatively, you can test whether the character is greater than or equal to 0 and is less than or equal to 9. What you need is the bottom-line evaluation of both tests.

## Boolean math

That's where the wonder of Boolean math comes into play. With just two values — true and false — you can assemble a string of expressions that yield Boolean results and then let Boolean arithmetic figure out whether the bottom line is true or false.

But you don't add or subtract Boolean values like numbers. Instead, in JavaScript, you use one of three Boolean operators at your disposal. Table 32-6 shows the three operator symbols. In case you're unfamiliar with the characters in the table, the symbols for the Or operator are created by typing Shift-backslash.

Table 32-6  
JavaScript Boolean Operators

<i>Syntax</i>	<i>Name</i>	<i>Operands</i>	<i>Results</i>
&&	And	Boolean	Boolean
	Or	Boolean	Boolean
!	Not	One Boolean	Boolean

Using Boolean operators with Boolean operands gets tricky if you're not used to it, so I have you start with the simplest Boolean operator: Not. This operator requires only one operand. The Not operator precedes any Boolean value to switch it back to the opposite value (from true to false, or from false to true). For instance

```
!true // result = false
!(10 > 5) // result = false
!(10 < 5) // result = true
!(document.title == "Flintstones") // result = true
```

As shown here, enclosing the operand of a Not expression inside parentheses is always a good idea. This forces JavaScript to evaluate the expression inside the parentheses before flipping it around with the Not operator.

The And (&&) operator joins two Boolean values to reach a true or false value based on the results of both values. This brings up something called a *truth table*, which helps you visualize all the possible outcomes for each value of an operand. Table 32-7 is a truth table for the And operator.

Table 32-7  
Truth Table for the And Operator

<i>Left Operand</i>	<i>And Operator</i>	<i>Right Operand</i>	<i>Result</i>
True	&&	True	True
True	&&	False	False
False	&&	True	False
False	&&	False	False

Only one condition yields a true result: Both operands must evaluate to true. It doesn't matter on which side of the operator a true or false value lives. Here are examples of each possibility:

```
5 > 1 && 50 > 10 // result = true
5 > 1 && 50 < 10 // result = false
5 < 1 && 50 > 10 // result = false
5 < 1 && 50 < 10 // result = false
```

In contrast, the Or (`||`) operator is more lenient about what it evaluates to true. The reason is that if one or the other (or both) operands is true, the operation returns true. The Or operator's truth table is shown in Table 32-8.

Table 32-8  
Truth Table for the Or Operator

<i>Left Operand</i>	<i>Or Operator</i>	<i>Right Operand</i>	<i>Result</i>
True		True	True
True		False	True
False		True	True
False		False	False

Therefore, if a true value exists on either side of the operator, a true value is the result. Let's take the previous examples and swap the And operators with Or operators so you can see the Or operator's impact on the results:

```
5 > 1 || 50 > 10 // result = true
5 > 1 || 50 < 10 // result = true
5 < 1 || 50 > 10 // result = true
5 < 1 || 50 < 10 // result = false
```

Only when both operands are false does the Or operator return false.

## Boolean operators at work

Applying Boolean operators to JavaScript the first time just takes a little time and some sketches on a pad of paper to help you figure out the logic of the expressions. Earlier I talked about using a Boolean operator to see whether a character fell within a range of ASCII values for data-entry validation. Listing 32-1 (not on the CD-ROM) is a function discussed in more depth in Chapter 37. This function accepts any string and sees whether each character of the string has an ASCII value less than 0 or greater than 9 — meaning that the input string is not a number.

**Listing 32-1: Is the Input String a Number?**

```
function isNumber(inputStr) {  
    for (var i = 0; i < inputStr.length; i++) {  
        var oneChar = inputStr.substring(i, i + 1)  
        if (oneChar < "0" || oneChar > "9") {  
            alert("Please make sure entries are numbers only.")  
            return false  
        }  
    }  
    return true  
}
```

Combining a number of JavaScript powers to extract individual characters (substrings) from a string object within a `for` loop, the statement you're interested in is the condition of the `if` construction:

```
(oneChar < "0" || oneChar > "9")
```

In one condition statement, you use the Or operator to test for both possibilities. If you check the Or truth table (Table 32-8), you see that this expression returns true if either one or both tests returns true. If that happens, the rest of the function alerts the user about the problem and returns a false value to the calling statement. Only if both tests within this condition evaluate to false for all characters of the string does the function return a true value.

From the simple Or operator, I go to the extreme, where the function checks — in one condition statement — whether a number falls within several numeric ranges. The script in Listing 32-2 comes from one of the bonus applications on the CD-ROM (Chapter 49), in which a user enters the first three digits of a U.S. Social Security number.

**Listing 32-2: Is a Number within Discontiguous Ranges?**

```
// function to determine if value is in acceptable range for this  
application  
function inRange(inputStr) {  
    num = parseInt(inputStr)  
    if (num < 1 || (num > 586 && num < 596) || (num > 599 && num <  
700) || num > 728) {  
        alert("Sorry, the number you entered is not part of our  
database. Try another three-digit number.")  
        return false  
    }  
    return true  
}
```

By the time this function is called, the user's data entry has been validated enough for JavaScript to know that the entry is a number. Now the function must

check whether the number falls outside of the various ranges for which the application contains matching data. The conditions that the function tests here are whether the number is

- ♦ Less than 1
- ♦ Greater than 586 and less than 596 (using the And operator)
- ♦ Greater than 599 and less than 700 (using the And operator)
- ♦ Greater than 728

Each of these tests is joined by an Or operator. Therefore, if any one of these conditions proves false, the whole `if` condition is false, and the user is alerted accordingly.

The alternative to combining so many Boolean expressions in one condition statement would be to nest a series of `if` constructions. But such a construction requires not only a great deal more code, but much repetition of the alert message for each condition that could possibly fail. The combined Boolean condition was, by far, the best way to go.

## Bitwise Operators

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

For scripters, bitwise operations are an advanced subject. Unless you're dealing with external processes on CGI's or the connection to Java applets, it's unlikely that you will use bitwise operators. Experienced programmers who concern themselves with more specific data types (such as long integers) are quite comfortable in this arena, so I simply provide an explanation of JavaScript capabilities. Table 32-9 lists JavaScript bitwise operators.

Table 32-9  
JavaScript's Bitwise Operators

<i>Operator</i>	<i>Name</i>	<i>Left Operand</i>	<i>Right Operand</i>
&	Bitwise And	Integer value	Integer value
	Bitwise Or	Integer value	Integer value
^	Bitwise XOR	Integer value	Integer value
~	Bitwise Not	(None)	Integer value
<<	Left shift	Integer value	Shift amount
>	Right shift	Integer value	Shift amount
>>	Zero fill right shift	Integer value	Shift amount

The numeric value operands can appear in any of the JavaScript language's three numeric literal bases (decimal, octal, or hexadecimal). Once the operator has an operand, the value is converted to binary representation (32 bits long). For the first three bitwise operations, the individual bits of one operand are compared with their counterparts in the other operand. The resulting value for each bit depends on the operator:

- ♦ **Bitwise And:** 1 if both digits are 1
- ♦ **Bitwise Or:** 1 if either digit is 1
- ♦ **Bitwise Exclusive Or:** 1 if only one digit is a 1

Bitwise Not, a unary operator, inverts the value of every bit in the single operand. The bitwise shift operators operate on a single operand. The second operand specifies the number of positions to shift the value's binary digits in the direction of the arrows of the operator symbols. For example, the left shift (<<) operator has the following effect:

```
4 << 2 // result = 16
```

The reason for this is that the binary representation for decimal 4 is 00000100 (to eight digits, anyway). The left shift operator instructs JavaScript to shift all digits two places to the left, giving the binary result 00010000, which converts to 16 in decimal format. If you're interested in experimenting with these operators, use the `javascript:` URL to enable JavaScript to evaluate expressions for you. More advanced books on C and C++ programming are also of help.

## The typeof Operator

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

A special unary operator doesn't fall into any of the categories set out at the beginning of this chapter. Unlike all the other operators, which are predominantly concerned with arithmetic and logic, the `typeof` operator defines the kind of value and expression to which a variable evaluates. Typically, this operator is used to identify whether a variable value is one of the following types: number, string, boolean, object, or undefined.

Having this investigative capability in JavaScript is helpful because variables cannot only contain any one of those data types but can change their data type on the fly. Your scripts may need to handle a value differently based on the value's type. The most common use of the `typeof` property is as part of a condition. For example

```
if (typeof myVal == "number") {
    myVal = parseInt(myVal)
}
```

The evaluated value of the `typeof` operation is, itself, a string.

## The void Operator

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

In all scriptable browsers you can use the `javascript:` pseudo-protocol to supply the parameter for `HREF` and `SRC` attributes in HTML tags, such as links. In the process you have to be careful that the function or statement being invoked by the URL does not return or evaluate to any values. If a value comes back from such an expression, then the page content is often replaced by that value or sometimes the directory of the client's hard disk. To avoid this possibility use the `void` operator in front of the function or expression being invoked by the `javascript:` URL.

Different versions of Navigator accept a couple different ways of using this operator, but the one that works best is just placing the operator before the expression or function and separating them by a space, as in

```
javascript: void doSomething()
```

On occasion, you may have to wrap the expression inside parentheses after the `void` operator. This is necessary only when the expression contains operators of a lower precedence than the `void` operator (see “Operator Precedence” later in the chapter). But don't automatically wrap all expressions in parentheses, because Navigator 4 can experience problems with these.

The `void` operator makes sure the function or expression returns no value that the HTML attribute can use. If your audience consists solely of browsers aware of this operator, you can use it in lieu of the link object's `onClick=` event handler, which returns false to inhibit the link action.

## The new Operator

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

Most JavaScript core objects have constructor functions built into the language. To access those functions, you use the `new` operator along with the name of the constructor. The function returns a reference to the object, which your scripts can then use to get and set properties or invoke object methods. For example, creating a new date object requires invoking the date object's constructor, as follows:

```
var today = new Date()
```

Some object constructor functions require parameters to help define the object. Others, as in the case of the date object, can accept a number of different

parameter formats, depending on the format of date information you have to set the initial object. The `new` operator can be used with the following core language objects:

<i>JavaScript 1.0</i>	<i>JavaScript 1.1</i>	<i>JavaScript 1.2</i>
Date	Array	RegExp
Object	Boolean	
(Custom object)	Function	
	Image	
	Number	
	String	

## The delete Operator

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

Array objects do not contain a method to remove an element from the collection. You can always empty the data in an element by setting that element to an empty string or null, but the element remains in the collection. With the `delete` operator (new in Navigator 4 and Internet Explorer 4), you can completely remove the element (or the entire array, for that matter). In fact this works to such an extent that if your array uses numeric indices, a deletion of a given index removes that index value from the total array but without collapsing the array (which would alter index values of items higher than the deleted item). For example, consider the following simple dense array:

```
var oceans = new Array("Atlantic", "Pacific", "Indian","Arctic")
```

This kind of array automatically assigns numeric indices to its entries for addressing later in constructions such as `for` loops:

```
for (var i = 0; i < oceans.length; i++) {
    if (oceans[i] == form.destination.value) {
        statements
    }
}
```

If you then issue the statement

```
delete oceans[2]
```

the array undergoes significant changes. First, the third element is removed from the array. Importantly, the length of the array does not change. Even so, the

index value (2) is removed from the array, such that schematically the array looks like the following:

```
oceans[0] = "Atlantic"
oceans[1] = "Pacific"
oceans[3] = "Arctic"
```

If you try to reference `oceans[2]` in this collection, the result is `undefined`.

The `delete` operator works best on arrays that have named indices. Your scripts will have more control over the remaining entries and their values, because they don't rely on what could be a missing entry of a numeric index sequence.

Also feel free to use the `delete` operator to eliminate any arrays or objects created by your scripts. JavaScript takes care of this anyway when the page unloads, but your scripts may want to delete such objects to reduce some internal ambiguity with your scripted objects.

## The this Operator

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

JavaScript includes an operator that allows script statements to refer to the very object in which they are located. The self-referential operator is `this`.

The most common application of the `this` operator is in event handlers that pass references of themselves to functions for further processing, as in

```
<INPUT TYPE="text" NAME="entry" onChange="process(this)">
```

A function receiving the value assigns it to a variable that can be used to reference the sender, its properties, and its methods.

Because the `this` operator references an object, that object's properties can be exposed with the aid of the operator. For example, to send the `value` property of a text input object to a function, the `this` operator stands in for the current object reference and appends the proper syntax to reference the `value` property:

```
<INPUT TYPE="text" NAME="entry" onChange="process(this.value)">
```

The `this` operator also works inside other objects, such as custom objects. When you define a constructor function for a custom object, it is common to use the `this` operator to define properties of the object and assign values to those properties. Consider the following example of an object creation sequence:

```
function bottledWater(brand, ozSize, flavor) {
    this.brand = brand
    this.ozSize = ozSize
    this.flavor = flavor
}
var myWater = new bottledWater("Crystal Springs", 16, "original")
```

When the new object is created via the constructor function, the `this` operators define each property of the object and then assign the corresponding incoming value to that property. Using the same names for the properties and parameter variables is perfectly fine, and makes the constructor easy to maintain.

By extension, if you assign a function as an object property (to behave as a method for the object), the `this` operator inside that function refers to the object, offering an avenue to the object's properties. For example, if I add the following function definition and statement to the `myWater` object created just above, the function can directly access the `brand` property of the object:

```
function adSlogan() {  
    return "Drink " + this.brand + ", it's wet and wild!"  
}  
myWater.getSlogan = adSlogan
```

When a statement invokes the `myWater.getSlogan()` method, the object invokes the `adSlogan()` function, but all within the context of the `myWater` object. Thus, the `this` operator applies to the surrounding object, making the `brand` property available via the `this` operator (`this.brand`).

## Operator Precedence

When you start working with complex expressions that hold a number of operators (for example, Listing 32-2), knowing the order in which JavaScript evaluates those expressions is vital. JavaScript assigns different priorities or weights to types of operators in an effort to achieve uniformity in the way it evaluates complex expressions.

In the following expression

```
10 + 4 * 5 // result = 30
```

JavaScript uses its precedence scheme to perform the multiplication before the addition — regardless of where the operators appear in the statement. In other words, JavaScript first multiplies 4 by 5, and then adds that result to 10 to get a result of 30. That may not be the way you want this expression to evaluate. Perhaps your intention was to add the 10 and 4 first and then to multiply that sum by 5. To make that happen, you have to override JavaScript's natural operator precedence. To do that, you must enclose an operator with lower precedence in parentheses. The following statement shows how you'd adjust the previous expression to make it behave differently:

```
(10 + 4) * 5 // result = 70
```

That one set of parentheses has a great impact on the outcome. Parentheses have the highest precedence in JavaScript, and if you nest parentheses in an expression, the innermost set evaluates first.

For help in constructing complex expressions, refer to Table 32-10 for JavaScript's operator precedence. My general practice: When in doubt about complex precedence issues, I build the expression with lots of parentheses according to the way I want the internal expressions to evaluate.

**Table 32-10**  
**JavaScript Operator Precedence**

<i>Precedence Level</i>	<i>Operator</i>	<i>Notes</i>
1	()	From innermost to outermost
	[]	Array index value
	function()	Any remote function call
2	!	Boolean Not
	~	Bitwise Not
	-	Negation
	++	Increment
	--	Decrement
	typeof	
	void	
	delete	Delete array or object entry
3	*	Multiplication
	/	Division
	%	Modulo
4	+	Addition
	-	Subtraction
5	<<	Bitwise shifts
	>	
	>>	
6	<	Comparison operators
	<=	
	>	
	>=	
7	==	Equality
	!=	
8	&	Bitwise And
9	^	Bitwise XOR
10		Bitwise Or
11	&&	Boolean And
12		Boolean Or

*(continued)*

Table 32-10 (continued)

<i>Precedence Level</i>	<i>Operator</i>	<i>Notes</i>
13	?	Conditional expression
14	=	Assignment operators
	+=	
	-=	
	*=	
	/=	
	%=	
	<<=	
	>=	
	>>=	
	&=	
	^=	
	=	
15	,	Comma (parameter delimiter)

This precedence scheme is devised to help you avoid being faced with two operators from the same precedence level that often appear in the same expression. When it happens (such as with addition and subtraction), JavaScript begins evaluating the expression from left to right.

One related fact involves a string of Boolean expressions strung together for a condition statement (Listing 32-2). JavaScript follows what is called *short-circuit evaluation*. As the nested expressions are evaluated left to right, the fate of the entire condition can sometimes be determined before all expressions have been evaluated. Anytime JavaScript encounters an And operator, if the left operand evaluates to false, the entire expression evaluates to false without JavaScript even bothering to evaluate the right operand. For an Or operator, if the left operand is true, JavaScript short-circuits that expression to true. This feature can trip you up if you don't perform enough testing on your scripts: If a syntax error or other error exists in a right operand, and you fail to test the expression in a way that forces that right operand to evaluate, you may not know that a bug exists in your code. Users of your page, of course, will find the bug quickly. Do your testing to head off bugs at the pass.



Note

Notice, too, that all math and string concatenation is performed prior to any comparison operators. This enables all expressions that act as operands for comparisons to evaluate fully before they are compared.

The key to working with complex expressions is to isolate individual expressions and try them out by themselves, if you can. See additional debugging tips in Chapter 45.

