The Core JavaScript Language Guide

RadView Software

The Core JavaScript Language Guide

August, 1998 RadView Publication Number: WL-0898-PG04

For product assistance or information, contact:

Toll free in the US	1-888-RADVIEW
FAX	(781) 862-5287
World Wide Web	http://www.radview.com
E-mail	support@radview.com

USA:

RadView Software Inc.
1050 Waltham St.
Lexington MA, 02421
e-mail: sales@radview.com
Phone: (781) 860-7570

Contents

Chapter

1

Values, Variables, and Literals

This chapter discusses values that JavaScript recognizes and describes the fundamental building blocks of JavaScript expressions: variables and literals.

Values

JavaScript recognizes the following types of values:

- Numbers, such as 42 or 3.14159
- Logical (Boolean) values, either true or false
- Strings, such as "Howdy!"
- `null`, a special keyword denoting a null value

Note Because JavaScript is case sensitive, `null` is not the same as `Null`, `NULL`, or any other variant.

This relatively small set of types of values, or *data types*, enables you to perform useful functions with your applications. There is no explicit distinction between integer and real-valued numbers. Nor is there an explicit date data type in Navigator. However, you can use the `Date` object and its methods to handle dates.

Objects and functions are the other fundamental elements in the language. You can think of objects as named containers for values, and functions as procedures that your application can perform.

Data Type Conversion

JavaScript is a loosely typed language. That means you do not have to specify the data type of a variable when you declare it, and data types are converted automatically as needed during script execution. So, for example, you could define a variable as follows:

```
var answer = 42
```

And later, you could assign the same variable a string value, for example,

```
answer = "Thanks for all the fish..."
```

Because JavaScript is loosely typed, this assignment does not cause an error message.

In expressions involving numeric and string values, JavaScript converts the numeric values to strings. For example, consider the following statements:

```
x = "The answer is " + 42
y = 42 + " is the answer."
```

The first statement returns the string "The answer is 42." The second statement returns the string "42 is the answer."

Variables

You use variables as symbolic names for values in your application. You give variables names by which you refer to them and which must conform to certain rules.

A JavaScript identifier, or *name,* must start with a letter or underscore ("_"); subsequent characters can also be digits (0-9). Because JavaScript is case sensitive, letters include the characters "A" through "Z" (uppercase) and the characters "a" through "z" (lowercase).

Some examples of legal names are `Number_hits`, `temp99`, and `_name`.

Variable Scope

You can declare a variable in two ways:

- By simply assigning it a value; for example, `x = 42`
- With the keyword `var`; for example, `var x = 42`

When you set a variable identifier by assignment outside of a function, it is called a *global* variable, because it is available everywhere in the current document. When you declare a variable within a function, it is called a *local* variable, because it is available only within the function.

Using `var` to declare a global variable is optional. However, you must use `var` to declare a variable inside a function.

You can access global variables declared in one window or frame from another window or frame by specifying the window or frame name. For example, if a variable called `phoneNumber` is declared in a `FRAMESET` document, you can refer to this variable from a child frame as `parent.phoneNumber`.

Literals

You use literals to represent values in JavaScript. These are fixed values, not variables, that you *literally* provide in your script.

Integers

Integers can be expressed in decimal (base 10), hexadecimal (base 16), and octal (base 8). A decimal integer literal consists of a sequence of digits without a leading 0 (zero). A leading 0 (zero) on an integer literal indicates it is in octal; a leading 0x (or 0X) indicates hexadecimal. Hexadecimal integers can include digits (0-9) and the letters a-f and A-F. Octal integers can include only the digits 0-7.

Some examples of integer literals are: 42 0xFFF, and -345.

Floating-Point Literals

A floating-point literal can have the following parts:
- a decimal integer
- a decimal point (“.”)
- a fraction (another decimal number)
- an exponent
- a type suffix

The exponent part is an “e” or “E” followed by an integer, which can be signed (preceded by “+” or “-”). A floating-point literal must have at least one digit and either a decimal point or “e” (or “E”).

Some examples of floating-point literals are 3.1415, -3.1E12, .1e12, and 2E-12

Boolean Literals

The Boolean type has two literal values: `true` and `false`.

String Literals

A string literal is zero or more characters enclosed in double (") or single (') quotation marks. A string must be delimited by quotation marks of the same type; that is, either both single quotation marks or both double quotation marks. The following are examples of string literals:
- `"blah"`
- `'blah'`
- `"1234"`
- `"one line \n another line"`

In addition to ordinary characters, you can also include special characters in strings, as shown in the last example in the preceding list. Table 1.1 lists the special characters that you can use in JavaScript strings.

Table 1.1 JavaScript special characters

Character	Meaning
`\b`	backspace
`\f`	form feed
`\n`	new line
`\r`	carriage return
`\t`	tab
`\\`	backslash character

Escaping Characters

For characters not listed in Table 1.1, a preceding backslash is ignored, with the exception of a quotation mark and the backslash character itself.

You can insert a quotation mark inside a string by preceding it with a backslash. This is known as *escaping* the quotation mark. For example,

```
var quote = "He read \"The Cremation of Sam McGee\" by R.W. Service."
document.write(quote)
```

The result of this would be

> He read "The Cremation of Sam McGee" by R.W. Service.

To include a literal backslash inside a string, you must escape the backslash character. For example, to assign the file path `c:\temp` to a string, use the following:

```
var home = "c:\\temp"
```

Array Literals

As discussed in Chapter 4, "Predefined Core Objects and Functions," JavaScript provides array objects. In earlier versions of Navigator, you could create arrays only using the array constructor function. In Navigator 4.0, however, you can also create arrays using literal notation. The syntax for creating a literal array is:

```
arrayName = [element0, element1, ..., elementN]
```

where `arrayName` is the name of the new array and each `elementI` is a value for on of the array's elements. When you create an array using literal notation, it is initialized with the specified values as its elements, and its length is set to the number of arguments.

You do not have to specify all elements in the new array. If you put 2 commas in a row, the array is created with spaces for the unspecified elements, as shown in the second example below.

If an array is created using literal notation in a top-level script, JavaScript interprets the object each time it evaluates the expression containing the array literal. In addition, a literal used in a function is created each time the function is called.

The following example creates the `coffees` array with three elements and a length of three:

```
coffees = ["French Roast", "Columbian", "Kona"]
```

The following example creates the `fish` array, giving values to two elements and having one empty element:

```
fish = ["Lion", , "Surgeon"]
```

With this expression `fish[0]` is "Lion", `fish[2]` is "Surgeon", and `fish[1]` is undefined.

Object Literals

In earlier versions of Navigator, you could create objects only using their constructor functions or using a function supplied by some other object for that purpose. In Navigator 4.0, however, you can also create objects using literal notation. The syntax for creating a literal object is:

```
objectName = {property1:value1, property2:value2,..., propertyN:valueN}
```

where `objectName` is the name of the new object, each `property`*`I`* is an identifier (either a name, a number, or a string literal), and each `value`*`I`* is an expression whose value is assigned to the `property`*`I`*.

If an object is created using literal notation in a top-level script, JavaScript interprets the object each time it evaluates the expression containing the object literal. In addition, a literal used in a function is created each time the function is called.

Assume you have the following statement:

```
if (cond) x = {hi:"there"}
```

In this case, JavaScript makes the literal object and assigns it to the variable `x` if and only if the expression `cond` is true.

The following example creates `myHonda` with three properties. Note that the `engine` property is also an object with its own properties.

```
myHonda = {color:"red",wheels:4,engine:{cylinders:4,size:2.2}}
```

Regular Expression Literals

Navigator 4.0 introduces regular expressions for doing pattern matching in strings. Regular expressions can be constructed using a literal notation. For more information, see “Regular Expressions” on page 26.

Chapter

2

Expressions and Operators

This chapter describes JavaScript expressions and operators, including assignment, comparison, arithmetic, bitwise, logical, string, and special operators. It also describes regular expressions.

Expressions

An *expression* is any valid set of literals, variables, operators, and expressions that evaluates to a single value; the value can be a number, a string, or a logical value.

Conceptually, there are two types of expressions: those that assign a value to a variable, and those that simply have a value. For example, the expression `x = 7` is an expression that assigns x the value seven. This expression itself evaluates to seven. Such expressions use *assignment operators*. On the other hand, the expression `3 + 4` simply evaluates to seven; it does not perform an assignment. The operators used in such expressions are referred to simply as *operators*.

JavaScript has the following types of expressions:

- Arithmetic: evaluates to a number, for example 3.14159
- String: evaluates to a character string, for example, “Fred” or “234”
- Logical: evaluates to true or false

The special keyword `null` denotes a null value. In contrast, variables that have not been assigned a value are *undefined* and cause a runtime error if used as numbers or as numeric variables. Array elements that have not been assigned a value, however, evaluate to false. For example, the following code executes the function `myFunction` because the array element is not defined:

```
myArray=new Array()
if (!myArray["notThere"])
   myFunction()
```

Operators

JavaScript has assignment, comparison, arithmetic, bitwise, logical, string, and special operators. This section describes the operators and contains information about operator precedence.

JavaScript has both *binary* and *unary* operators. A binary operator requires two operands, one before the operator and one after the operator:

```
operand1 operator operand2
```

For example, `3+4` or `x*y`.

A unary operator requires a single operand, either before or after the operator:

```
operator operand
```

or

```
operand operator
```

For example, `x++` or `++x`.

In addition, JavaScript has one ternary operator, the conditional operator. A ternary operator requires three operands.

Assignment Operators

An assignment operator assigns a value to its left operand based on the value of its right operand. The basic assignment operator is equal (=), which assigns the value of its right operand to its left operand. That is, x = y assigns the value of y to x.

The other assignment operators are shorthand for standard operations, as shown in Table 2.1.

Table 2.1 Assignment operators

Shorthand operator	Meaning
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x %= y`	`x = x % y`
`x <<= y`	`x = x << y`
`x >>= y`	`x = x >> y`
`x >>>= y`	`x = x >>> y`
`x &= y`	`x = x & y`
`x ^= y`	`x = x ^ y`
`x |= y`	`x = x | y`

Comparison Operators

A comparison operator compares its operands and returns a logical value based on whether the comparison is true or not. The operands can be numerical or string values. When used on string values, the comparisons are based on the standard lexicographical ordering. They are described in Table 2.2.

Table 2.2 Comparison operators

Operator	Description	Examples returning true[a]
Equal (==)	Returns true if the operands are equal.	`3 == var1`
Not equal (!=)	Returns true if the operands are not equal.	`var1 != 4`
Greater than (>)	Returns true if left operand is greater than right operand.	`var2 > var1`
Greater than or equal (>=)	Returns true if left operand is greater than or equal to right operand.	`var2 >= var1` `var1 >= 3`
Less than (<)	Returns true if left operand is less than right operand.	`var1 < var2`
Less than or equal (<=)	Returns true if left operand is less than or equal to right operand.	`var1 <= var2` `var2 <= 5`

a. In these examples, assume `var1` has been assigned the value 3 and `var2` had been assigned the value 4.

Arithmetic Operators

Arithmetic operators take numerical values (either literals or variables) as their operands and return a single numerical value. The standard arithmetic operators are addition (+), subtraction (-), multiplication (*), and division (/). These operators work as they do in other programming languages.

In addition, JavaScript provides the arithmetic operators listed in Table 2.3.

Table 2.3 Arithmetic Operators

Operator	Description	Example
% (Modulus)	Binary operator. Returns the integer remainder of dividing the 2 operands.	12 % 5 returns 2.
++ (Increment)	Unary operator. Adds one to its operand. If used as a prefix operator (++x), returns the value of its operand after adding one; if used as a postfix operator (x++), returns the value of its operand before adding one.	If x is 3, then ++x sets x to 4 and returns 4, whereas x++ sets x to 4 and returns 3.

Table 2.3 Arithmetic Operators

Operator	Description	Example
`--` (Decrement)	Unary operator. Subtracts one to its operand. The return value is analogous to that for the increment operator.	If `x` is 3, then `--x` sets `x` to 2 and returns 2, whereas `x++` sets `x` to 2 and returns 3.
`-` (Unary negation)	Unary operator. Returns the negation of its operand.	If `x` is 3, then `-x` returns -3.

Bitwise Operators

Bitwise operators treat their operands as a set of bits (zeros and ones), rather than as decimal, hexadecimal, or octal numbers. For example, the decimal number nine has a binary representation of 1001. Bitwise operators perform their operations on such binary representations, but they return standard JavaScript numerical values.

The following table summarizes JavaScript's bitwise operators

Table 2.4 Bitwise operators

Operator	Usage	Description
Bitwise AND	`a & b`	Returns a one in each bit position if bits of both operands are ones.
Bitwise OR	`a | b`	Returns a one in a bit if bits of either operand is one.
Bitwise XOR	`a ^ b`	Returns a one in a bit position if bits of one but not both operands are one.
Bitwise NOT	`~ a`	Flips the bits of its operand.
Left shift	`a << b`	Shifts `a` in binary representation `b` bits to left, shifting in zeros from the right.
Sign-propagating right shift	`a >> b`	Shifts `a` in binary representation `b` bits to right, discarding bits shifted off.
Zero-fill right shift	`a >>> b`	Shifts `a` in binary representation `b` bits to the right, discarding bits shifted off, and shifting in zeros from the left.

Bitwise Logical Operators

Conceptually, the bitwise logical operators work as follows:

- The operands are converted to thirty-two-bit integers and expressed by a series of bits (zeros and ones).
- Each bit in the first operand is paired with the corresponding bit in the second operand: first bit to first bit, second bit to second bit, and so on.
- The operator is applied to each pair of bits, and the result is constructed bitwise.

For example, the binary representation of nine is 1001, and the binary representation of fifteen is 1111. So, when the bitwise operators are applied to these values, the results are as follows:

- 15 & 9 yields 9 (1111 & 1001 = 1001)
- 15 | 9 yields 15 (1111 | 1001 = 1111)
- 15 ^ 9 yields 6 (1111 ^ 1001 = 0110)

Bitwise Shift Operators

The bitwise shift operators take two operands: the first is a quantity to be shifted, and the second specifies the number of bit positions by which the first operand is to be shifted. The direction of the shift operation is controlled by the operator used.

Shift operators convert their operands to thirty-two-bit integers and return a result of the same type as the left operator.

The shift operators are listed in Table 2.5.

Table 2.5 Bitwise shift operators

Operator	Description	Example
<< (Left shift)	This operator shifts the first operand the specified number of bits to the left. Excess bits shifted off to the left are discarded. Zero bits are shifted in from the right.	`9<<2` yields 36, because 1001 shifted 2 bits to the left becomes 100100, which is 36.
>> (Sign-propagating right shift)	This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Copies of the leftmost bit are shifted in from the left.	9>>2 yields 2, because 1001 shifted 2 bits to the right becomes 10, which is 2. Likewise, -9>>2 yields -3, because the sign is preserved.
>>> (Zero-fill right shift)	This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Zero bits are shifted in from the left.	19>>>2 yields 4, because 10011 shifted 2 bits to the right becomes 100, which is 4. For non-negative numbers, zero-fill right shift and sign-propagating right shift yield the same result.

Logical Operators

Logical operators take Boolean (logical) values as operands and return a Boolean value. They are described in Table 2.6.

Table 2.6 Logical operators

Operator	Usage	Description	Example[a]
and (&&)	`expr1 && expr2`	Returns `expr1` if it converts to `false`. Otherwise, returns `expr2`.	`var1 && var2` returns `"Dog"`. `var2 && var3` returns `false`
or (||)	`expr1 || expr2`	Returns `expr1` if it converts to `true`. Otherwise, returns `expr2`.	`var1 || var2` returns `"Cat"`. `var3 || var1` returns `"Cat"`. `var3 || (3==4)` returns `false`.
not (!)	`!expr`	If `expr` is true, returns false; if `expr` is false, returns true.	`!var1` returns `false`. `!var3` returns `true`.

a. Assume `var1` is `"Cat"`, `var2` is `"Dog"`, and `var3` is `false`.

Short-Circuit Evaluation

As logical expressions are evaluated left to right, they are tested for possible "short-circuit" evaluation using the following rules:

- `false` && *anything* is short-circuit evaluated to false.
- `true` || *anything* is short-circuit evaluated to true.

The rules of logic guarantee that these evaluations are always correct. Note that the *anything* part of the above expressions is not evaluated, so any side effects of doing so do not take effect.

String Operators

In addition to the comparison operators, which can be used on string values, the concatenation operator (+) concatenates two string values together, returning another string that is the union of the two operand strings. For example, `"my " + "string"` returns the string `"my string"`.

The shorthand assignment operator += can also be used to concatenate strings. For example, if the variable `mystring` has the value "alpha," then the expression `mystring += "bet"` evaluates to "alphabet" and assigns this value to `mystring`.

Special Operators

conditional operator

The conditional operator is the only JavaScript operator that takes 3 operands. The operator can have one of two values based on a condition. The syntax is

```
(condition) ? val1 : val2
```

If `condition` is true, the operator has the value of `val1`. Otherwise it has the value of `val2`. You can use the conditional operator anywhere you would use a standard operator.

For example,

```
status = (age >= 18) ? "adult" : "minor"
```

This statement assigns the value "adult" to the variable `status` if `age` is eighteen or more. Otherwise, it assigns the value "minor" to `status`.

comma operator

The comma operator (`,`) simply evaluates both of its operands and returns the value of the second operand. This operator is primarily used inside a `for` loop, to allow multiple variables to be updated each time through the loop.

For example, if `a` is a 2-dimensional array with 10 elements on a side, the following code uses the comma operator to increment two variables at once. The code prints the values of the diagonal elements in the array:

```
for (var i=0, j=10; i <= 10; i++, j--)
   document.writeln("a["+i+","+j+"]= " + a[i,j])
```

delete

The `delete` operator deletes an object's property or an element at a specified index in an array. Its syntax is:

```
delete objectName.property
delete objectname[index]
delete property
```

where `objectName` is the name of an object, `property` is an existing property, and `index` is an integer representing the location of an element in an array

The third form is legal only within a `with` statement. If the deletion succeeds, the `delete` operator sets the property or element to `undefined`. `delete` always returns undefined.

new

You can use the `new` operator to create an instance of a user-defined object type or of one of the predefined object types `Array`, `Boolean`, `Date`, `Function`, `Image`, `Number`, `Object`, `Option`, `RegExp`, or `String`. On the server, you can also use it with `DbPool`, `Lock`, `File`, or `SendMail`. Use `new` as follows:

```
objectName = new objectType ( param1 [,param2] ...[,paramN] )
```

For more information, see `new` in the *JavaScript Reference*.

typeof

The `typeof` operator is used in either of the following ways:

```
1. typeof operand
2. typeof (operand)
```

The `typeof` operator returns a string indicating the type of the unevaluated operand. `operand` is the string, variable, keyword, or object for which the type is to be returned. The parentheses are optional.

Suppose you define the following variables:

```
var myFun = new Function("5+2")
var shape="round"
var size=1
var today=new Date()
```

The `typeof` operator returns the following results for these variables:

```
typeof myFun is object
typeof shape is string
typeof size is number
typeof today is object
typeof dontExist is undefined
```

For the keywords `true` and `null`, the `typeof` operator returns the following results:

```
typeof true is boolean
typeof null is object
```

For a number or string, the `typeof` operator returns the following results:

```
typeof 62 is number
typeof 'Hello world' is string
```

For property values, the `typeof` operator returns the type of value the property contains:

```
typeof document.lastModified is string
typeof window.length is number
typeof Math.LN2 is number
```

For methods and functions, the `typeof` operator returns results as follows:

```
typeof blur is function
typeof eval is function
typeof parseInt is function
typeof shape.split is function
```

For predefined objects, the `typeof` operator returns results as follows:

```
typeof Date is function
typeof Function is function
typeof Math is function
typeof Option is function
typeof String is function
```

void

The void operator is used in either of the following ways:

```
1. javascript:void (expression)
2. javascript:void expression
```

The void operator specifies an expression to be evaluated without returning a value. `expression` is a JavaScript expression to evaluate. The parentheses surrounding the expression are optional, but it is good style to use them.

You can use the `void` operator to specify an expression as a hypertext link. The expression is evaluated but is not loaded in place of the current document.

The following code creates a hypertext link that does nothing when the user clicks it. When the user clicks the link, `void(0)` evaluates to 0, but that has no effect in JavaScript.

```
<A HREF="javascript:void(0)">Click here to do nothing</A>
```

The following code creates a hypertext link that submits a form when the user clicks it.

```
<A HREF="javascript:void(document.form.submit())">
Click here to submit</A>
```

Operator Precedence

The *precedence* of operators determines the order they are applied when evaluating an expression. You can override operator precedence by using parentheses.

Table 2.7 describes the precedence of operators, from lowest to highest.

Table 2.7 Operator precedence

Operator type	Individual operators
assignment	`= += -= *= /= %= <<= >>= >>>= &= ^= |=`
conditional	`?:`
logical-or	`||`
logical-and	`&&`
bitwise-or	`|`
bitwise-xor	`^`
bitwise-and	`&`
equality	`== !=`
relational	`< <= > >=`
bitwise shift	`<< >> >>>`
addition/subtraction	`+ -`
multiply/divide	`* / %`
negation/increment	`! ~ - ++ -- typeof void`
call, member	`() [] .`

Regular Expressions

JavaScript 1.2, available in Navigator 4.0, adds regular expressions to the language. Regular expressions are patterns used to match character combinations in strings. In JavaScript, regular expressions are also objects. For example, to search for all occurrences of 'the' in a string, you create a pattern consisting of 'the' and use the pattern to search for its match in a string. Regular expression patterns can be constructed using either literal notation (for example, `/abc/`) or the `RegExp` constructor function (for example, `re = new RegExp("abc")`). These patterns are used with the `exec` and `test` methods of regular expressions, and with the `match`, `replace`, `search`, and `split` methods of `String`.

Creating a Regular Expression

You construct a regular expression in one of two ways:

- Using literal notation, as in:

  ```
  re = /ab+c/
  ```

  Literal notation provides compilation of the regular expression when the script is evaluated. When the regular expression will remain constant use literal notation for better performance.

- Calling the constructor function of the `RegExp` object, as in:

  ```
  re = new RegExp("ab+c")
  ```

  Using the constructor function provides runtime compilation of the regular expression. Use the constructor function when you know the regular expression pattern will be changing, or you don't know the pattern and are getting it from another source, such as user input. Once you have a defined regular expression, if the regular expression is used throughout the script, and if its source changes, you can use the `compile` method to compile a new regular expression for efficient reuse.

Writing a Regular Expression Pattern

A regular expression pattern is composed of simple characters, such as `/abc/`, or a combination of simple and special characters, such as `/ab*c/` or `/Chapter (\d+)\.\d*/`. The last example includes parentheses which are used as a memory device. The match made with this part of the pattern is remembered for later use, as described in "Using Parenthesized Substring Matches" on page 34.

Using Simple Patterns

Simple patterns are constructed of characters for which you want to find a direct match. For example, the pattern `/abc/` matches character combinations in strings only when exactly the characters 'abc' occur together and in that order. Such a match would succeed in the strings "Hi, do you know your abc's?" and "The latest airplane designs evolved from slabcraft." In both cases the match is with the substring 'abc'. There is no match in the string "Grab crab" because it does not contain the substring 'abc'.

Using Special Characters

When the search for a match requires something more than a direct match, such as finding one or more b's, or finding whitespace, the pattern includes special characters. For example, the pattern `/ab*c/` matches any character combination in which a single 'a' is followed by zero or more 'b's (`*` means 0 or more occurrences of the preceding character) and then immediately followed by 'c'. In the string "cbbabbbbcdebc," the pattern matches the substring 'abbbbc'.

Table 2.8 provides a complete list and description of the special characters that can be used in regular expressions.

Table 2.8 Special characters in regular expressions.

Character	Meaning
\	For characters that are usually treated literally, indicates that the next character is special and not to be interpreted literally. For example, `/b/` matches the character 'b'. By placing a backslash in front of b, that is by using `/\b/`, the character becomes special to mean match a word boundary. -or- For characters that are usually treated specially, indicates that the next character is not special and should be interpreted literally. For example, `*` is a special character that means 0 or more occurrences of the preceding character should be matched; for example, `/a*/` means match 0 or more a's. To match `*` literally, precede the it with a backslash; for example, `/a\*/` matches 'a*'.
^	Matches beginning of input or line. For example, `/^A/` does not match the 'A' in "an A," but does match it in "An A."
$	Matches end of input or line. For example, `/t$/` does not match the 't' in "eater", but does match it in "eat"
*	Matches the preceding character 0 or more times. For example, `/bo*/` matches 'boooo' in "A ghost booooed" and 'b' in "A bird warbled", but nothing in "A goat bleated".
+	Matches the preceding character 1 or more times. Equivalent to `{1,}`. For example, `/a+/` matches the 'a' in "candy" and all the a's in "caaaaaaandy."

Table 2.8 Special characters in regular expressions. (Continued)

Character	Meaning
`?`	Matches the preceding character 0 or 1 time. For example, `/e?le?/` matches the 'el' in "angel" and the 'le' in "angle."
`.`	(The decimal point) matches any single character except the newline character. For example, `/.n/` matches 'an' and 'on' in "nay, an apple is on the tree", but not 'nay'.
`(x)`	Matches 'x' and remembers the match. For example, `/(foo)/` matches and remembers 'foo' in "foo bar." The matched substring can be recalled from the resulting array's elements `[1]`, ..., `[n]`, or from the predefined `RegExp` object's properties `$1`, ..., `$9`.
`x|y`	Matches either 'x' or 'y'. For example, `/green|red/` matches 'green' in "green apple" and 'red' in "red apple."
`{n}`	Where `n` is a positive integer. Matches exactly `n` occurrences of the preceding character. For example, `/a{2}/` doesn't match the 'a' in "candy," but it matches all of the a's in "caandy," and the first two a's in "caaandy."
`{n,}`	Where `n` is a positive integer. Matches at least `n` occurrences of the preceding character. For example, `/a{2,}` doesn't match the 'a' in "candy", but matches all of the a's in "caandy" and in "caaaaaaandy."
`{n,m}`	Where `n` and `n` are positive integers. Matches at least `n` and at most `n` occurrences of the preceding character. For example, `/a{1,3}/` matches nothing in "cndy", the 'a' in "candy," the first two a's in "caandy," and the first three a's in "caaaaaaandy" Notice that when matching "caaaaaaandy", the match is "aaa", even though the original string had more a's in it.
`[xyz]`	A character set. Matches any one of the enclosed characters. You can specify a range of characters by using a hyphen. For example, `[abcd]` is the same as `[a-c]`. They match the 'b' in "brisket" and the 'c' in "ache".

Table 2.8 Special characters in regular expressions. (Continued)

Character	Meaning
`[^xyz]`	A negated or complemented character set. That is, it matches anything that is not enclosed in the brackets. You can specify a range of characters by using a hyphen. For example, `[^abc]` is the same as `[^a-c]`. They initially match 'r' in "brisket" and 'h' in "chop."
`[\b]`	Matches a backspace. (Not to be confused with `\b`.)
`\b`	Matches a word boundary, such as a space. (Not to be confused with `[\b]`.) For example, `/\bn\w/` matches the 'no' in "noonday";`/\wy\b/` matches the 'ly' in "possibly yesterday."
`\B`	Matches a non-word boundary. For example, `/\w\Bn/` matches 'on' in "noonday", and `/y\B\w/` matches 'ye' in "possibly yesterday."
`\c`*X*	Where *X* is a control character. Matches a control character in a string. For example, `/\cM/` matches control-M in a string.
`\d`	Matches a digit character. Equivalent to `[0-9]`. For example, `/\d/` or `/[0-9]/` matches '2' in "B2 is the suite number."
`\D`	Matches any non-digit character. Equivalent to `[^0-9]`. For example, `/\D/` or `/[^0-9]/` matches 'B' in "B2 is the suite number."
`\f`	Matches a form-feed.
`\n`	Matches a linefeed.
`\r`	Matches a carriage return.
`\s`	Matches a single white space character, including space, tab, form feed, line feed. Equivalent to `[ \f\n\r\t\v]`. for example, `/\s\w*/` matches ' bar' in "foo bar."
`\S`	Matches a single character other than white space. Equivalent to `[^ \f\n\r\t\v]`. For example, `/\S/\w*` matches 'foo' in "foo bar."
`\t`	Matches a tab
`\v`	Matches a vertical tab.

Table 2.8 Special characters in regular expressions. (Continued)

Character	Meaning
`\w`	Matches any alphanumeric character including the underscore. Equivalent to `[A-Za-z0-9_]`. For example, `/\w/` matches 'a' in "apple," '5' in "$5.28," and '3' in "3D."
`\W`	Matches any non-word character. Equivalent to `[^A-Za-z0-9_]`. For example, `/\W/` or `/[^$A-Za-z0-9_]/` matches '%' in "50%."
`\`*`n`	Where *n* is a positive integer. A back reference to the last substring matching the *n* parenthetical in the regular expression (counting left parentheses). For example, `/apple(,)\sorange\1/` matches 'apple, orange', in "apple, orange, cherry, peach." A more complete example follows this table. **Note:** If the number of left parentheses is less than the number specified in *n*, the *n* is taken as an octal escape as described in the next row.
`\o`*`octal`* `\x`*`hex`*	Where `\o`*`octal`* is an octal escape value or `\x`*`hex`* is a hexadecimal escape value. Allows you to embed ASCII codes into regular expressions.

Using Parentheses

Parentheses around any part of the regular expression pattern cause that part of the matched substring to be remembered. Once remembered, the substring can be recalled for other use, as described in "Using Parenthesized Substring Matches" on page 34.

For example, the pattern `/Chapter (\d+)\.\d*/` illustrates additional escaped and special characters and indicates that part of the pattern should be remembered. It matches precisely the characters 'Chapter ' followed by one or more numeric characters (`\d` means any numeric character and `+` means 1 or more times), followed by a decimal point (which in itself is a special character; preceding the decimal point with \ means the pattern must look for the literal character '.'), followed by any numeric character 0 or more times (`\d` means numeric character, `*` means 0 or more times). In addition, parentheses are used to remember the first matched numeric characters.

This pattern is found in "Open Chapter 4.3, paragraph 6" and '4' is remembered. The pattern is not found in "Chapters 3 and 4", because that string does not have a period after the '3'.

Working With Regular Expressions

Regular expressions are used with the regular expression methods `test` and `exec` and with the `String` methods `match`, `replace`, `search`, and `split`. These methods are explained in detail in the *JavaScript Reference*.

`exec`	A regular expression method that executes a search for a match in a string. It returns an array of information.
`test`	A regular expression method that tests for a match in a string. It returns true or false.
`match`	A `String` method that executes a search for a match in a string. It returns an array of information or null on a mismatch.
`search`	A `String` method that tests for a match in a string. It returns the index of the match, or -1 if the search fails.
`replace`	A `String` method that executes a search for a match in a string, and replaces the matched substring with a replacement substring.
`split`	A `String` method that uses a regular expression or a fixed string to break a string into an array of substrings.

When you want to know whether a pattern is found in a string, use the `test` or `search` method; for more information (but slower execution) use the `exec` or `match` methods. If you use `exec` or `match` and if the match succeeds, these methods return an array and update properties of the associated regular expression object and also of the predefined regular expression object, `RegExp`. If the match fails, the `exec` method returns `null` (which converts to `false`).

In the following example, the script uses the `exec` method to find a match in a string.

```
<SCRIPT LANGUAGE="JavaScript1.2">
myRe=/d(b+)d/g;
myArray = myRe.exec("cdbbdbsbz");
</SCRIPT>
```

If you do not need to access the properties of the regular expression, an alternative way of creating `myArray` is with this script:

```
<SCRIPT LANGUAGE="JavaScript1.2">
myArray = /d(b+)d/g.exec("cdbbdbsbz");
</SCRIPT>
```

If you want to be able to recompile the regular expression, yet another alternative is this script:

```
<SCRIPT LANGUAGE="JavaScript1.2">
myRe= new RegExp ("d(b+)d", "g:);
myArray = myRe.exec("cdbbdbsbz");
</SCRIPT>
```

With these scripts, the match succeeds and returns the array and updates the properties shown in Table 2.9.

Table 2.9 Results of regular expression execution.

Object	Property or Index	Description	In this example
`myArray`		The matched string and all remembered substrings	`["dbbd", "bb"]`
	`index`	The 0-based index of the match in the input string	`1`
	`input`	The original string	`"cdbbdbsbz"`
	`[0]`	The last matched characters	`"dbbd"`
`myRe`	`lastIndex`	The index at which to start the next match. (This property is set only if the regular expression uses the `g` option, described in "Executing a Global Search and Ignoring Case" on page 36.)	`5`
	`source`	The text of the pattern	`"d(b+)d"`
`RegExp`	`lastMatch`	The last matched characters	`"dbbd"`
	`leftContext`	The substring preceding the most recent match	`"c"`
	`rightContext`	The substring following the most recent match	`"bsbz"`

`RegExp.leftContext` and `RegExp.rightContext` can be computed from the other values. `RegExp.leftContext` is equivalent to:

```
myArray.input.substring(0, myArray.index)
```

and `RegExp.rightContext` is equivalent to:

```
myArray.input.substring(myArray.index + myArray[0].length)
```

As shown in the second form of this example, you can use the literal form of a regular expression without assigning it to a variable. If you do, however, every occurrence of the literal is a new regular expression. For this reason, if you use

the literal form without assigning it to a variable, you cannot subsequently access the properties of that regular expression. For example, assume you have this script:

```
<SCRIPT LANGUAGE="JavaScript1.2">
myRe=/d(b+)d/g;
myArray = myRe.exec("cdbbdbsbz");
document.writeln("The value of lastIndex is " + myRe.lastIndex);
</SCRIPT>
```

This script displays:

The value of lastIndex is 5

However, if you have this script:

```
<SCRIPT LANGUAGE="JavaScript1.2">
myArray = /d(b+)d/g.exec("cdbbdbsbz");
document.writeln("The value of lastIndex is " + /d(b+)d/g.lastIndex);
</SCRIPT>
```

It displays:

The value of lastIndex is 0

The occurrences of `/d(b+)d/g` in the two statements are different regular expression objects and hence have different values for their `lastIndex` property. If you need to access the properties of a literal regular expression, you should first assign it to a variable.

Using Parenthesized Substring Matches

Including parentheses in a regular expression pattern causes the corresponding submatch to be remembered. For example, `/a(b)c/` matches the characters 'abc' and remembers 'b'. To recall these parenthesized substring matches, use the `RegExp` properties `$1`, ..., `$9` or the `Array` elements `[1]`, ..., `[n]`.

The number of possible parenthesized substrings is unlimited. The predefined `RegExp` object holds up to the last nine and the returned array holds all that were found. The following examples illustrate how to use parenthesized substring matches.

Example 1. The following script uses the `replace` method to switch the words in the string. For the replacement text, the script uses the values of the `$1` and `$2` properties.

```
<SCRIPT LANGUAGE="JavaScript1.2">
re = /(\w+)\s(\w+)/;
str = "John Smith";
newstr = str.replace(re, "$2, $1");
document.write(newstr)
</SCRIPT>
```

This prints "Smith, John".

Example 2. In the following example, `RegExp.input` is set by the Change event. In the `getInfo` function, the `exec` method uses the value of `RegExp.input` as its argument. Note that `RegExp` must be prepended to its `$` properties (because they appear outside the replacement string). (Example 3 is a more efficient, though possibly more cryptic, way to accomplish the same thing.)

```
<HTML>

<SCRIPT LANGUAGE="JavaScript1.2">
function getInfo(){
   re = /(\w+)\s(\d+)/
   re.exec();
   window.alert(RegExp.$1 + ", your age is " + RegExp.$2);
}
</SCRIPT>

Enter your first name and your age, and then press Enter.

<FORM>
<INPUT TYPE="text" NAME="NameAge" onChange="getInfo(this);">
</FORM>

</HTML>
```

Example 3. The following example is similar to Example 2. Instead of using the `RegExp.$1` and `RegExp.$2`, this example creates an array and uses `a[1]` and `a[2]`. It also uses the shortcut notation for using the `exec` method.

```
<HTML>

<SCRIPT LANGUAGE="JavaScript1.2">
function getInfo(){
   a = /(\w+)\s(\d+)/();
   window.alert(a[1] + ", your age is " + a[2]);
}
</SCRIPT>

Enter your first name and your age, and then press Enter.

<FORM>
<INPUT TYPE="text" NAME="NameAge" onChange="getInfo(this);">
</FORM>

</HTML>
```

Executing a Global Search and Ignoring Case

Regular expressions have two optional flags that allow for global and case insensitive searching. To indicate a global search, use the `g` flag. To indicate a case insensitive search, use the `i` flag. These flags can be used separately or together in either order, and are included as part of the regular expression.

To include a flag with the regular expression, use this syntax:

```
re = /pattern/[g|i|gi]
re = new RegExp("pattern", ['g'|'i'|'gi'])
```

Note that the flags, `i` and `g`, are an integral part of a regular expression. They cannot be added or removed later.

For example, `re = /\w+\s/g` creates a regular expression that looks for one or more characters followed by a space, and it looks for this combination throughout the string.

```
<SCRIPT LANGUAGE="JavaScript1.2">
re = /\w+\s/g;
str = "fee fi fo fum";
myArray = str.match(re);
document.write(myArray);
</SCRIPT>
```

This displays ["fee ", "fi ", "fo "]. In this example, you could replace the line:

```
re = /\w+\s/g;
```

with:

```
re = new RegExp("\\w+\\s", "g");
```

and get the same result.

Examples

Changing the Order in an Input String

The following example illustrates the formation of regular expressions and the use of `string.split()` and `string.replace()`.

It cleans a roughly-formatted input string containing names (first name first) separated by blanks, tabs and exactly one semicolon.

Finally, it reverses the name order (last name first) and sorts the list.

```
<SCRIPT LANGUAGE="JavaScript1.2">

// The name string contains multiple spaces and tabs,
// and may have multiple spaces between first and last names.
names = new String ( "Harry Trump ;Fred Barney; Helen Rigby ;\
       Bill Abel ;Chris Hand ")

document.write ("---------- Original String" + "<BR>" + "<BR>")
document.write (names + "<BR>" + "<BR>")

// Prepare two regular expression patterns and array storage.
// Split the string into array elements.

// pattern: possible white space then semicolon then possible white space
pattern = /\s*;\s*/

// Break the string into pieces separated by the pattern above and
// and store the pieces in an array called nameList
nameList = names.split (pattern)

// new pattern: one or more characters then spaces then characters.
// Use parentheses to "memorize" portions of the pattern.
// The memorized portions are referred to later.
pattern = /(\w+)\s+(\w+)/

// New array for holding names being processed.
bySurnameList = new Array;

// Display the name array and populate the new array
// with comma-separated names, last first.
//
// The replace method removes anything matching the pattern
// and replaces it with the memorized string—second memorized portion
// followed by comma space followed by first memorized portion.
//
// The variables $1 and $2 refer to the portions
// memorized while matching the pattern.

document.write ("---------- After Split by Regular Expression" + "<BR>")
for ( i = 0; i < nameList.length; i++) {
   document.write (nameList[i] + "<BR>")
   bySurnameList[i] = nameList[i].replace (pattern, "$2, $1")
}

// Display the new array.
document.write ("---------- Names Reversed" + "<BR>")
for ( i = 0; i < bySurnameList.length; i++) {
   document.write (bySurnameList[i] + "<BR>")
}
```

```
// Sort by last name, then display the sorted array.
bySurnameList.sort()
document.write ("---------- Sorted" + "<BR>")
for ( i = 0; i < bySurnameList.length; i++) {
   document.write (bySurnameList[i] + "<BR>")
}

document.write ("---------- End" + "<BR>")

</SCRIPT>
```

Using Special Characters to Verify Input

In the following example, a user enters a phone number. When the user presses Enter, the script checks the validity of the number. If the number is valid (matches the character sequence specified by the regular expression), the script posts a window thanking the user and confirming the number. If the number is invalid, the script posts a window telling the user that the phone number isn't valid.

The regular expression looks for zero or one open parenthesis `\(?`, followed by three digits `\d{3}`, followed by zero or one close parenthesis `\)?`, followed by one dash, forward slash, or decimal point and when found, remember the character `([-\/\.])`, followed by three digits `\d{3}`, followed by the remembered match of a dash, forward slash, or decimal point `\1`, followed by four digits `\d{4}`.

The Change event activated when the user presses Enter, sets the value of `RegExp.input`.

```
<HTML>
<SCRIPT LANGUAGE = "JavaScript1.2">

re = /\(?\d{3}\)?([-\/\.])\d{3}\1\d{4}/

function testInfo() {
   OK = re.exec()
   if (!OK)
      window.alert (RegExp.input +
         " isn't a phone number with area code!")
   else
      window.alert ("Thanks, your phone number is " + OK[0])
}

</SCRIPT>

Enter your phone number (with area code) and then press Enter.
<FORM>
```

```
<INPUT TYPE="text" NAME="Phone" onChange="testInfo(this);">
</FORM>

</HTML>
```

Chapter

3

Object Model

JavaScript is based on a simple object-oriented paradigm. An object is a construct with properties that are JavaScript variables or other objects. An object also has functions associated with it that are known as the object's *methods*. In addition to objects that are predefined in the Navigator client and the server, you can define your own objects.

This chapter describes how to use objects, properties, functions, and methods, and how to create your own objects.

Objects and Properties

A JavaScript object has properties associated with it. You access the properties of an object with a simple notation:

```
objectName.propertyName
```

Both the object name and property name are case sensitive. You define a property by assigning it a value. For example, suppose there is an object named `myCar` (for now, just assume the object already exists). You can give it properties named `make`, `model`, and `year` as follows:

```
myCar.make = "Ford"
myCar.model = "Mustang"
myCar.year = 69;
```

An array is an ordered set of values associated with a single variable name. Properties and arrays in JavaScript are intimately related; in fact, they are different interfaces to the same data structure. So, for example, you could access the properties of the `myCar` object as follows:

```
myCar["make"] = "Ford"
myCar["model"] = "Mustang"
myCar["year"] = 67
```

This type of array is known as an *associative array*, because each index element is also associated with a string value. To illustrate how this works, the following function displays the properties of the object when you pass the object and the object's name as arguments to the function:

```
function show_props(obj, obj_name) {
   var result = ""
   for (var i in obj)
      result += obj_name + "." + i + " = " + obj[i] + "\n"
   return result
}
```

So, the function call `show_props(myCar, "myCar")` would return the following:

```
myCar.make = Ford
myCar.model = Mustang
myCar.year = 67
```

Functions

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure—a set of statements that performs a specific task. To use a function, you must first define it; then your script can call it.

Defining Functions

A function definition consists of the `function` keyword, followed by

- The name of the function.
- A list of arguments to the function, enclosed in parentheses and separated by commas.

- The JavaScript statements that define the function, enclosed in curly braces, { }. The statements in a function can include calls to other functions defined in the current application.

In Navigator JavaScript, it is good practice to define all your functions in the HEAD of a page so that when a user loads the page, the functions are loaded first.

For example, here is the definition of a simple function named `pretty_print`:

```
function pretty_print(str) {
   document.write("<HR><P>" + str)
}
```

This function takes a string, `str`, as its argument, adds some HTML tags to it using the concatenation operator (+), and then displays the result to the current document using the `write` method.

In addition to defining functions as described here, you can also define `Function` objects, as described in "Function Object" on page 59.

Using Functions

In a Navigator application, you can use (or *call*) any function defined in the current page. You can also use functions defined by other named windows or frames. In a server-side JavaScript application, you can use any function compiled with the application.

Defining a function does not execute it. You have to call the function for it to do its work. For example, if you defined the example function `pretty_print` in the HEAD of the document, you could call it as follows:

```
<SCRIPT>
pretty_print("This is some text to display")
</SCRIPT>
```

The arguments of a function are not limited to strings and numbers. You can pass whole objects to a function, too. The `show_props` function (defined in "Objects and Properties" on page 41) is an example of a function that takes an object as an argument.

A function can even be recursive, that is, it can call itself. For example, here is a function that computes factorials:

```
function factorial(n) {
   if ((n == 0) || (n == 1))
      return 1
   else {
      result = (n * factorial(n-1) )
   return result
   }
}
```

You could then display the factorials of one through five as follows:

```
for (x = 0; x < 5; x++) {
   document.write("<BR>", x, " factorial is ", factorial(x))
}
```

The results are:

0 factorial is 1
1 factorial is 1
2 factorial is 2
3 factorial is 6
4 factorial is 24
5 factorial is 120

Using the arguments Array

The arguments of a function are maintained in an array. Within a function, you can address the parameters passed to it as follows:

```
functionName.arguments[i]
```

where `functionName` is the name of the function and `i` is the ordinal number of the argument, starting at zero. So, the first argument passed to a function named `myfunc` would be `myfunc.arguments[0]`. The total number of arguments is indicated by the variable `arguments.length`.

Using the `arguments` array, you can call a function with more arguments than it is formally declared to accept using. This is often useful if you don't know in advance how many arguments will be passed to the function. You can use `arguments.length` to determine the number of arguments actually passed to the function, and then treat each argument using the `arguments` array.

For example, consider a function defined to create HTML lists. The only formal argument for the function is a string that is "U" for an unordered (bulleted) list or "O" for an ordered (numbered) list. The function is defined as follows:

```
function list(type) {
   document.write("<" + type + "L>") // begin list
   // iterate through arguments
   for (var i = 1; i < list.arguments.length; i++)
      document.write("<LI>" + list.arguments[i])
      document.write("</" + type + "L>") // end list
}
```

You can pass any number of arguments to this function, and it will then display each argument as an item in the indicated type of list. For example, the following call to the function

```
list("o", "one", 1967, "three", "etc., etc...")
```

results in this output:

1. one
2. 1967
3. three
4. etc., etc...

In JavaScript 1.2, `arguments` includes additional properties, as described in the *JavaScript Reference*.

Creating New Objects

Both client-side and server-side JavaScript have a number of predefined objects. In addition, you can create your own objects. In JavaScript 1.2, you can create objects using literal notation, as described in "Object Literals" on page 12. Alternatively, you can create your own object with these two steps:

1. Define the object type by writing a constructor function.
2. Create an instance of the object with `new`.

To define an object type, create a function for the object type that specifies its name, properties, and methods. For example, suppose you want to create an object type for cars. You want this type of object to be called `car`, and you want it to have properties for make, model, year, and color. To do this, you would write the following function:

```
function car(make, model, year) {
   this.make = make
   this.model = model
   this.year = year
}
```

Notice the use of `this` to assign values to the object's properties based on the values passed to the function.

Now you can create an object called `mycar` as follows:

```
mycar = new car("Eagle", "Talon TSi", 1993)
```

This statement creates `mycar` and assigns it the specified values for its properties. Then the value of `mycar.make` is the string "Eagle", `mycar.year` is the integer 1993, and so on.

You can create any number of `car` objects by calls to `new`. For example,

```
kenscar = new car("Nissan", "300ZX", 1992)
vpgscar = new car("Mazda", "Miata", 1990)
```

An object can have a property that is itself another object. For example, suppose you define an object called `person` as follows:

```
function person(name, age, sex) {
   this.name = name
   this.age = age
   this.sex = sex
}
```

and then instantiate two new `person` objects as follows:

```
rand = new person("Rand McKinnon", 33, "M")
ken = new person("Ken Jones", 39, "M")
```

Then you can rewrite the definition of `car` to include an `owner` property that takes a `person` object, as follows:

```
function car(make, model, year, owner) {
   this.make = make
   this.model = model
   this.year = year
   this.owner = owner
}
```

To instantiate the new objects, you then use the following:

```
car1 = new car("Eagle", "Talon TSi", 1993, rand)
car2 = new car("Nissan", "300ZX", 1992, ken)
```

Notice that instead of passing a literal string or integer value when creating the new objects, the above statements pass the objects `rand` and `ken` as the arguments for the owners. Then if you want to find out the name of the owner of car2, you can access the following property:

```
car2.owner.name
```

Note that you can always add a property to a previously defined object. For example, the statement

```
car1.color = "black"
```

adds a property `color` to car1, and assigns it a value of "black." However, this does not affect any other objects. To add the new property to all objects of the same type, you have to add the property to the definition of the `car` object type.

Indexing Object Properties

In Navigator 2.0, you can refer to an object's properties by their property name or by their ordinal index. In Navigator 3.0 or later, however, if you initially define a property by its name, you must always refer to it by its name, and if you initially define a property by an index, you must always refer to it by its index.

This applies when you create an object and its properties with a constructor function, as in the above example of the `Car` object type, and when you define individual properties explicitly (for example, `myCar.color = "red"`). So if you define object properties initially with an index, such as `myCar[5] = "25 mpg"`, you can subsequently refer to the property as `myCar[5]`.

The exception to this rule is objects reflected from HTML, such as the `forms` array. You can always refer objects in these arrays by either their ordinal number (based on where they appear in the document) or their name (if defined). For example, if the second `<FORM>` tag in a document has a `NAME` attribute of "myForm", you can refer to the form as `document.forms[1]` or `document.forms["myForm"]` or `document.myForm`.

Defining Properties for an Object Type

You can add a property to a previously defined object type by using the `prototype` property. This defines a property that is shared by all objects of the specified type, rather than by just one instance of the object. The following code adds a `color` property to all objects of type `car`, and then assigns a value to the `color` property of the object `car1`. For more information, see the `prototype` property of the `Function` object in the *JavaScript Reference*.

```
Car.prototype.color=null
car1.color="black"
birthday.description="The day you were born"
```

Defining Methods

A *method* is a function associated with an object. You define a method the same way you define a standard function. Then you use the following syntax to associate the function with an existing object:

```
object.methodname = function_name
```

where `object` is an existing object, `methodname` is the name you are assigning to the method, and `function_name` is the name of the function.

You can then call the method in the context of the object as follows:

```
object.methodname(params);
```

You can define methods for an object type by including a method definition in the object constructor function. For example, you could define a function that would format and display the properties of the previously-defined `car` objects; for example,

```
function displayCar() {
   var result = "A Beautiful " + this.year + " " + this.make
      + " " + this.model
   pretty_print(result)
}
```

where `pretty_print` is the function (defined in "Functions" on page 42) to display a horizontal rule and a string. Notice the use of `this` to refer to the object to which the method belongs.

You can make this function a method of `car` by adding the statement

```
this.displayCar = displayCar;
```

to the object definition. So, the full definition of `car` would now look like

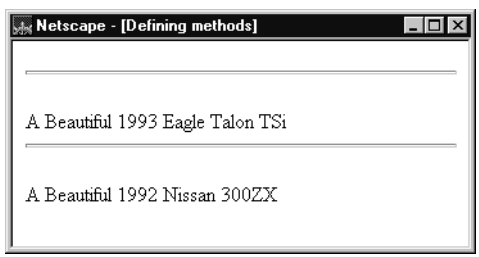
```
function car(make, model, year, owner) {
   this.make = make
   this.model = model
   this.year = year
   this.owner = owner
   this.displayCar = displayCar
}
```

Then you can call the `displayCar` method for each of the objects as follows:

```
car1.displayCar()
car2.displayCar()
```

This produces the output shown in Figure 3.1

Figure 3.1 Displaying method output



Using this for Object References

JavaScript has a special keyword, `this`, that you can use within a method to refer to the current object. For example, suppose you have a function called `validate` that validates an object's `value` property, given the object and the high and low values:

```
function validate(obj, lowval, hival) {
   if ((obj.value < lowval) || (obj.value > hival))
      alert("Invalid Value!")
}
```

Then, you could call `validate` in each form element's `onChange` event handler, using `this` to pass it the form element, as in the following example:

```
<INPUT TYPE="text" NAME="age" SIZE=3
   onChange="validate(this, 18, 99)">
```

In general, `this` refers to the calling object in a method.

When combined with the `form` property, `this` can refer to the current object's parent form. In the following example, the form `myForm` contains a `Text` object and a button. When the user clicks the button, the value of the `Text` object is set to the form's name. The button's `onClick` event handler uses `this.form` to refer to the parent form, `myForm`.

```
<FORM NAME="myForm">
Form name:<INPUT TYPE="text" NAME="text1" VALUE="Beluga">
<P>
<INPUT NAME="button1" TYPE="button" VALUE="Show Form Name"
   onClick="this.form.text1.value=this.form.name">
</FORM>
```

Object Deletion

In JavaScript for Navigator 2.0, you cannot remove objects—they exist until you leave the page containing the object. In JavaScript for Navigator 3.0, you can remove an object by setting its object reference to null (if that is the last reference to the object). JavaScript finalizes the object immediately, as part of the assignment expression.

Chapter

4

Predefined Core Objects and Functions

Several objects are predefined in core JavaScript and can be used in either client-side or server-side scripts. A handful of predefined functions can also be used in both client and server scripts.

Objects

The predefined core objects are `Array`, `Boolean`, `Date`, `Function`, `Math`, `Number`, `RegExp`, and `String`.

Array Object

JavaScript does not have an explicit array data type. However, you can use the predefined `Array` object and its methods to work with arrays in your applications. The `Array` object has methods for manipulating arrays in various ways, such as joining, reversing, sorting them. It has a property for determining the array length and other properties for use with regular expressions.

An *array* is an ordered set of values that you refer to with a name and an index. For example, you could have an array called `emp` that contains employees' names indexed by their employee number. So `emp[1]` would be employee number one, `emp[2]` employee number two, and so on.

To create an `Array` object:

```
1. arrayObjectName = new Array([element0, element1, ..., elementN])
2. arrayObjectName = new Array([arrayLength])
```

`arrayObjectName` is either the name of a new object or a property of an existing object. When using `Array` properties and methods, `arrayObjectName` is either the name of an existing `Array` object or a property of an existing object.

`element0, element1, ..., elementN` is a list of values for the array's elements. When this form is specified, the array is initialized with the specified values as its elements, and the array's `length` property is set to the number of arguments.

In Navigator 2.0 and Navigator 3.0, `arrayLength` is the initial length of the array. In Navigator 4.0, if the `<SCRIPT>` tag does not specify `"JavaScript1.2"` as the value of the `LANGUAGE` attribute, this is still true. However, if it does specify `"JavaScript1.2",` then `Array(arrayLength)` creates an array of length one with `arrayLength` as its only element. That is, it no longer considers a single integer argument as a special case.

In Navigator 4.0, in addition to creating arrays using the `Array` function constructor, you can also create them using literal notation, as described in "Array Literals" on page 12.

The `Array` object has the following methods:

- `concat` joins two arrays and returns a new array.
- `join` joins all elements of an array into a string.
- `pop` removes the last element from an array and returns that element.
- `push` adds one or more elements to the end of an array and returns that last element added.
- `reverse` transposes the elements of an array: the first array element becomes the last and the last becomes the first.
- `shift` removes the first element from an array and returns that element
- `slice` extracts a section of an array and returns a new array.
- `splice` adds and/or removes elements from an array.

- `sort` sorts the elements of an array.
- `unshift` adds one or more elements to the front of an array and returns the new length of the array.

For example, suppose you define the following array:

```
myArray = new Array("Wind","Rain","Fire")
```

`myArray.join()` returns "Wind,Rain,Fire"; `myArray.reverse` transposes the array so that `myArray[0]` is "Fire", `myArray[1]` is "Rain", and `myArray[2]` is "Wind". `myArray.sort` sorts the array so that `myArray[0]` is "Fire", `myArray[1]` is "Rain", and `myArray[2]` is "Wind".

Populating an Array

You can populate an array by assigning values to its elements. For example,

```
emp[1] = "Casey Jones"
emp[2] = "Phil Lesh"
emp[3] = "August West"
```

You can also populate an array when you create it:

```
myArray = new Array("Hello", myVar, 3.14159)
```

The following code creates a two-dimensional array and displays the results.

```
a = new Array(4)
for (i=0; i < 4; i++) {
   a[i] = new Array(4)
   for (j=0; j < 4; j++) {
      a[i][j] = "["+i+","+j+"]"
   }
}
for (i=0; i < 4; i++) {
   str = "Row "+i+":"
   for (j=0; j < 4; j++) {
      str += a[i][j]
   }
   document.write(str,"<p>")
}
```

This example displays the following results:

```
Multidimensional array test
Row 0:[0,0][0,1][0,2][0,3]
Row 1:[1,0][1,1][1,2][1,3]
Row 2:[2,0][2,1][2,2][2,3]
Row 3:[3,0][3,1][3,2][3,3]
```

Referring to Array Elements

You can refer to an array's elements by using the element's value or ordinal number. For example, suppose you define the following array:

```
myArray = new Array("Wind","Rain","Fire")
```

You can then refer to the first element of the array as `myArray[0]` or `myArray["Wind"]`.

Arrays and Regular Expressions

When an array is the result of a match between a regular expression and a string, the array returns properties and elements that provide information about the match. An array is the return value of `regexp.exec`, `string.match`, and `string.replace`. For information on using arrays with regular expressions, see "Regular Expressions" on page 26.

Boolean Object

Use the predefined `Boolean` object when you need to convert a non-boolean value to a boolean value. You can use the `Boolean` object any place JavaScript expects a primitive boolean value. JavaScript returns the primitive value of the `Boolean` object by automatically invoking the `valueOf` method.

To create a `Boolean` object:

```
booleanObjectName = new Boolean(value)
```

`booleanObjectName` is either the name of a new object or a property of an existing object. When using `Boolean` properties, `booleanObjectName` is either the name of an existing `Boolean` object or a property of an existing object.

`value` is the initial value of the `Boolean` object. The `value` is converted to a boolean value, if necessary. If `value` is omitted or is 0, `null`, `false`, or the empty string "", the object has an initial value of `false`. All other values, including the string "false" create an object with an initial value of `true`.

The following examples create `Boolean` objects:

```
bfalse = new Boolean(false)
btrue = new Boolean(true)
```

Date Object

JavaScript does not have a date data type. However, you can use the `Date` object and its methods to work with dates and times in your applications. The `Date` object has a large number of methods for setting, getting, and manipulating dates. It does not have any properties.

JavaScript handles dates similarly to Java. The two languages have many of the same date methods, and both languages store dates as the number of milliseconds since January 1, 1970, 00:00:00.

Note Currently, you cannot work with dates prior to January 1, 1970.

To create a `Date` object:

```
dateObjectName = new Date([parameters])
```

where `dateObjectName` is the name of the `Date` object being created; it can be a new object or a property of an existing object.

The `parameters` in the preceding syntax can be any of the following:

- Nothing: creates today's date and time. For example, `today = new Date()`.
- A string representing a date in the following form: "Month day, year hours:minutes:seconds." For example, `Xmas95 = new Date("December 25, 1995 13:30:00")`. If you omit hours, minutes, or seconds, the value will be set to zero.
- A set of integer values for year, month, and day. For example, `Xmas95 = new Date(95,11,25)`. A set of values for year, month, day, hour, minute, and seconds. For example, `Xmas95 = new Date(95,11,25,9,30,0)`.

Methods of the Date Object

The `Date` object methods for handling dates and times fall into these broad categories:

- "set" methods, for setting date and time values in `Date` objects.
- "get" methods, for getting date and time values from `Date` objects.
- "to" methods, for returning string values from `Date` objects.
- parse and UTC methods, for parsing `Date` strings.

With the "get" and "set" methods you can get and set seconds, minutes, hours, day of the month, day of the week, months, and years separately. There is a `getDay` method that returns the day of the week, but no corresponding `setDay` method, because the day of the week is set automatically. These methods use integers to represent these values as follows:

- Seconds and minutes: 0 to 59
- Hours: 0 to 23
- Day: 0 to 6 (day of the week)
- Date: 1 to 31 (day of the month)
- Months: 0 (January) to 11 (December)
- Year: years since 1900

For example, suppose you define the following date:

```
Xmas95 = new Date("December 25, 1995")
```

Then `Xmas95.getMonth()` returns 11, and `Xmas95.getYear()` returns 95.

The `getTime` and `setTime` methods are useful for comparing dates. The `getTime` method returns the number of milliseconds since January 1, 1970, 00:00:00 for a `Date` object.

For example, the following code displays the number of days left in the current year:

```
today = new Date()
endYear = new Date("December 31, 1990") // Set day and month
endYear.setYear(today.getYear()) // Set year to this year
msPerDay = 24 * 60 * 60 * 1000 // Number of milliseconds per day
```

```
daysLeft = (endYear.getTime() - today.getTime()) / msPerDay
daysLeft = Math.round(daysLeft)
document.write("Number of days left in the year: " + daysLeft)
```

This example creates a `Date` object named `today` that contains today's date. It then creates a `Date` object named `endYear` and sets the year to the current year. Then, using the number of milliseconds per day, it computes the number of days between today and `endYear`, using `getTime` and rounding to a whole number of days.

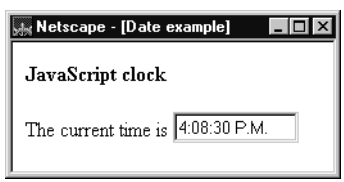
The `parse` method is useful for assigning values from date strings to existing `Date` objects. For example, the following code uses `parse` and `setTime` to assign a date value to the `IPOdate` object:

```
IPOdate = new Date()
IPOdate.setTime(Date.parse("Aug 9, 1995"))
```

Using the Date Object: an Example

The following example shows a simple application of `Date`: it displays a continuously-updated digital clock in an HTML text field. This is possible because you can dynamically change the contents of a text field with JavaScript (in contrast to ordinary text, which you cannot update without reloading the document). The display in Navigator is shown in Figure 4.1.

Figure 4.1 Digital clock example



The `<BODY>` of the document is:

```
<BODY onLoad="JSClock()">
<FORM NAME="clockForm">
The current time is <INPUT TYPE="text" NAME="digits" SIZE=12 VALUE="">
</FORM>
</BODY>
```

The <BODY> tag includes an onLoad event handler. When the page loads, the event handler calls the function JSClock, defined in the <HEAD>. A form called clockForm includes a single text field named digits, whose value is initially an empty string.

The <HEAD> of the document defines JSClock as follows:

```
<HEAD>
<SCRIPT language="JavaScript1.2">
<!--
function JSClock() {
   var time = new Date()
   var hour = time.getHours()
   var minute = time.getMinutes()
   var second = time.getSeconds()
   var temp = "" + ((hour > 12) ? hour - 12 : hour)
   temp += ((minute < 10) ? ":0" : ":") + minute
   temp += ((second < 10) ? ":0" : ":") + second
   temp += (hour >= 12) ? " P.M." : " A.M."
   document.clockForm.digits.value = temp
   id = setTimeout("JSClock()",1000)
}
//-->
</SCRIPT>
</HEAD>
```

The JSClock function first creates a new Date object called time; since no arguments are given, time is created with the current date and time. Then calls to the getHours, getMinutes, and getSeconds methods assign the value of the current hour, minute and seconds to hour, minute, and second.

The next four statements build a string value based on the time. The first statement creates a variable temp, assigning it a value using a conditional expression; if hour is greater than 12, (hour - 13), otherwise simply hour.

The next statement appends a minute value to temp. If the value of minute is less than 10, the conditional expression adds a string with a preceding zero; otherwise it adds a string with a demarcating colon. Then a statement appends a seconds value to temp in the same way.

Finally, a conditional expression appends "PM" to temp if hour is 12 or greater; otherwise, it appends "AM" to temp.

The next statement assigns the value of temp to the text field:

```
document.aform.digits.value = temp
```

This displays the time string in the document.

The final statement in the function is a recursive call to `JSClock`:

```
id = setTimeout("JSClock()", 1000)
```

The predefined JavaScript `setTimeout` function specifies a time delay to evaluate an expression, in this case a call to `JSClock`. The second argument indicates a delay of 1,000 milliseconds (one second). This updates the display of `time` in the form at one-second intervals.

Note that the function returns a value (assigned to `id`), used only as an identifier (which can be used by the `clearTimeout` method to cancel the evaluation).

Function Object

The predefined `Function` object specifies a string of JavaScript code to be compiled as a function.

To create a `Function` object:

```
functionObjectName = new Function ([arg1, arg2, ... argn], functionBody)
```

`functionObjectName` is the name of a variable or a property of an existing object. It can also be an object followed by a lowercase event handler name, such as `window.onerror`.

`arg1, arg2, ... argn` are arguments to be used by the function as formal argument names. Each must be a string that corresponds to a valid JavaScript identifier; for example "x" or "theForm".

`functionBody` is a string specifying the JavaScript code to be compiled as the function body.

`Function` objects are evaluated each time they are used. This is less efficient than declaring a function and calling it within your code, because declared functions are compiled.

In addition to defining functions as described here, you can also use the `function` statement, as described in the *JavaScript Reference*.

The following code assigns a function to the variable `setBGColor`. This function sets the current document's background color.

```
var setBGColor = new Function("document.bgColor='antiquewhite'")
```

To call the `Function` object, you can specify the variable name as if it were a function. The following code executes the function specified by the `setBGColor` variable:

```
var colorChoice="antiquewhite"
if (colorChoice=="antiquewhite") {setBGColor()}
```

You can assign the function to an event handler in either of the following ways:

```
1. document.form1.colorButton.onclick=setBGColor
2. <INPUT NAME="colorButton" TYPE="button"
      VALUE="Change background color"
      onClick="setBGColor()">
```

Creating the variable `setBGColor` shown above is similar to declaring the following function:

```
function setBGColor() {
   document.bgColor='antiquewhite'
}
```

Assigning a function to a variable is similar to declaring a function, but there are differences:

- When you assign a function to a variable using `var setBGColor = new Function("...")`, `setBGColor` is a variable for which the current value is a reference to the function created with `new Function()`.
- When you create a function using `function setBGColor() {...}`, `setBGColor` is not a variable, it is the name of a function.

In Navigator 4.0, you can nest a function within a function. (That is, JavaScript now supports lambda expressions and lexical closures.) The nested function can use the arguments and variables of the outer function. The outer function cannot use the arguments and variables of the nested function.

Math Object

The predefined `Math` object has properties and methods for mathematical constants and functions. For example, the `Math` object's `PI` property has the value of pi (3.141...), which you would use in an application as

```
Math.PI
```

Similarly, standard mathematical functions are methods of `Math`. These include trigonometric, logarithmic, exponential, and other functions. For example, if you want to use the trigonometric function sine, you would write

```
Math.sin(1.56)
```

Note that all trigonometric methods of `Math` take arguments in radians.

Table 4.1 summarizes the `Math` object's methods.

Table 4.1 Methods of `Math`

Method	Description
`abs`	Absolute value
`sin, cos, tan`	Standard trigonometric functions; argument in radians
`acos, asin, atan`	Inverse trigonometric functions; return values in radians
`exp, log`	Exponential and natural logarithm, base `e`
`ceil`	Returns least integer greater than or equal to argument
`floor`	Returns greatest integer less than or equal to argument
`min, max`	Returns greater or lesser (respectively) of two arguments
`pow`	Exponential; first argument is base, second is exponent
`round`	Rounds argument to nearest integer
`sqrt`	Square root

Unlike many other objects, you never create a `Math` object of your own. You always use the predefined `Math` object.

It is often convenient to use the `with` statement when a section of code uses several math constants and methods, so you don't have to type "Math" repeatedly. For example,

```
with (Math) {
   a = PI * r*r
   y = r*sin(theta)
   x = r*cos(theta)
}
```

Number Object

The `Number` object has properties for numerical constants, such as maximum value, not-a-number, and infinity. You cannot change the values of these properties and you use them as follows:

```
biggestNum = Number.MAX_VALUE
smallestNum = Number.MIN_VALUE
infiniteNum = Number.POSITIVE_INFINITY
negInfiniteNum = Number.NEGATIVE_INFINITY
notANum = Number.NaN
```

You always refer to a property of the predefined `Number` object as shown above, and not as a property of a `Number` object you create yourself.

Table 4.2 summarizes the `Number` object's properties.

Table 4.2 Properties of `Number`

Method	Description
`MAX_VALUE`	The largest representable number
`MIN_VALUE`	The smallest representable number
`NaN`	Special "not a number" value
`NEGATIVE_INFINITY`	Special infinite value; returned on overflow
`POSITIVE_INFINITY`	Special negative infinite value; returned on overflow

RegExp Object

The `RegExp` object lets you work with regular expressions. It is described in "Regular Expressions" on page 26.

String Object

JavaScript does not have a string data type. However, you can use the `String` object and its methods to work with strings in your applications. The `String` object has a large number of methods for manipulating strings. It has one property for determining the string's length.

To create a `String` object:

```
stringObjectName = new String(string)
```

`stringObjectName` is the name of a new `String` object.

`string` is any string.

For example, the following statement creates a `String` object called `mystring`:

```
mystring = new String ("Hello, World!")
```

String literals are also `String` objects; for example, the literal "Howdy" is a `String` object.

A `String` object has one property, `length`, that indicates the number of characters in the string. So, using the previous example, the expression

```
x = mystring.length
```

assigns a value of 13 to `x`, because "Hello, World!" has 13 characters.

A `String` object has two types of methods: those that return a variation on the string itself, such as `substring` and `toUpperCase`, and those that return an HTML-formatted version of the string, such as `bold` and `link`.

For example, using the previous example, both `mystring.toUpperCase()` and `"hello, world!".toUpperCase()` return the string "HELLO, WORLD!".

The `substring` method takes two arguments and returns a subset of the string between the two arguments. Using the previous example, `mystring.substring(4, 9)` returns the string "o, Wo." For more information, see `String.substring` in the *JavaScript Reference*.

The `String` object also has a number of methods for automatic HTML formatting, such as `bold` to create boldface text and `link` to create a hyperlink. For example, you could create a hyperlink to a hypothetical URL with the `link` method as follows:

```
mystring.link("http://www.helloworld.com")
```

Table 4.3 summarizes the methods of `String` objects:

Table 4.3 Methods of String

Method	Description
`anchor`	Creates HTML named anchor
`big, blink, bold, fixed, italics, small, strike, sub, sup`	Creates HTML formatted string
`charAt, charCodeAt`	Returns the character or character code at the specified position in string
`indexOf, lastIndexOf`	Returns the position of specified substring in the string or last position of specified substring, respectively
`link`	Creates HTML hyperlink
`concat`	Combines the text of two strings and returns a new string
`fromCharCode`	Constructs a string from the specified sequence of ISO-Latin-1 codeset values
`split`	Splits a `String` object into an array of strings by separating the string into substrings
`slice`	Extracts a section of an string and returns a new string.
`substring, substr`	Returns the specified subset of the string, either by specifying the start and end indexes or the start index and a length
`match, replace, search`	Used to work with regular expressions
`toLowerCase, toUpperCase`	Returns the string in all lowercase or all uppercase, respectively

Functions

JavaScript has several "top-level" functions predefined in the language `eval`, `isNan`, `Number`, `String`, `parseInt, parseFloat`, `escape`, `unescape`, `taint`, and `untaint`. For more information on all of these functions, see the *JavaScript Reference*.

eval Function

The `eval` function evaluates a string of JavaScript code without reference to a particular object. The syntax of `eval` is:

```
eval(expr)
```

where `expr` is a string to be evaluated.

If the string represents an expression, `eval` evaluates the expression. If the argument represents one or more JavaScript statements, `eval` performs the statements. Do not call `eval` to evaluate an arithmetic expression; JavaScript evaluates arithmetic expressions automatically.

isNaN Function

The `isNaN` function evaluates an argument to determine if it is "NaN" (not a number). The syntax of `isNaN` is:

```
isNaN(testValue)
```

where `testValue` is the value you want to evaluate.

On platforms that support NaN, the `parseFloat` and `parseInt` functions return "NaN" when they evaluate a value that is not a number. `isNaN` returns true if passed "NaN," and false otherwise.

The following code evaluates `floatValue` to determine if it is a number and then calls a procedure accordingly:

```
floatValue=parseFloat(toFloat)

if (isNaN(floatValue)) {
   notFloat()
} else {
   isFloat()
}
```

parseInt and parseFloat Functions

The two "parse" functions, `parseInt` and `parseFloat`, return a numeric value when given a string as an argument. For detailed descriptions and examples, see the *JavaScript Reference*. The syntax of `parseFloat` is

```
parseFloat(str)
```

where `parseFloat` parses its argument, the string `str`, and attempts to return a floating-point number. If it encounters a character other than a sign (+ or -), a numeral (0-9), a decimal point, or an exponent, then it returns the value up to that point and ignores that character and all succeeding characters. If the first character cannot be converted to a number, it returns "NaN" (not a number).

The syntax of `parseInt` is

```
parseInt(str [, radix])
```

`parseInt` parses its first argument, the string `str`, and attempts to return an integer of the specified `radix` (base), indicated by the second, optional argument, `radix`. For example, a radix of ten indicates to convert to a decimal number, eight octal, sixteen hexadecimal, and so on. For radixes above ten, the letters of the alphabet indicate numerals greater than nine. For example, for hexadecimal numbers (base 16), A through F are used.

If `parseInt` encounters a character that is not a numeral in the specified radix, it ignores it and all succeeding characters and returns the integer value parsed up to that point. If the first character cannot be converted to a number in the specified radix, it returns "NaN." The `parseInt` function truncates numbers to integer values.

Number and String Functions

The Number and String functions let you convert an object to a number or a string. The syntax of these functions is:

```
Number(objRef)
String(objRef)
```

where `objRef` is an object reference.

The following example converts the `Date` object to a readable string.

```
<SCRIPT>
D = new Date (430054663215);
document.write (String(D) +" <BR>");
</SCRIPT>
```

This prints "Thu Aug 18 04:37:43 Pacific Daylight Time 1983."

escape and unescape Functions

The `escape` and `unescape` functions let you encode and decode strings. The `escape` function returns the hexadecimal encoding of an argument in the ISO Latin character set. The `unescape` function returns the ASCII string for the specified value.

The syntax of these functions is:

```
escape(string)
unescape(string)
```

These functions are used primarily with server-side JavaScript to encode and decode name/value pairs in URLs.

taint and untaint Functions

The `taint` and `untaint` functions are used for adding and removing data tainting in Navigator 3.0. Data tainting prevents other scripts from passing information that should be secure and private, such as directory structures or user session history. JavaScript cannot pass tainted values on to any server without the end user's permission.

Chapter

5

Overview of JavaScript Statements

JavaScript supports a compact set of statements that you can use to incorporate a great deal of interactivity in Web pages. This chapter provides an overview of these statements.

The statements fall into the following categories:

- Conditional statements: `if...else`, and `switch`
- Loop Statements: `for`, `while`, `do while`, `labeled`, `break`, and `continue` (`labeled` is not itself a looping statement, but is frequently used with these statements)
- Object Manipulation Statements and Operators: `for...in`, `new`, `this`, and `with`
- Comments: single-line (//) and multiline (/*...*/)

The following sections provide a brief overview of each statement. See the *JavaScript Reference* for details.

Conditional Statement

A conditional statement is a set of commands that executes if a specified condition is true. JavaScript supports two conditional statements: `if...else` and `switch`.

if...else Statement

Use the `if` statement to perform certain statements if a logical condition is true; use the optional `else` clause to perform other statements if the condition is false. An `if` statement looks as follows:

```
if (condition) {
   statements1
}
[else {
   statements2
} ]
```

The condition can be any JavaScript expression that evaluates to true or false. The statements to be executed can be any JavaScript statements, including further nested `if` statements. If you want to use more than one statement after an `if` or `else` statement, you must enclose the statements in curly braces, {}.

Example. In the following example, the function `checkData` returns true if the number of characters in a `Text` object is three; otherwise, it displays an alert and returns false.

```
function checkData () {
   if (document.form1.threeChar.value.length == 3) {
      return true
   } else {
      alert("Enter exactly three characters. " +
      document.form1.threeChar.value + " is not valid.")
      return false
   }
}
```

switch Statement

A `switch` statement allows a program to evaluate an expression and attempt to match the expression's value to a case label. If a match is found, the program executes the associated statement. A `switch` statement looks as follows:

```
switch (expression){
   case label :
      statement;
      break;
   case label :
      statement;
      break;
   ...
   default : statement;
}
```

The program first looks for a label matching the value of expression and then executes the associated statement. If no matching label is found, the program looks for the optional default statement, and if found, executes the associated statement. If no default statement is found, the program continues execution at the statement following the end of `switch`.

The optional `break` statement associated with each case label ensures that the program breaks out of `switch` once the matched statement is executed and continues execution at the statement following switch. If `break` is omitted, the program continues execution at the next statement in the `switch` statement.

Example. In the following example, if `expr` evaluates to "Bananas", the program matches the value with case "Bananas" and executes the associated statement. When `break` is encountered, the program terminates `switch` and executes the statement following `switch`. If `break` were omitted, the statement for case "Cherries" would also be executed.

```
switch (expr) {
   case "Oranges" :
      document.write("Oranges are $0.59 a pound.<BR>");
      break;
   case "Apples" :
      document.write("Apples are $0.32 a pound.<BR>");
      break;
   case "Bananas" :
      document.write("Bananas are $0.48 a pound.<BR>");
      break;
   case "Cherries" :
      document.write("Cherries are $3.00 a pound.<BR>");
      break;
```

```
   default :
      document.write("Sorry, we are out of " + i + ".<BR>");
}
document.write("Is there anything else you'd like?<BR>");
```

Loop Statements

A loop is a set of commands that executes repeatedly until a specified condition is met. JavaScript supports two loop statements: `for` and `while`. In addition, you can use the `break` and `continue` statements within loop statements.

Another statement, `for...in`, executes statements repeatedly but is used for object manipulation. See "Object Manipulation Statements and Operators" on page 78.

for Statement

A `for` loop repeats until a specified condition evaluates to false. The JavaScript `for` loop is similar to the Java and C `for` loop. A `for` statement looks as follows:

```
for ([initial-expression]; [condition]; [increment-expression]) {
   statements
}
```

When a `for` loop executes, the following occurs:

1. The initializing expression `initial-expression`, if any, is executed. This expression usually initializes one or more loop counters, but the syntax allows an expression of any degree of complexity.
2. The `condition` expression is evaluated. If the value of `condition` is true, the loop statements execute. If the value of `condition` is false, the `for` loop terminates.
3. The update expression `increment-expression` executes.
4. The `statements` execute, and control returns to step 2.

Example. The following function contains a `for` statement that counts the number of selected options in a scrolling list (a `Select` object that allows multiple selections). The `for` statement declares the variable `i` and initializes it to zero. It checks that `i` is less than the number of options in the `Select` object, performs the succeeding `if` statement, and increments `i` by one after each pass through the loop.

```
<SCRIPT>
function howMany(selectObject) {
   var numberSelected=0
   for (var i=0; i < selectObject.options.length; i++) {
      if (selectObject.options[i].selected==true)
         numberSelected++
   }
   return numberSelected
}
</SCRIPT>
<FORM NAME="selectForm">
<P><B>Choose some music types, then click the button below:</B>
<BR><SELECT NAME="musicTypes" MULTIPLE>
<OPTION SELECTED> R&B
<OPTION> Jazz
<OPTION> Blues
<OPTION> New Age
<OPTION> Classical
<OPTION> Opera
</SELECT>
<P><INPUT TYPE="button" VALUE="How many are selected?"
onClick="alert ('Number of options selected: ' +
howMany(document.selectForm.musicTypes))">
</FORM>
```

do...while Statement

The `do...while` statement repeats until a specified condition evaluates to false. A `do...while` statement looks as follows:

```
do {
   statement
} while (condition)
```

`statement` executes once before the condition is checked. If `condition` returns `true`, the statement executes again. At the end of every execution, the condition is checked. When the condition returns `false`, execution stops and control passes to the statement following `do...while`.

Example. In the following example, the `do` loop iterates at least once and reiterates until i is no longer less than 5.

```
do {
   i+=1;
   document.write(i);
} while (i<5);
```

while Statement

A `while` statement executes its statements as long as a specified condition evaluates to true. A `while` statement looks as follows:

```
while (condition) {
   statements
}
```

If the condition becomes false, the statements within the loop stop executing and control passes to the statement following the loop.

The condition test occurs before the statements in the loop are executed. If the condition returns true, the statements are executed and the condition is tested again. If the condition returns false, execution stops and control is passed to the statement following `while`.

Example 1. The following `while` loop iterates as long as `n` is less than three:

```
n = 0
x = 0
while( n < 3 ) {
   n ++
   x += n
}
```

With each iteration, the loop increments `n` and adds that value to `x`. Therefore, `x` and `n` take on the following values:

- After the first pass: `n` = 1 and `x` = 1
- After the second pass: `n` = 2 and `x` = 3
- After the third pass: `n` = 3 and `x` = 6

After completing the third pass, the condition `n < 3` is no longer true, so the loop terminates.

Example 2: infinite loop. Make sure the condition in a loop eventually becomes false; otherwise, the loop will never terminate. The statements in the following `while` loop execute forever because the condition never becomes false:

```
while (true) {
   alert("Hello, world") }
```

labeled Statement

A `labeled` statement provides an identifier that can be used with `break` or `continue` to indicate where the program should continue execution. A labeled statement looks as follows:

```
label :
   statement
```

In a `labeled` statement, `break` or `continue` must be followed with a label, and the label must be the identifier of the labeled statement containing`break` or `continue`. The statements in a labeled statement can be of any type.

Example. A statement labeled `checkiandj` contains a statement labeled `checkj`. If break is encountered, the program breaks out of the `checkj` statement and continues with the remainder of the `checkiandj` statement. If break had a label of `checkiandj`, the program would break out of the `checkiandj` statement and continue at the statement following `checkiandj`.

```
checkiandj :
   if (4==i) {
      document.write("You've entered " + i + ".<BR>");
      checkj :
         if (2==j) {
            document.write("You've entered " + j + ".<BR>");
            break checkj;
            document.write("The sum is " + (i+j) + ".<BR>");
         }
      document.write(i + "-" + j + "=" + (i-j) + ".<BR>");
   }
```

break Statement

The `break` statement can be used in a `while`, `for`, and `labeled` statement.

- In a `while` or `for` statement, `break` terminates the current loop and transfers control to the statement following the terminated loop.
- In a `labeled` statement, `break` is followed by a label that identifies a `labeled` statement. This type of `break` allows the program to terminate the identified statement and transfer control to the statement following the terminated statement. `break` must be in a statement identified by the label used by `break`.

In a `while` or `for` statement, `break` looks as follows:

```
break
```

In a `labeled` statement, `break` looks as follows:

```
break label
```

Example 1. The following function has a `break` statement that terminates the `while` loop when `i` is three, and then returns the value 3 * `x`.

```
function testBreak(x) {
   var i = 0
   while (i < 6) {
      if (i == 3)
         break
      i++
   }
   return i*x
}
```

Example 2. A statement labeled `checkiandj` contains a statement labeled `checkj`. If `break` is encountered, the program terminates the `checkj` statement and continues with the remainder of the `checkiandj` statement. If `break` had a label of `checkiandj`, the program would terminate the `checkiandj` statement and continue at the statement following `checkiandj`.

```
checkiandj :
   if (4==i) {
      document.write("You've entered " + i + ".<BR>");
      checkj :
         if (2==j) {
            document.write("You've entered " + j + ".<BR>");
            break checkj;
            document.write("The sum is " + (i+j) + ".<BR>");
         }
      document.write(i + "-" + j + "=" + (i-j) + ".<BR>");
   }
```

continue Statement

The `continue` statement can be used in a `while`, `for`, and `labeled` statement.

- In a `while` or `for` statement, `continue` terminates the current loop and continues execution of the loop with the next iteration. In contrast to the `break` statement, `continue` does not terminate the execution of the loop entirely. In a `while` loop, it jumps back to the `condition`. In a `for` loop, it jumps to the `increment-expression`.
- In a `labeled` statement, `continue` is followed by a label that identifies a `labeled` statement. This type of `continue` allows the program to terminate execution of a `labeled` statement and continue to the identified `labeled` statement. `continue` must be in a looping statement identified by the label used by `continue`.

In a `while` or `for` statement, `continue` looks as follows:

```
continue
```

In a `labeled` statement, `continue` looks as follows:

```
continue label
```

Example 1. The following example shows a `while` loop with a `continue` statement that executes when the value of `i` is three. Thus, `n` takes on the values one, three, seven, and twelve.

```
i = 0
n = 0
while (i < 5) {
   i++
   if (i == 3)
      continue
   n += i
}
```

Example 2. A statement labeled `checkiandj` contains a statement labeled `checkj`. If `continue` is encountered, the program terminates the current iteration of `checkj` and begins the next iteration. Each time `continue` is encountered, `checkj` reiterates until its condition returns `false`. When `false` is returned, the remainder of the `checkiandj` statement is completed, and `checkiandj` reiterates until its condition returns `false`. When `false` is returned, the program continues at the statement following `checkiandj`.

If `continue` had a label of `checkiandj`, the program would continue at the top of the `checkiandj` statement.

```
checkiandj :
   while (i<4) {
      document.write(i + "<BR>");
      i+=1;
      checkj :
         while (j>4) {
            document.write(j + "<BR>");
            j-=1;
            if ((j%2)==0);
               continue checkj;
            document.write(j + " is odd.<BR>");
         }
      document.write("i = " + i + "<br>");
      document.write("j = " + j + "<br>");
   }
```

Object Manipulation Statements and Operators

JavaScript has several ways of manipulating objects: `new` operator, `this` keyword, `for...in` statement, and `with` statement.

new Operator

You can use the `new` operator to create an instance of a user-defined object type or of one of the predefined object types `Array`, `Boolean`, `Date`, `Function`, `Image`, `Number`, `Object`, `Option`, `RegExp`, or `String`. Use `new` as follows:

```
objectName = new objectType ( param1 [,param2] ...[,paramN] )
```

The following example creates an `Array` object with 25 elements, then assigns values to the first three elements:

```
musicTypes = new Array(25)
musicTypes[0] = "R&B"
musicTypes[1] = "Blues"
musicTypes[2] = "Jazz"
```

The following examples create several `Date` objects:

```
today = new Date()
birthday = new Date("December 17, 1995 03:24:00")
birthday = new Date(95,12,17)
```

The following example creates a user-define object type `car`, with properties for make, model, and year. The example then creates an object called `mycar` and assigns values to its properties. The value of `mycar.make` is the string "Eagle", `mycar.year` is the integer 1993, and so on.

```
function car(make, model, year) {
   this.make = make
   this.model = model
   this.year = year
}

mycar = new car("Eagle", "Talon TSi", 1993)
```

For more information on `new`, see the *JavaScript Reference*.

In Navigator 4.0, you can also create new objects using literal notation, as described in "Object Literals" on page 12.

this Keyword

Use the `this` keyword to refer to the current object. In general, `this` refers to the calling object in a method. Use `this` as follows:

```
this[.propertyName]
```

Example 1. Suppose a function called `validate` validates an object's `value` property, given the object and the high and low values:

```
function validate(obj, lowval, hival) {
   if ((obj.value < lowval) || (obj.value > hival))
      alert("Invalid Value!")
}
```

You could call `validate` in each form element's `onChange` event handler, using `this` to pass it the form element, as in the following example:

```
<B>Enter a number between 18 and 99:</B>
<INPUT TYPE = "text" NAME = "age" SIZE = 3
   onChange="validate(this, 18, 99)">
```

Example 2. When combined with the `form` property, `this` can refer to the current object's parent form. In the following example, the form `myForm` contains a `Text` object and a button. When the user clicks the button, the value of the `Text` object is set to the form's name. The button's `onClick` event handler uses `this.form` to refer to the parent form, `myForm`.

```
<FORM NAME="myForm">
Form name:<INPUT TYPE="text" NAME="text1" VALUE="Beluga">
<P>
<INPUT NAME="button1" TYPE="button" VALUE="Show Form Name"
   onClick="this.form.text1.value=this.form.name">
</FORM>
```

for...in Statement

The `for...in` statement iterates a specified variable over all the properties of an object. For each distinct property, JavaScript executes the specified statements. A `for...in` statement looks as follows:

```
for (variable in object) {
   statements }
```

Example. The following function takes as its argument an object and the object's name. It then iterates over all the object's properties and returns a string that lists the property names and their values.

```
function dump_props(obj, obj_name) {
   var result = ""
   for (var i in obj) {
      result += obj_name + "." + i + " = " + obj[i] + "<BR>"
   }
   result += "<HR>"
   return result
}
```

For an object `car` with properties `make` and `model`, `result` would be:

car.make = Ford
car.model = Mustang

with Statement

The `with` statement establishes the default object for a set of statements. Within the set of statements, any property references that do not specify an object are assumed to be for the default object. A `with` statement looks as follows:

```
with (object){
   statements
}
```

Example. The following `with` statement specifies that the `Math` object is the default object. The statements following the `with` statement refer to the `PI` property and the `cos` and `sin` methods, without specifying an object. JavaScript assumes the `Math` object for these references.

```
var a, x, y
var r=10
with (Math) {
   a = PI * r * r
   x = r * cos(PI)
   y = r * sin(PI/2)
}
```

Comments

Comments are author notations that explain what a script does. Comments are ignored by the interpreter. JavaScript supports Java-style comments:

- Comments on a single line are preceded by a double-slash (//).
- Comments that span multiple lines are preceded by /* and followed by */:

Example. The following example shows two comments:

```
// This is a single-line comment.

/* This is a multiple-line comment. It can be of any length, and
you can put whatever you want here. */
```

Appendix

A

Reserved Words

This appendix lists the reserved words in JavaScript.

The reserved words in this list cannot be used as JavaScript variables, functions, methods, or object names. Some of these words are keywords used in JavaScript; others are reserved for future use.

abstract	else	int	switch
boolean	extends	interface	synchronized
break	false	long	this
byte	final	native	throw
case	finally	new	throws
catch	float	null	transient
char	for	package	true
class	function	private	try
const	goto	protected	typeof
continue	if	public	var
default	implements	return	void
delete	import	short	while
do	in	static	with
double	instanceof	super	

Index

Note: This index has not yet been updated.

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W