JavaScript

What Is JavaScript?

JavaScript is Netscape's cross-platform, object-oriented scripting language. Core JavaScript contains a core set of objects, such as Array, Date, and Math, and a core set of language elements such as operators, control structures, and statements. Core JavaScript can be extended for a variety of purposes by supplementing it with additional objects; for example:

- Client-side JavaScript extends the core language by supplying objects to control a browser (Navigator or another web browser) and its Document Object Model (DOM). For example, client-side extensions allow an application to place elements on an HTML form and respond to user events such as mouse clicks, form input, and page navigation.
- Server-side JavaScript extends the core language by supplying objects relevant to running JavaScript on a server. For example, server-side extensions allow an application to communicate with a relational database, provide continuity of information from one invocation to another of the application, or perform file manipulations on a server.

Differences between client-side JavaScript and server-side JavaScript

client-side JavaScript	server -side JavaScript
Client-side JavaScript statements embedded in an HTML page can respond to user events such as mouse clicks, form input, and page navigation. For example, you can write a JavaScript function to verify that users enter valid information into a form requesting a telephone number or zip code. Without any network transmission, the embedded JavaScript on the HTML page can check the entered data and display a dialog box if the user enters invalid data.	On the server, you also embed JavaScript in HTML pages. The server-side statements can connect to relational databases from different vendors, share information across users of an application, access the file system on the server, or communicate with other applications through LiveConnect and Java. HTML pages with server-side JavaScript can also include client-side JavaScript. In contrast to pure client-side JavaScript pages, HTML pages that use server-side JavaScript are compiled into bytecode executable files. These application executables are run by a web server that contains the JavaScript runtime engine. For this reason, creating JavaScript applications is a two-stage process.
Client applications run in a browser, such as Netscape Navigator	server applications run on a server, such as Netscape Enterprise Server.

Debugging JavaScript

JavaScript allows you to write complex computer programs. As with all languages, you may make mistakes while writing your scripts. The Netscape JavaScript Debugger allows you to debug your scripts.

Embedding JavaScript in HTML

You can embed JavaScript in an HTML document as statements and functions within a <SCRIPT> tag, by specifying a file as the JavaScript source, by specifying a JavaScript expression as the value of an HTML attribute, or as event handlers within certain other HTML tags (primarily form elements).
This chapter contains the following sections:

- Using the SCRIPT Tag
- Specifying a File of JavaScript Code
- Using JavaScript Expressions as HTML Attribute Values
- Using Quotation Marks
- Specifying Alternate Content with the NOSCRIPT Tag

Using the SCRIPT Tag

The <SCRIPT> tag is an extension to HTML that can enclose any number of JavaScript statements as shown here:

```
<SCRIPT>

   JavaScript statements...

</SCRIPT>
```

A document can have multiple <SCRIPT> tags, and each can enclose any number of JavaScript statements.

What can a JavaScript Do?

- **JavaScript gives HTML designers a programming tool -** HTML authors are normally not programmers, but JavaScript is a scripting language with a very simple syntax! Almost anyone can put small "snippets" of code into their HTML pages
- **JavaScript can put dynamic text into an HTML page -** A JavaScript statement like this: document.write("<h1>" + name + "</h1>") can write a variable text into an HTML page
- **JavaScript can react to events -** A JavaScript can be set to execute when something happens, like when a page has finished loading or when a user clicks on an HTML element
- **JavaScript can read and write HTML elements -** A JavaScript can read and change the content of an HTML element
- **JavaScript can be used to validate data -** A JavaScript can be used to validate form data before it is submitted to a server. This saves the server from extra processing

- **JavaScript can be used to detect the visitor's browser** - A JavaScript can be used to detect the visitor's browser, and - depending on the browser - load another page specifically designed for that browser
- **JavaScript can be used to create cookies** - A JavaScript can be used to store and retrieve information on the visitor's computer

The Real Name is ECMAScript

JavaScript's official name is "ECMAScript". The standard is developed and maintained by the ECMA organisation.

ECMA-262 is the official JavaScript standard. The standard is based on JavaScript (Netscape) and JScript (Microsoft).

The language was invented by Brendan Eich at Netscape (with Navigator 2.0), and has appeared in all Netscape and Microsoft browsers since 1996.

The development of ECMA-262 started in 1996, and the first edition of was adopted by the ECMA General Assembly in June 1997.

The standard was approved as an international ISO (ISO/IEC 16262) standard in 1998.

The development of the standard is still in progress.

Values, 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; null is also a primitive value. Because JavaScript is case sensitive, null is not the same as Null, NULL, or any other variant.
- undefined, a top-level property whose value is undefined; undefined is also a primitive value.

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 JavaScript. 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 dynamically 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 dynamically typed, this assignment does not cause an error message.
In expressions involving numeric and string values with the + operator, JavaScript converts numeric values to strings. For example, consider the following statements:

```
x = "The answer is " + 42 // returns "The answer is 42"
y = 42 + " is the answer" // returns "42 is the answer"
```

In statements involving other operators, JavaScript does not convert numeric values to strings. For example:

```
"37" - 7 // returns 30

"37" + 7 // returns 377
```

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.

Declaring Variables

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

Evaluating Variables

A variable or array element that has not been assigned a value has the value undefined. The result of evaluating an unassigned variable depends on how it was declared:

- If the unassigned variable was declared without var, the evaluation results in a runtime error.
- If the unassigned variable was declared with var, the evaluation results in the undefined value, or NaN in numeric contexts.

The following code demonstrates evaluating unassigned variables.

```
Function f1() {
return y-2;
}
f1() //Causes runtime error
function f2() {
return var y - 2;
}
f2() //returns NaN
```

You can use undefined to determine whether a variable has a value. In the following code, the variable input is not assigned a value, and the if statement evaluates to true.

```
var input;
if(input === undefined){
   doThis();
} else {
   doThat();
}
```

The undefined value behaves as false when used as a Boolean value. For example, the following code executes the function myFunction because the array element is not defined:

```
myArray=new Array()
if (!myArray[0])
   myFunction()
```

When you evaluate a null variable, the null value behaves as 0 in numeric contexts and as false in Boolean contexts. For example:

```
var n = null

n * 32 //returns 0
```

Variable Scope

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. This section describes the following types of literals:

- Array Literals
- Boolean Literals

- Floating-Point Literals
- Integers
- Object Literals
- String Literals

Array Literals

An array literal is a list of zero or more expressions, each of which represents an array element, enclosed in square brackets ([]). When you create an array using an array literal, it is initialized with the specified values as its elements, and its length is set to the number of arguments specified.
The following example creates the coffees array with three elements and a length of three:
coffees = ["Nescafe Classic", "Sunrise", "Bru"]
An array literal is a type of object initializer. If an array is created using a literal in a top-level script, JavaScript interprets the array 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. Array literals are also Array objects.

Extra Commas in Array Literals

You do not have to specify all elements in an array literal. If you put two commas in a row, the array is created with spaces for the unspecified elements. The following example creates the fish array:
aninal = ["Lion", , "tiger"]
This array has two elements with values and one empty element (animal[0] is "Lion", animal[1] is undefined, and animal[2] is "Angel"):
If you include a trailing comma at the end of the list of elements, the comma is ignored. In the following example, the length of the array is three. There is no myList[3]. All other commas in the list indicate a new element.
myList = ['home', , 'school',];
In the following example, the length of the array is four, and myList[0] is missing.
myList = [, 'home', , 'school'];
In the following example, the length of the array is four, and myList[3] is missing. Only the last comma is ignored. This trailing comma is optional.
myList = ['home', , 'school', ,];

Boolean Literals

The Boolean type has two literal values: true and false.
Do not confuse the primitive Boolean values true and false with the true and false values of the Boolean object. The Boolean object is a wrapper around the primitive Boolean data type.

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

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

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.

Object Literals

An object literal is a list of zero or more pairs of property names and associated values of an object, enclosed in curly braces ({}). You should not use an object literal at the beginning of a statement. This will lead to an error.
The following is an example of an object literal. The first element of the car object defines a property, myCar; the second element, the getCar property, invokes a function (Cars("honda")); the third element, the special property, uses an existing variable (Sales).

```
var Sales = "Toyota";
 function CarTypes(name) {
   if(name == "Honda")
      return name;
   else
      return "Sorry, we don't sell " + name + ".";
}
car = {myCar: "Saturn", getCar: CarTypes("Honda"), special:
Sales}
document.write(car.myCar); // Saturn
document.write(car.getCar); // Honda
document.write(car.special); // Toyota
```

Additionally, you can use an index for the object, the index property,or nest an object inside another. The following example uses these options. These features, however, may not be supported by other ECMA-compliant browsers.

```
car = {manyCars: {a: "Saab", b: "Jeep"}, 7: "Mazda"}
document.write(car.manyCars.b); // Jeep
document.write(car[7]); // Mazda
```

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"

You can call any of the methods of the String object on a string literal value--JavaScript automatically converts the string literal to a temporary String object, calls the method, then discards the temporary String object. You can also use the String.length property with a string literal.
You should use string literals unless you specifically need to use a String object. See "String Object" on page 118 for details on String objects.

Using Special Characters in Strings

In addition to ordinary characters, you can also include special characters in strings, as shown in the following example.
"one line \n another line"
The following table lists the special characters that you can use in JavaScript strings.
JavaScript special characters

Character	**Meaning**
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Tab
\'	Apostrophe or single quote
\"	Double quote
\\	Backslash character (\)
\XXX	The character with the Latin-1 encoding specified by up to three octal digits XXX between 0 and 377. For example, \251 is the octal sequence for the copyright symbol.

\xXX	The character with the Latin-1 encoding specified by the two hexadecimal digits XX between 00 and FF. For example, \xA9 is the hexadecimal sequence for the copyright symbol.
\uXXXX	The Unicode character specified by the four hexadecimal digits XXXX. For example, \u00A9 is the Unicode sequence for the copyright symbol. See "Unicode Escape Sequences" on page 44.

Escaping Characters

For characters not listed in the above table, 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 \”JavaScript\” on the way home”;
document.write(quote)
The result of this would be
He read "javaScript” on the way home.

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"

Expressions and Operators

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

Operators

JavaScript has the following types of operators. This section describes the operators and contains information about operator precedence.

- Assignment Operators
- Comparison Operators

- Arithmetic Operators
- Bitwise Operators
- Logical Operators
- String Operators
- Special Operators

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 the following table.
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. The operands can be numerical or string values. Strings are compared based on standard lexicographical ordering, using Unicode values. The following table describes the comparison operators.

Comparison operators

Operator	**Description**	**Examples returning true**[1]
Equal (==)	Returns true if the operands are equal. If the two operands are not of the same type, JavaScript attempts to convert the operands to an appropriate type for the comparison.	3 == var1 "3" == var1 3 == '3'
Not equal (!=)	Returns true if the operands are not equal. If the two operands are not of the same type, JavaScript attempts to convert the operands to an appropriate type for the comparison.	var1 != 4 var2 != "3"
Strict equal (===)	Returns true if the operands are equal and of the same type.	3 === var1
Strict not equal (!==)	Returns true if the operands are not equal and/or not of the same type.	var1 !== "3" 3 !== '3'
Greater than (>)	Returns true if the left operand is greater than the right operand.	var2 > var1
Greater than or equal (>=)	Returns true if the left operand is greater than or equal to the right operand.	var2 >= var1 var1 >= 3
Less than (<)	Returns true if the left operand is less than the right operand.	var1 < var2
Less than or equal (<=)	Returns true if the left operand is less than or equal to the right operand.	var1 <= var2 var2 <= 5

[1] These examples assume that var1 has been assigned the value 3 and var2 has 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 most other programming languages, except the / operator returns a floating-point division in JavaScript, not a truncated division as it does in languages such as C or Java. For example:

```
1/2 //returns 0.5 in JavaScript
1/2 //returns 0 in Java
```

In addition, JavaScript provides the arithmetic operators listed in the following table.

Arithmetic Operators

Operator	**Description**	**Example**
% (Modulus)	Binary operator. Returns the integer remainder of dividing the two 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.
-- (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 32 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.

Bitwise operators

Operator	**Usage**	**Description**
Bitwise AND	a & b	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bitwise OR	a | b	Returns a one in each bit position for which the corresponding bits of either or both operands are ones.
Bitwise XOR	a ^ b	Returns a one in each bit position for which the corresponding bits of either but not both operands are ones.
Bitwise NOT	~ a	Inverts 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 the following table.

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 are typically used with Boolean (logical) values; when they are, they return a Boolean value. However, the && and || operators actually return the value of one of the specified operands, so if these operators are used with non-Boolean values, they may return a non-Boolean value. The logical operators are described in the following table.

Logical operators

Operator	Usage	Description
&&	expr1 && expr2	(Logical AND) Returns expr1 if it can be converted to false; otherwise, returns expr2. Thus, when used with Boolean values, && returns true if both operands are true; otherwise, returns false.
||	expr1 || expr2	(Logical OR) Returns expr1 if it can be converted to true; otherwise, returns expr2. Thus, when used with Boolean values, || returns true if either operand is true; if both are false, returns false.
!	!expr	(Logical NOT) Returns false if its single operand can be converted to true; otherwise, returns true.

Examples of expressions that can be converted to false are those that evaluate to null, 0, the empty string (""), or undefined.
The following code shows examples of the && (logical AND) operator.

```
a1=true && true       // t && t returns true
a2=true && false      // t && f returns false
a3=false && true      // f && t returns false
a4=false && (3 == 4)  // f && f returns false
a5="Cat" && "Dog"     // t && t returns Dog
a6=false && "Cat"     // f && t returns false
a7="Cat" && false     // t && f returns false
The following code shows examples of the || (logical OR)
operator.
o1=true || true       // t || t returns true
o2=false || true      // f || t returns true
o3=true || false      // t || f returns true
o4=false || (3 == 4)  // f || f returns false
o5="Cat" || "Dog"     // t || t returns Cat
o6=false || "Cat"     // f || t returns Cat
o7="Cat" || false     // t || f returns Cat
The following code shows examples of the ! (logical NOT)
operator.
n1=!true              // !t returns false
n2=!false             // !f returns true
n3=!"Cat"             // !t returns 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

JavaScript provides the following special operators:

- conditional operator
- comma operator
- delete
- new
- this
- typeof
- void

conditional operator

The conditional operator is the only JavaScript operator that takes three 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=9; i <= 9; i++, j--)
   document.writeln("a["+i+","+j+"]= " + a[i,j])
```

delete

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

```
delete objectName
delete objectName.property
delete objectName[index]
delete property // legal only within a with statement
```

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 fourth form is legal only within a with statement, to delete a property from an object.
You can use the delete operator to delete variables declared implicitly but not those declared with the var statement.
If the delete operator succeeds, it sets the property or element to undefined. The delete operator returns true if the operation is possible; it returns false if the operation is not possible.

```
x=42
var y= 43
myobj=new Number()
myobj.h=4     // create property h
delete x     // returns true (can delete if declared implicitly)
delete y    // returns false (cannot delete if declared with var)

delete Math.PI // returns false (cannot delete predefined
properties)

delete myobj.h // returns true (can delete user-defined
properties)

delete myobj   // returns true (can delete user-defined object)
```

Deleting array elements

When you delete an array element, the array length is not affected. For example, if you delete a[3], a[4] is still a[4] and a[3] is undefined.
When the delete operator removes an array element, that element is no longer in the array. In the following example, trees[3] is removed with delete.

```
trees=new Array("redwood","bay","cedar","oak","maple")
delete trees[3]
if (3 in trees) {
   // this does not get executed
}
```

If you want an array element to exist but have an undefined value, use the undefined keyword instead of the delete operator. In the following example, trees[3] is assigned the value undefined, but the array element still exists:

```
trees=new Array("redwood","bay","cedar","oak","maple")
trees[3]=undefined
if (3 in trees) {
   // this gets executed
}
```

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] )
```

this

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

typeof

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

```
typeof operand
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:

```
void(expression)
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.
The following table describes the precedence of operators, from lowest to highest.

Operator precedence

Operator type	**Individual operators**
comma	,
assignment	= += -= *= /= %= <<= >>= >>>= &= ^= |=
conditional	?:
logical-or	||
logical-and	&&

bitwise-or	|
bitwise-xor	^
bitwise-and	&
equality	== !=
relational	< <= > >=
bitwise shift	<< >> >>>
addition/subtraction	+ -
multiply/divide	* / %
negation/increment	! ~ - + ++ -- typeof void delete
call	()
create instance	New
member	. []

Predefined Functions

JavaScript has several top-level predefined functions:

- eval
- isFinite
- isNaN
- parseInt and parseFloat
- Number and String
- escape and unescape

The following sections introduce these functions. See for detailed information on all of these functions.

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.

isFinite Function

The isFinite function evaluates an argument to determine whether it is a finite number. The syntax of isFinite is:

```
isFinite(number)
```

where number is the number to evaluate.
If the argument is NaN, positive infinity or negative infinity, this method returns false, otherwise it returns true.
The following code checks client input to determine whether it is a finite number.

```
if(isFinite(ClientInput) == true)
{
   /* take specific steps */
}
```

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.
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.
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).

parseInt
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 the string 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.
```
D = new Date (430054663215)
// The following returns
// "Thu Aug 18 04:37:43 GMT-0700 (Pacific Daylight Time) 1983"
x = String(D)
```

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 hexadecimal encoding 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.

Predefined Core Objects

This section describes the predefined objects in core JavaScript: Array, Boolean, Date, Function, Math, Number, RegExp, and String. The predefined client-side objects are

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, and 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.

Creating an Array

To create an Array object:

```
arrayObjectName = new Array(element0, element1, ..., elementN)
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.
arrayLength is the initial length of the array. The following code creates an array of five elements:

```
billingMethod = new Array(5)
```

Array literals are also Array objects; for example, the following literal is an Array object..

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

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)
```

Referring to Array Elements

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

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

You then refer to the first element of the array as myArray[0] and the second element of the array as myArray[1].
The index of the elements begins with zero (0), but the length of array (for example, myArray.length) reflects the number of elements in the array.

Array Methods

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".

Two-Dimensional Arrays

The following code creates a two-dimensional array.
```
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+"]"
   }
}
```
The following code displays the array:
```
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:
```
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]
```

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 Chapter 4, "Regular Expressions."

Boolean Object

The Boolean object is a wrapper around the primitive Boolean data type. Use the following syntax to create a Boolean object:
booleanObjectName = new Boolean(value)
Do not confuse the primitive Boolean values true and false with the true and false values of the Boolean object. Any object whose value is not undefined or null, including a Boolean object whose value is false, evaluates to true when passed to a conditional statement. See "if...else Statement" on page 80 for more information.

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.
The Date object range is -100,000,000 days to 100,000,000 days relative to 01 January, 1970 UTC.
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(1995,11,25). A set of values for year, month, day, hour, minute, and seconds. For example, Xmas95 = new Date(1995,11,25,9,30,0).

JavaScript 1.2 and earlier versions. The Date object behaves as follows:

- Dates prior to 1970 are not allowed.
- JavaScript depends on platform-specific date facilities and behavior; the behavior of the Date object varies from platform to platform.

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 (Sunday) to 6 (Saturday)
- 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.getFullYear() 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(1995,11,31,23,59,59,999) // Set day and month
endYear.setFullYear(today.getFullYear()) // 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) //returns days left in the year
```

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

In the following example, the function JSClock() returns the time in the format of a digital clock.

```
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."
   return temp
}
```

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.

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. See the Client-Side JavaScript Reference for more information.
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'
}
```

You can nest a function within a function. The nested (inner) function is private to its containing (outer) function:

- The inner function can be accessed only from statements in the outer function.
- The inner function can use the arguments and variables of the outer function. The outer function cannot use the arguments and variables of the inner 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.
The following table summarizes the Math object's methods.
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.
The following table summarizes the Number object's properties.

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 Chapter 4, "Regular Expressions."

String Object

The String object is a wrapper around the string primitive data type. Do not confuse a string literal with the String object. For example, the following code creates the string literal s1 and also the String object s2:

```
s1 = "foo" //creates a string literal value
s2 = new String("foo") //creates a String object
```

You can call any of the methods of the String object on a string literal value--JavaScript automatically converts the string literal to a temporary String object, calls the method, then discards the temporary String object. You can also use the String.length property with a string literal.
You should use string literals unless you specifically need to use a String object, because String objects can have counterintuitive behavior. For example:

```
s1 = "2 + 2" //creates a string literal value
s2 = new String("2 + 2")//creates a String object
eval(s1) //returns the number 4
eval(s2) //returns the string "2 + 2"
```

A String object has one property, length, that indicates the number of characters in the string. For example, the following code assigns x the value 13, because "Hello, World!" has 13 characters:

```
myString = "Hello, World!"
x = mystring.length
```

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."

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")
The following table summarizes the methods of String objects.

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

Statements

JavaScript supports a compact set of statements that you can use to incorporate a great deal of interactivity in Web pages. JavaScript supports various statements.

Conditional Statements: if...else and switch

- Loop Statements: for, while, do while, label, break, and continue (label is not itself a looping statement, but is frequently used with these statements)
- Object Manipulation Statements: for...in and with
- Comments

Any expression is also a statement. Use the semicolon (;) character to separate statements in JavaScript code.

Conditional Statements

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, {}.

Do not confuse the primitive Boolean values true and false with the true and false values of the Boolean object. Any object whose value is not undefined or null, including a Boolean object whose value is false, evaluates to true when passed to a conditional statement. For example:

```
var b = new Boolean(false);
if (b) // this condition evaluates to true
```

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 the for, do while, while, and label loop statements (label is not itself a looping statement, but is frequently used with these statements). 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.

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 ([initialExpression]; [condition]; [incrementExpression]) {
   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 statements execute.
4. The update expression incrementExpression executes, 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") }
```

label Statement

A label provides a statement with an identifier that lets you refer to it elsewhere in your program. For example, you can use a label to identify a loop, and then use the break or continue statements to indicate whether a program should interrupt the loop or continue its execution.

The syntax of the label statement looks like the following:

```
Label:
   statement
```

The value of label may be any JavaScript identifier that is not a reserved word. The statement that you identify with a label may be any type.

Example. In this example, the label markLoop identifies a while loop.

```
markLoop:
while (theMark == true)
   doSomething();
}
```

break Statement

Use the break statement to terminate a loop, switch, or label statement.

- When you use break with a while, do-while, for, or switch statement, break terminates the innermost enclosing loop or switch immediately and transfers control to the following statement.
- When you use break within an enclosing label statement, it terminates the statement and transfers control to the following statement. If you specify a label when you issue the break, the break statement terminates the specified statement.

The syntax of the break statement looks like the following:

```
 break
  break [label]
```

The first form of the syntax terminates the innermost enclosing loop, switch, or label; the second form of the syntax terminates the specified enclosing label statement.

Example. The following example iterates through the elements in an array until it finds the index of an element whose value is theValue:

```
for (i = 0; i < a.length; i++) {
   if (a[i] = theValue);
      break;
}
```

continue Statement

The continue statement can be used to restart a while, do-while, for, or label 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 label statement, continue is followed by a label that identifies a label statement. This type of continue restarts a label statement or continues execution of a labelled loop with the next iteration. continue must be in a looping statement identified by the label used by continue.

The syntax of the continue statement looks like the following:

```
continue
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

JavaScript uses the for...in and with statements to manipulate objects.

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. JavaScript looks up any unqualified names within the set of statements to determine if the names are

properties of the default object. If an unqualified name matches a property, then the property is used in the statement; otherwise, a local or global variable is used.
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. */
```

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.
This chapter contains the following sections:

- Defining Functions
- Calling Functions
- Using the arguments Array
- Predefined Functions

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.

Generally, you should define all your functions in the HEAD of a page so that when a user loads the page, the functions are loaded first. Otherwise, the user might perform an action while the page is still loading that triggers an event handler and calls an undefined function, leading to an error.
For example, the following code defines a simple function named square:

```
function square(number) {
   return number * number;
}
```

The function square takes one argument, called number. The function consists of one statement that indicates to return the argument of the function multiplied by itself. The return statement specifies the value returned by the function.
return number * number
All parameters are passed to functions by value; the value is passed to the function, but if the function changes the value of the parameter, this change is not reflected globally or in the calling function. However, if you pass an object as a parameter to a function and the function changes the object's properties, that change is visible outside the function, as shown in the following example:

```
function myFunc(theObject) {
   theObject.make="Toyota"
}
mycar = {make:"Honda", model:"Accord", year:1998}
x=mycar.make      // returns Honda
myFunc(mycar)     // pass object mycar to the function
y=mycar.make      // returns Toyota (prop was changed by the
function)
```

In addition to defining functions as described here, you can also define Function objects, as described A method is a function associated with an object.

Calling 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.
Defining a function does not execute it. Defining the function simply names the function and specifies what to do when the function is called. Calling the function actually performs the specified actions with the indicated parameters. For example, if you define the function square, you could call it as follows.
square(5)
The preceding statement calls the function with an argument of five. The function executes its statements and returns the value twenty-five.
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 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 compute the factorials of one through five as follows:

```
a=factorial(1) // returns 1
b=factorial(2) // returns 2
c=factorial(3) // returns 6
d=factorial(4) // returns 24
e=factorial(5) // returns 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:
arguments[i]

functionName.arguments[i]
where i is the ordinal number of the argument, starting at zero. So, the first argument passed to a function would be arguments[0]. The total number of arguments is indicated by arguments.length.
Using the arguments array, you can call a function with more arguments than it is formally declared to accept. 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 that concatenates several strings. The only formal argument for the function is a string that specifies the characters that separate the items to concatenate. The function is defined as follows:

```
function myConcat(separator) {
   result="" // initialize list
   // iterate through arguments
   for (var i=1; i<arguments.length; i++) {
      result += arguments[i] + separator
   }
   return result
}
```

You can pass any number of arguments to this function, and it creates a list using each argument as an item in the list.

```
// returns "red, orange, blue, "

myConcat(", ","red","orange","blue")
```

```
// returns "elephant; giraffe; lion; cheetah;"
myConcat("; ","elephant","giraffe","lion", "cheetah")
// returns "sage. basil. oregano. pepper. parsley. "
myConcat(". ","sage","basil","oregano", "pepper", "parsley")
```

Regular Expressions

Regular expressions are patterns used to match character combinations in strings. In JavaScript, regular expressions are also objects. These patterns are used with the exec and test methods of RegExp, and with the match, replace, search, and split methods of String. This chapter describes JavaScript regular expressions.
JavaScript 1.1 and earlier. Regular expressions are not available in JavaScript 1.1 and earlier.

Creating a Regular Expression

You construct a regular expression in one of two ways:

- Using an object initializer, as follows:

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

  Object initializers provide compilation of the regular expression when the script is evaluated. When the regular expression will remain constant, use this for better performance. Calling the constructor function of the RegExp object, as follows:

  ```
  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 73.

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'.
The following table provides a complete list and description of the special characters that can be used in regular expressions.

Special characters in regular expressions.

Character	**Meaning**
\	Either of the following: • 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. • 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 grunted".
+	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."
?	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 m are positive integers. Matches at least n and at most m 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-d]. They match the 'b' in "brisket" and the 'c' in "ache".
[^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 or a newline character. (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."
\cX	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.
\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.
\ooctal \xhex	Where \ooctal is an octal escape value or \xhex 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 73.
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 "Chapter 3 and 4", because that string does not have a period after the '3'.

Working with Regular Expressions

Regular expressions are used with the RegExp methods test and exec and with the String methods match, replace, search, and split.

Methods that use regular expressions

Method	Description
exec	A RegExp method that executes a search for a match in a string. It returns an array of information.
test	A RegExp 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="JavaScript">
```

```
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="JavaScript">

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 the following table.

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..)	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 a regular expression created with an object initializer without assigning it to a variable. If you do, however, every occurrence is a new regular expression. For this reason, if you use this 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="JavaScript">
```

```
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="JavaScript">

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 regular expression created with an object initializer, 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="JavaScript">

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="JavaScript">

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="JavaScript">

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="JavaScript">

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

The following examples show some uses of regular expressions.

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="JavaScript">
// 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 informing the user that the phone number is not 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 = "JavaScript">
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>
```

Working with Objects

JavaScript is designed on a simple object-based 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.
This chapter contains the following sections:

- Objects and Properties
- Creating New Objects
- Predefined Core 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 = 1969;
```

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"] = 1967
```

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 = 1967
```

Creating New Objects

JavaScript has a number of predefined objects. In addition, you can create your own objects. In JavaScript 1.2, you can create an object using an object initializer. Alternatively, you can first create a constructor function and then instantiate an object using that function and the new operator.

Using Object Initializers

In addition to creating objects using a constructor function, you can create objects using an object initializer. Using object initializers is sometimes referred to as creating objects with literal notation. "Object initializer" is consistent with the terminology used by C++.
The syntax for an object using an object initializer is:
objectName = {property1:value1, property2:value2,..., propertyN:valueN}
where objectName is the name of the new object, each propertyI is an identifier (either a name, a number, or a string literal), and each valueI is an expression whose value is assigned to the propertyI. The objectName and assignment is optional. If you do not need to refer to this object elsewhere, you do not need to assign it to a variable.
If an object is created with an object initializer in a top-level script, JavaScript interprets the object each time it evaluates the expression containing the object literal. In addition, an initializer used in a function is created each time the function is called.
The following statement creates an object and assigns it to the variable x if and only if the expression cond is true.
if (cond) x = {hi:"there"}
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}}
You can also use object initializers to create arrays. See "Array Literals" on page 37.
JavaScript 1.1 and earlier versions. You cannot use object initializers. You can create objects only using their constructor functions or using a function supplied by some other object for that purpose. See "Using a Constructor Function" on page 102.

Using a Constructor Function

Alternatively, you can create an 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 JavaScript 1.0, you can refer to an object's properties by their property name or by their ordinal index. In JavaScript 1.1 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 to 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.

```
Car.prototype.color=null

car1.color="black"
```

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)

}
```

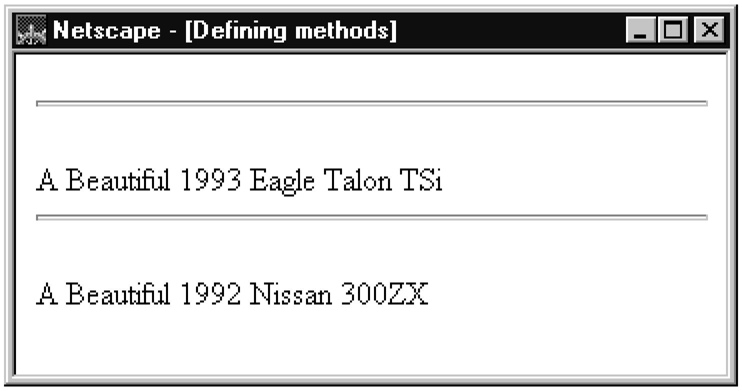
where pretty_print is function to display a horizontal rule and a string. Notice the use of this to refer to the object to which the method belongs.
You can make this function a method of car by adding the statement
this.displayCar = displayCar;
to the object definition. So, the full definition of car 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 the following figure.

Displaying method output



A Beautiful 1993 Eagle Talon TSi

A Beautiful 1992 Nissan 300ZX

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>
```

Deleting Objects

You can remove an object by using the delete operator. The following code shows how to remove an object.

myobj=new Number()

delete myobj // removes the object and returns true

Specifying the JavaScript Version

Each version of Navigator supports a different version of JavaScript. To ensure that users of various versions of Navigator avoid problems when viewing pages that use JavaScript, use the LANGUAGE attribute of the <SCRIPT> tag to specify the version of JavaScript with which a script complies. For example, to use JavaScript 1.2 syntax, you specify the following:

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

Using the LANGUAGE tag attribute, you can write scripts compliant with earlier versions of Navigator. You can write different scripts for the different versions of the browser. If the specific browser does not support the specified JavaScript version, the code is ignored. If you do not specify a LANGUAGE attribute, the default behavior depends on the Navigator version.

The following table lists the <SCRIPT> tags supported by different Netscape versions.

JavaScript and Navigator versions

Navigator version	**Default JavaScript version**	**<SCRIPT> tags supported**
Navigator earlier than 2.0	JavaScript not supported	None
Navigator 2.0	JavaScript 1.0	<SCRIPT LANGUAGE="JavaScript">
Navigator 3.0	JavaScript 1.1	<SCRIPT LANGUAGE="JavaScript1.1"> and all earlier versions
Navigator 4.0-4.05	JavaScript 1.2	<SCRIPT LANGUAGE="JavaScript1.2"> and all earlier versions
Navigator 4.06-4.5	JavaScript 1.3	<SCRIPT LANGUAGE="JavaScript1.3"> and all earlier versions

Navigator ignores code within <SCRIPT> tags that specify an unsupported version. For example, Navigator 3.0 does not support JavaScript 1.2, so if a user runs a JavaScript 1.2 script in Navigator 3.0, the script is ignored.

Example 1. This example shows how to define functions three times, once for JavaScript 1.0, once using JavaScript 1.1 features, and a third time using JavaScript 1.2 features.

```
<SCRIPT LANGUAGE="JavaScript">

// Define 1.0-compatible functions such as doClick() here

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

// Redefine those functions using 1.1 features

// Also define 1.1-only functions

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

// Redefine those functions using 1.2 features

// Also define 1.2-only functions

</SCRIPT>
<FORM ...>

<INPUT TYPE="button" onClick="doClick(this)" ...>

...

</FORM>
```

Example 2. This example shows how to use two separate versions of a JavaScript document, one for JavaScript 1.1 and one for JavaScript 1.2. The default document that loads is for JavaScript 1.1. If the user is running Navigator 4.0, the replace method replaces the page.

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

// Replace this page in session history with the 1.2 version

location.replace("js1.2/mypage.html");

</SCRIPT>
```

Example 3. This example shows how to test the navigator.userAgent property to determine which version of Navigator 4.0 is running. The code then conditionally executes 1.1 and 1.2 features.

```
<SCRIPT LANGUAGE="JavaScript">

if (navigator.userAgent.indexOf("4.0") != -1)

   jsVersion = "1.2";

else if (navigator.userAgent.indexOf("3.0") != -1)

   jsVersion = "1.1";

else

   jsVersion = "1.0";

</SCRIPT>
```

Hiding Scripts Within Comment Tags

Only Navigator versions 2.0 and later recognize JavaScript. To ensure that other browsers ignore JavaScript code, place the entire script within HTML comment tags, and precede the ending comment tag with a double-slash (//) that indicates a JavaScript single-line comment:

```
<SCRIPT>

<!-- Begin to hide script contents from old browsers.

JavaScript statements...

// End the hiding here. -->

</SCRIPT>
```

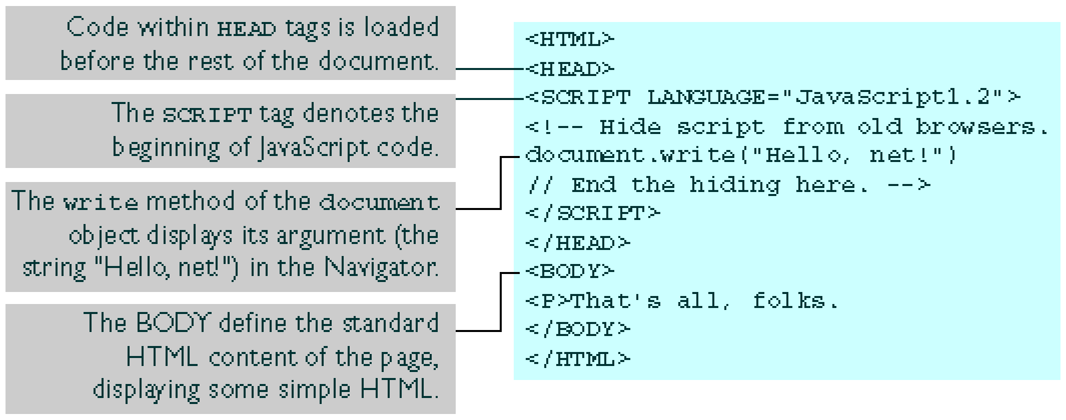
Since browsers typically ignore unknown tags, non-JavaScript-capable browsers will ignore the beginning and ending SCRIPT tags. All the script statements in between are enclosed in an HTML comment, so they are ignored too. Navigator properly interprets the SCRIPT tags and ignores the line in the script beginning with the double-slash (//).
Although you are not required to use this technique, it is considered good etiquette so that your pages do not generate unformatted script statements for those not using Navigator 2.0 or later. For simplicity, some of the examples in this book do not hide scripts.

Example: a First Script

Figure 9.1 shows a simple script that displays the following in Navigator:
Hello, net!
That's all, folks.

Notice that there is no difference in appearance between the first line, generated with JavaScript, and the second line, generated with plain HTML.

A simple script


Code within HEAD tags is loaded before the rest of the document.
The SCRIPT tag denotes the beginning of JavaScript code.
The write method of the document object displays its argument (the string "Hello, net!") in the Navigator.
The BODY define the standard HTML content of the page, displaying some simple HTML.

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript1.2">
<!-- Hide script from old browsers.
document.write("Hello, net!")
// End the hiding here. -->
</SCRIPT>
</HEAD>
<BODY>
<P>That's all, folks.
</BODY>
</HTML>
```

You may sometimes see a semicolon at the end of each line of JavaScript. In general, semicolons are optional and are required only if you want to put more than one statement on a single line.

Specifying a File of JavaScript Code

The SRC attribute of the <SCRIPT> tag lets you specify a file as the JavaScript source (rather than embedding the JavaScript in the HTML). For example:

```
<SCRIPT SRC="common.js">

</SCRIPT>
```

This attribute is especially useful for sharing functions among many different pages.
The closing </SCRIPT> tag is required.
JavaScript statements within a <SCRIPT> tag with a SRC attribute are ignored except by browsers that do not support the SRC attribute, such as Navigator 2.

URLs the SRC Attribute Can Specify

The SRC attribute can specify any URL, relative or absolute. For example:

```
<SCRIPT SRC="http://home.netscape.com/functions/jsfuncs.js">
```

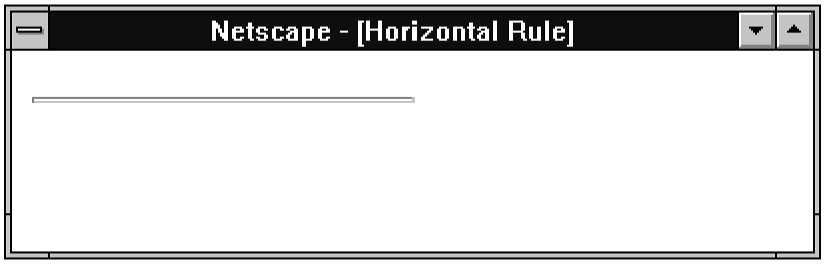
If you load a document with any URL other than a file: URL, and that document itself contains a <SCRIPT SRC="..."> tag, the internal SRC attribute cannot refer to another file: URL.

Requirements for Files Specified by the SRC Attribute

External JavaScript files cannot contain any HTML tags: they must contain only JavaScript statements and function definitions.
External JavaScript files should have the file name suffix .js, and the server must map the .js suffix to the MIME type application/x-javascript, which the server sends back in the HTTP header. To map the suffix to the MIME type, add the following line to the mime.types file in the server's config directory, and then restart the server.
type=application/x-javascript   exts=js
If the server does not map the .js suffix to the application/x-javascript MIME type, Navigator improperly loads the JavaScript file specified by the SRC attribute.
NOTE: This requirement does not apply if you use local files.

Using JavaScript Expressions as HTML Attribute Values

Using JavaScript entities, you can specify a JavaScript expression as the value of an HTML attribute. Entity values are evaluated dynamically. This allows you to create more flexible HTML constructs, because the attributes of one HTML element can depend on information about elements placed previously on the page.
You may already be familiar with HTML character entities by which you can define characters with special numerical codes or names by preceding the name with an ampersand (&) and terminating it with a semicolon (;). For example, you can include a greater-than symbol (>) with the character entity &GT; and a less-than symbol (<) with &LT;.
JavaScript entities also start with an ampersand (&) and end with a semicolon (;). Instead of a name or number, you use a JavaScript expression enclosed in curly braces {}. You can use JavaScript entities only where an HTML attribute value would normally go. For example, suppose you define a variable barWidth. You could create a horizontal rule with the specified percentage width as follows:
<HR WIDTH="&{barWidth};%" ALIGN="LEFT">
So, for example, if barWidth were 50, this statement would create the display shown in the following figure.
Display created using JavaScript entity



As with other HTML, after layout has occurred, the display of a page can change only if you reload the page.
Unlike regular entities which can appear anywhere in the HTML text flow, JavaScript entities are interpreted only on the right-hand side of HTML attribute name/value pairs. For example, if you have this statement:
```
<H4>&{myTitle};</H4>
```
It displays the string myTitle rather than the value of the variable myTitle.

Using Quotation Marks

Whenever you want to indicate a quoted string inside a string literal, use single quotation marks (') to delimit the string literal. This allows the script to distinguish the literal inside the string. In the following example, the function bar contains the literal "left" within a **double-quoted attribute value:**
```
function bar(widthPct) {

   document.write("<HR ALIGN='left' WIDTH=" + widthPct + "%>")
}
```

Here's another example:

```
<INPUT TYPE="button" VALUE="Press Me"
onClick="myfunc('astring')">
```

Specifying Alternate Content with the NOSCRIPT Tag

Use the <NOSCRIPT> tag to specify alternate content for browsers that do not support JavaScript. HTML enclosed within a <NOSCRIPT> tag is displayed by browsers that do not support JavaScript; code within the tag is ignored by Navigator. Note however, that if the user has disabled JavaScript from the Advanced tab of the Preferences dialog, Navigator displays the code within the <NOSCRIPT> tag.
The following example shows a <NOSCRIPT> tag.

```
<NOSCRIPT>

<B>This page uses JavaScript, so you need to get Netscape
Navigator 2.0

or later!

<BR>

<A HREF="http://home.netscape.com/comprod/mirror/index.html">

<IMG SRC="NSNow.gif"></A>
```

If you are using Navigator 2.0 or later, and you see this message,

you should enable JavaScript by on the Advanced page of the

Preferences dialog box.

</NOSCRIPT>

Events

JavaScript applications in Navigator are largely event-driven. Events are actions that usually occur as a result of something the user does. For example, clicking a button is an event, as is changing a text field or moving the mouse over a link. For your script to react to an event, you define event handlers, such as onChange and onClick.

Event Handling:

- Defining an Event Handler
- The Event Object
- Event Capturing
- Validating Form Input

The following table summarizes the JavaScript events. For information on the which versions of JavaScript support each event, see the Client-Side JavaScript Reference.

JavaScript event handlers

Event	**Applies to**	**Occurs when**	**Event handler**
Abort	images	User aborts the loading of an image (for example by clicking a link or clicking the Stop button)	onAbort
Blur	windows and all form elements	User removes input focus from window or form element	onBlur
Change	text fields, textareas, select lists	User changes value of element	onChange
Click	buttons, radio buttons, checkboxes, submit buttons, reset buttons, links	User clicks form element or link	onClick
DragDrop	windows	User drops an object onto the browser window, such as dropping a file on the browser window	onDragDrop
Error	images, windows	The loading of a document or image causes an error	onError
Focus	windows and all form elements	User gives input focus to window or form element	onFocus
KeyDown	documents, images, links, text areas	User depresses a key	onKeyDown
KeyPress	documents, images, links, text areas	User presses or holds down a key	onKeyPress
KeyUp	documents, images, links, text areas	User releases a key	onKeyUp

Load	document body	User loads the page in the Navigator	onLoad
MouseDown	documents, buttons, links	User depresses a mouse button	onMouseDown
MouseMove	nothing by default	User moves the cursor	onMouseMove
MouseOut	areas, links	User moves cursor out of a client-side image map or link	onMouseOut
MouseOver	links	User moves cursor over a link	onMouseOver
MouseUp	documents, buttons, links	User releases a mouse button	onMouseUp
Move	windows	User or script moves a window	onMove
Reset	forms	User resets a form (clicks a Reset button)	onReset
Resize	windows	User or script resizes a window	onResize
Select	text fields, textareas	User selects form element's input field	onSelect
Submit	forms	User submits a form	onSubmit
Unload	document body	User exits the page	onUnload

Defining an Event Handler

You define an event handler (a JavaScript function or series of statements) to handle an event. If an event applies to an HTML tag (that is, the event applies to the JavaScript object created from that tag), then you can define an event handler for it. The name of an event handler is the name of the event, preceded by "on." For example, the event handler for the focus event is onFocus.

To create an event handler for an HTML tag, add an event handler attribute to the tag. Put JavaScript code in quotation marks as the attribute value. The general syntax is

```
<TAG eventHandler="JavaScript Code">
```

where TAG is an HTML tag, eventHandler is the name of the event handler, and JavaScript Code is a sequence of JavaScript statements.

For example, suppose you have created a JavaScript function called compute. You make Navigator call this function when the user clicks a button by assigning the function call to the button's onClick event handler:

```
<INPUT TYPE="button" VALUE="Calculate"
onClick="compute(this.form)">
```

You can put any JavaScript statements as the value of the onClick attribute. These statements are executed when the user clicks the button. To include more than one statement, separate statements with semicolons (;).

Notice that in the preceding example, this.form refers to the current form. The keyword this refers to the current object, which in this case is the button. The construct this.form then refers to the form containing the button. The onClick event handler is a call to the compute function, with the current form as the argument.

When you create an event handler, the corresponding JavaScript object gets a property with the name of the event handler. This property allows you to access the object's event handler. For example, in the preceding example, JavaScript creates a Button object with an onclick property whose value is "compute(this.form)".

Be sure to alternate double quotation marks with single quotation marks. Because event handlers in HTML must be enclosed in quotation marks, you must use single quotation marks to delimit string arguments. For example:

```
<INPUT TYPE="button" NAME="Button1" VALUE="Open Sesame!"

   onClick="window.open('mydoc.html', 'newWin')">
```

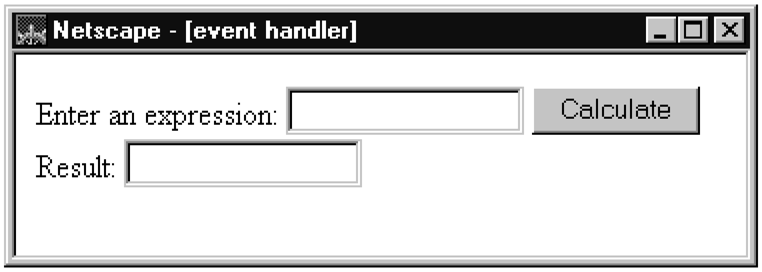
In general, it is good practice to define functions for your event handlers instead of using multiple JavaScript statements:

- It makes your code modular--you can use the same function as an event handler for many different items.
- It makes your code easier to read.

Example: Using an Event Handler

In the form shown in the following figure, you can enter an expression (for example, 2+2) in the first text field, and then click the button. The second text field then displays the value of the expression (in this case, 4).

Form with an event handler



The script for this form is as follows:

```
<HEAD>

<SCRIPT>

<!--- Hide script from old browsers

function compute() {

   if (confirm("Are you sure?"))

      f.result.value = eval(f.expr.value)

   else

      alert("Please come back again.")

}

// end hiding from old browsers -->

</SCRIPT>

</HEAD>
<BODY>

<FORM name =”f”>

Enter an expression:

<INPUT TYPE="text" NAME="expr" SIZE=15 >

<INPUT TYPE="button" VALUE="Calculate" onClick="compute()">

<BR>

Result:

<INPUT TYPE="text" NAME="result" SIZE=15 >

</FORM>

</BODY>
```

The HEAD of the document defines a single function, compute, taking one argument, f, which is a Form object. The function uses the window.confirm method to display a Confirm dialog box with OK and Cancel buttons.
If the user clicks OK, then confirm returns true, and the value of the result text field is set to the value of eval(f.expr.value). The JavaScript function eval evaluates its argument, which can be any string representing any JavaScript expression or statements.
If the user clicks Cancel, then confirm returns false and the alert method displays another message.
The form contains a button with an onClick event handler that calls the compute function. When the user clicks the button, JavaScript calls compute with the argument this.form that denotes the current Form object. In compute, this form is referred to as the argument f.

Calling Event Handlers Explicitly

Follow these guidelines when calling event handlers.

- You can reset an event handler specified by HTML, as shown in the following example.

```
<SCRIPT LANGUAGE="JavaScript">

function fun1() {

   ...

}

function fun2() {

   ...

}

</SCRIPT>
<FORM NAME="myForm">

<INPUT TYPE="button" NAME="myButton"

   onClick="fun1()">

</FORM>
<SCRIPT>

document.myForm.myButton.onclick=fun2

</SCRIPT>
```

- Event handlers are function references, so you must assign fun2 itself, not fun2() (the latter calls fun2 and has whatever type and value fun2 returns).
- Because the event handler HTML attributes are literal function bodies, you cannot use **`<INPUT onClick=fun1>`** in the HTML source to make fun1 the onClick handler for an input. Instead, you must set the value in JavaScript, as in the preceding example.

The Event Object

Each event has an associated event object. The event object provides information about the event, such as the type of event and the location of the cursor at the time of the event. When an event occurs, and if an event handler has been written to handle the event, the event object is sent as an argument to the event handler.
In the case of a MouseDown event, for example, the event object contains the type of event (in this case "MouseDown"), the x and y position of the mouse cursor at the time of the event, a number representing the mouse button used, and a field containing the modifier keys (Control, Alt, Meta, or Shift) that were depressed at the time of the event. The properties of the event object vary from one type of event to another.

Event Capturing

Typically, the object on which an event occurs handles the event. For example, when the user clicks a button, it is often the button's event handler that handles the event. Sometimes you may want the window or document object to handle certain types of events instead of leaving them for the individual parts of the document. For example, you may want the document object to handle all MouseDown events no matter where they occur in the document.
JavaScript's event capturing model allows you to define methods that capture and handle events before they reach their intended target. To accomplish this, the window, document, and layer objects use these event-specific methods:

- captureEvents--captures events of the specified type.
- releaseEvents--ignores the capturing of events of the specified type.
- routeEvent--routes the captured event to a specified object.
- handleEvent--handles the captured event (not a method of layer).

JavaScript 1.1 and earlier versions. Event capturing is not available.
As an example, suppose you wanted to capture all Click events occurring in a window. Briefly, the steps for setting up event capturing are:

1. Enable Event Capturing
2. Define the Event Handler
3. Register the Event Handler

The following sections explain these steps.

Enable Event Capturing

To set up the window to capture all Click events, use a statement such as the following:

window.captureEvents(Event.CLICK);
The argument to captureEvents is a property of the event object and indicates the type of event to capture. To capture multiple events, the argument is a list separated by or (|). For example, the following statement captures Click, MouseDown, and MouseUp events:
window.captureEvents(Event.CLICK | Event.MOUSEDOWN | Event.MOUSEUP)
NOTE: If a window with frames needs to capture events in pages loaded from different locations, you need to use captureEvents in a signed script and call enableExternalCapture. For information on signed scripts, see Chapter 14, "JavaScript Security."

Define the Event Handler

Next, define a function that handles the event. The argument e is the event object for the event.

```
function clickHandler(e) {

   //What goes here depends on how you want to handle the event.

   //This is described below.

}
```

You have the following options for handling the event:

- Return true. In the case of a link, the link is followed and no other event handler is checked. If the event cannot be canceled, this ends the event handling for that event.

  ```
  function clickHandler(e) {

     return true;

  }
  ```

  This allows the event to be completely handled by the document or window. The event is not handled by any other object, such as a button in the document or a child frame of the window.

- Return false. In the case of a link, the link is not followed. If the event is non-cancelable, this ends the event handling for that event.

  ```
  function clickHandler(e) {

     return false;

  }
  ```

  This allows you to suppress the handling of an event type. The event is not handled by any other object, such as a button in the document or a child frame of the window. You can use this, for example, to suppress the right mouse button in an application.

- Call routeEvent. JavaScript looks for other event handlers for the event. If another object is attempting to capture the event (such as the document), JavaScript calls its event handler. If no other object is attempting to capture the event, JavaScript looks for an event handler for the event's original target (such as a button). The routeEvent function returns the value returned by the event handler. The capturing object can look at this return and decide how to proceed.

  When routeEvent calls an event handler, the event handler is activated. If routeEvent calls an event handler whose function is to display a new page, the action takes place without returning to the capturing object.

```
function clickHandler(e) {

    var retval = routeEvent(e);

    if (retval == false) return false;

    else return true;

  }
```

- Call the handleEvent method of an event receiver. Any object that can register event handlers is an event receiver. This method explicitly calls the event handler of the event receiver and bypasses the capturing hierarchy. For example, if you wanted all Click events to go to the first link on the page, you could use:

```
function clickHandler(e) {

   window.document.links[0].handleEvent(e);

}
```

  As long as the link has an onClick handler, the link will handle any click event it receives.

Register the Event Handler

Finally, register the function as the window's event handler for that event:
window.onClick = clickHandler;

A Complete Example

In the following example, the window and document capture and release events:

```
<HTML>

<SCRIPT>
function fun1(e) {
   alert ("The window got an event of type: " + e.type +

      " and will call routeEvent.");

   window.routeEvent(e);

   alert ("The window returned from routeEvent.");

   return true;

}
function fun2(e) {

   alert ("The document got an event of type: " + e.type);

   return false;

}
function setWindowCapture() {

   window.captureEvents(Event.CLICK);

}
function releaseWindowCapture() {

   window.releaseEvents(Event.CLICK);

}
function setDocCapture() {

   document.captureEvents(Event.CLICK);

}
function releaseDocCapture() {

   document.releaseEvents(Event.CLICK);

}
window.onclick=fun1;

document.onclick=fun2;
</SCRIPT>

...

</HTML>
```

Validating Form Input

One of the most important uses of JavaScript is to validate form input to server-based programs such as server-side JavaScript applications or CGI programs. This is useful for several reasons:

- It reduces load on the server. "Bad data" are already filtered out when input is passed to the server-based program.
- It reduces delays in case of user error. Validation otherwise has to be performed on the server, so data must travel from client to server, be processed, and then returned to client for valid input.
- It simplifies the server-based program.

Generally, you'll want to validate input in (at least) two places:

- As the user enters it, with an onChange event handler on each form element that you want validated.
- When the user submits the form, with an onClick event handler on the button that submits the form.

The JavaScript page on DevEdge contains pointers to sample code. One such pointer is a complete set of form validation functions. This section presents some simple examples, but you should check out the samples on DevEdge.

Example Validation Functions

The following are some simple validation functions.

```
<HEAD>

<SCRIPT LANGUAGE="JavaScript">

function isaPosNum(s) {

   return (parseInt(s) > 0)

}
function qty_check(item, min, max) {

   var returnVal = false

   if (!isaPosNum(item.value))

      alert("Please enter a positive number")

   else if (parseInt(item.value) < min)

      alert("Please enter a " + item.name + " greater than " +
min)
```

```
   else if (parseInt(item.value) > max)

      alert("Please enter a " + item.name + " less than " + max)

   else

      returnVal = true

   return returnVal

}
function validateAndSubmit(theform) {
   if (qty_check(theform.quantity, 0, 999)) {
      alert("Order has been Submitted")
      return true
   }
   else {
      alert("Sorry, Order Cannot Be Submitted!")
      return false
   }
}
</SCRIPT>

</HEAD>
```

isaPosNum is a simple function that returns true if its argument is a positive number, and false otherwise.
qty_check takes three arguments: an object corresponding to the form element being validated (item) and the minimum and maximum allowable values for the item (min and max). It checks that the value of item is a number between min and max and displays an alert if it is not.
validateAndSubmit takes a Form object as its argument; it uses qty_check to check the value of the form element and submits the form if the input value is valid. Otherwise, it displays an alert and does not submit the form.

Using the Validation Functions

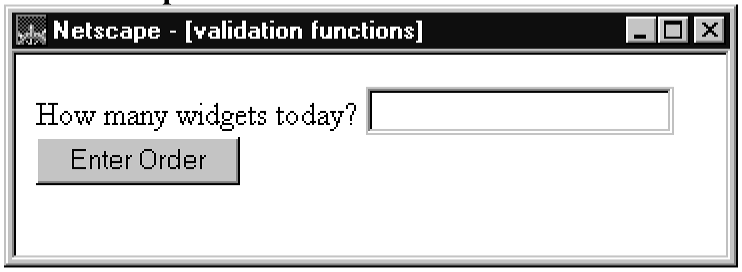
In this example, the BODY of the document uses qty_check as an onChange event handler for a text field and validateAndSubmit as the onClick event handler for a button.

```
<BODY>

<FORM NAME="widget_order" ACTION="lwapp.html" METHOD="post">

How many widgets today?

<INPUT TYPE="text" NAME="quantity" onChange="qty_check(this, 0,
999)">

<BR>
```

```
<INPUT TYPE="button" VALUE="Enter Order"
onClick="validateAndSubmit(this.form)">

</FORM>

</BODY>
```

This form submits the values to a page in a server-side JavaScript application called lwapp.html. It also could be used to submit the form to a CGI program. The form is shown in the following figure.

A JavaScript form



The onChange event handler is triggered when you change the value in the text field and move focus from the field by either pressing the Tab key or clicking the mouse outside the field. Notice that both event handlers use this to represent the current object: in the text field, it is used to pass the JavaScript object corresponding to the text field to qty_check, and in the button it is used to pass the JavaScript Form object to validateAndSubmit.

To submit the form to the server-based program, this example uses a button that calls validateAndSubmit, which submits the form using the submit method, if the data are valid. You can also use a submit button (defined by <INPUT TYPE="submit">) and then put an onSubmit event handler on the form that returns false if the data are not valid. For example,

```
<FORM NAME="widget_order" ACTION="lwapp.html" METHOD="post"

   onSubmit="return qty_check(theform.quantity, 0, 999)">

...

<INPUT TYPE="submit">

...

</FORM>
```

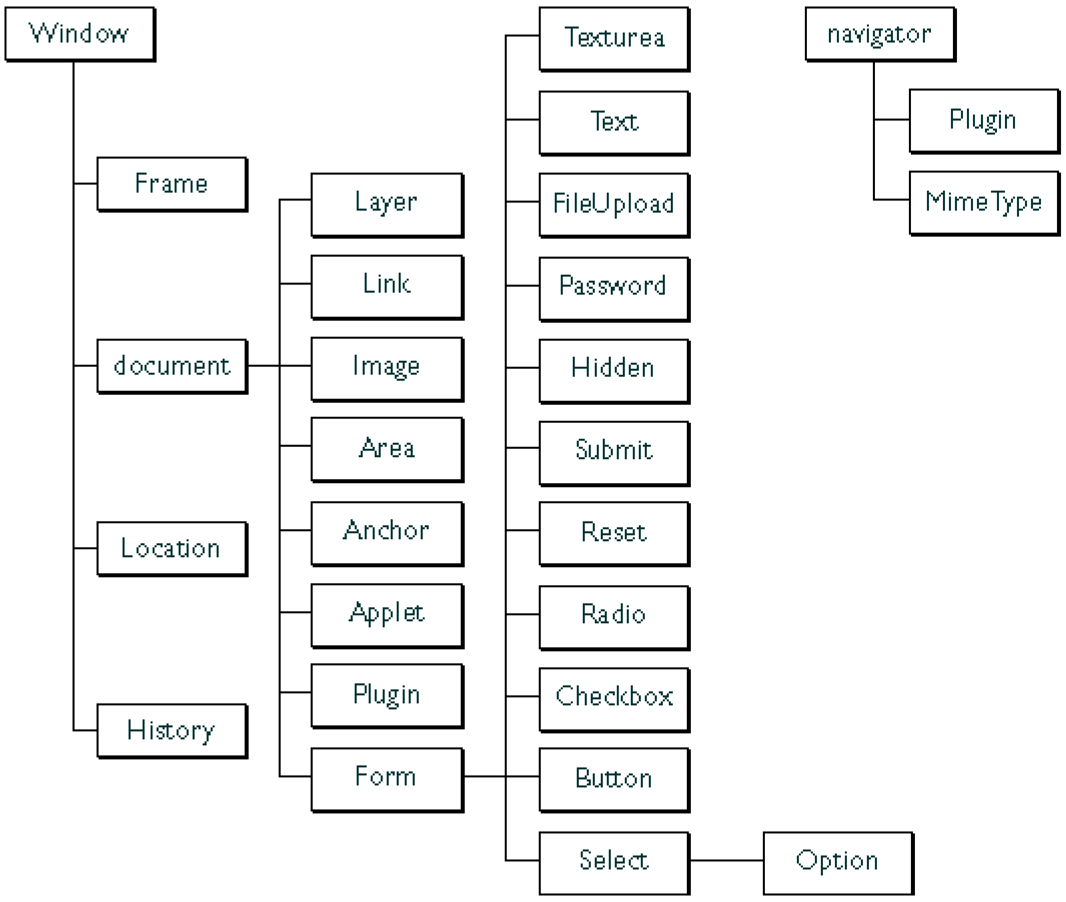
When qty_check returns false if the data are invalid, the onSubmit handler will prohibit the form from being submitted.

Using Navigator Objects

Navigator Object Hierarchy (Same for Internet Explorer)

When you load a document in Navigator, it creates a number of JavaScript objects with property values based on the HTML in the document and other pertinent information. These objects exist in a hierarchy that reflects the structure of the HTML page itself. The following figure illustrates this object hierarchy.

Navigator object hierarchy



In this hierarchy, an object's "descendants" are properties of the object. For example, a form named form1 is an object as well as a property of document, and is referred to as document.form1.

.

Every page has the following objects:

- navigator: has properties for the name and version of Navigator being used, for the MIME types supported by the client, and for the plug-ins installed on the client.

- window: the top-level object; has properties that apply to the entire window. Each "child window" in a frames document also has a window object.
- document: contains properties based on the content of the document, such as title, background color, links, and forms.
- location: has properties based on the current URL.
- history: contains properties representing URLs the client has previously requested.

Depending on its content, the document may contain other objects. For instance, each form (defined by a FORM tag) in the document has a corresponding Form object.
To refer to specific properties, you must specify the object name and all its ancestors. Generally, an object gets its name from the NAME attribute of the corresponding HTML tag. For more information and examples, see Chapter 12, "Using Windows and Frames."
For example, the following code refers to the value property of a text field named text1 in a form named myform in the current document:
document.myform.text1.value
If an object is on a form, you must include the form name when referring to that object, even if the object does not need to be on a form. For example, images do not need to be on a form. The following code refers to an image that is on a form:
document.imageForm.aircraft.src='f15e.gif'
The following code refers to an image that is not on a form:
document.aircraft.src='f15e.gif'

Document Properties: an Example

The properties of the document object are largely content-dependent. That is, they are created based on the HTML in the document. For example, document has a property for each form and each anchor in the document.

```
Suppose you create a page named simple.html that contains the
following HTML:
<HEAD><TITLE>A Simple Document</TITLE>

<SCRIPT>

function update(form) {

   alert("Form being updated")

}

</SCRIPT>

</HEAD>
<BODY>

<FORM NAME="myform" ACTION="foo.cgi" METHOD="get" >Enter a value:

<INPUT TYPE="text" NAME="text1" VALUE="blahblah" SIZE=20 >
```

```
Check if you want:

<INPUT TYPE="checkbox" NAME="Check1" CHECKED

   onClick="update(this.form)"> Option #1

<P>

<INPUT TYPE="button" NAME="button1" VALUE="Press Me"

   onClick="update(this.form)">

</FORM>

</BODY>
```

Given the preceding HTML example, the basic objects might have properties like those shown in the following table.

Example object property values

Property	**Value**
document.title	"A Simple Document"
document.fgColor	#000000
document.bgColor	#ffffff
location.href	"http://www.royalairways.com/samples/simple.html"
history.length	7

Notice that the value of document.title reflects the value specified in the TITLE tag. The values for document.fgColor (the color of text) and document.bgColor (the background color) were not set in the HTML, so they are based on the default values specified in the Preferences dialog box (when the user chooses Preferences from the Navigator Edit menu).

Because the document has a form, there is also a Form object called myform (based on the form's NAME attribute) that has child objects for the checkbox and the button. Each of these objects has a name based on the NAME attribute of the HTML tag that defines it, as follows:

- document.myform, the form
- document.myform.Check1, the checkbox
- document.myform.button1, the button

The Form object myform has other properties based on the attributes of the FORM tag, for example,

- action is http://www.royalairways.com/samples/mycgi.cgi, the URL to which the form is submitted.
- method is "get," based on the value of the METHOD attribute.
- length is 3, because there are three input elements in the form.

The Form object has child objects named button1 and text1, corresponding to the button and text fields in the form. These objects have their own properties based on their HTML attribute values, for example,

- button1.value is "Press Me"
- button1.name is "Button1"
- text1.value is "blahblah"
- text1.name is "text1"

In practice, you refer to these properties using their full names, for example, document.myform.button1.value. This full name is based on the Navigator object hierarchy, starting with document, followed by the name of the form, myform, then the element name, button1, and, finally, the property name.

JavaScript Reflection and HTML Layout

JavaScript object property values are based on the content of your HTML document, sometimes referred to as reflection because the property values reflect the HTML. To understand JavaScript reflection, it is important to understand how the Navigator performs layout--the process by which Navigator transforms HTML tags into graphical display on your computer.
Generally, layout happens sequentially in the Navigator: the Navigator starts at the top of the HTML file and works downward, displaying output to the screen as it goes. Because of this "top-down" behavior, JavaScript reflects only HTML that it has encountered. For example, suppose you define a form with a couple of text-input elements:

```
<FORM NAME="statform">

<INPUT TYPE = "text" name = "userName" size = 20>

<INPUT TYPE = "text" name = "Age" size = 3>
</form>
```

These form elements are reflected as JavaScript objects that you can use after the form is defined: document.statform.userName and document.statform.Age. For example, you could display the value of these objects in a script after defining the form:

```
<SCRIPT>

document.write(document.statform.userName.value)

document.write(document.statform.Age.value)
```

```
</SCRIPT>
```

However, if you tried to do this before the form definition (above it in the HTML page), you would get an error, because the objects don't exist yet in the Navigator.
Likewise, once layout has occurred, setting a property value does not affect its value or appearance. For example, suppose you have a document title defined as follows:
<TITLE>My JavaScript Page</TITLE>
This is reflected in JavaScript as the value of document.title. Once the Navigator has displayed this in the title bar of the Navigator window, you cannot change the value in JavaScript. If you have the following script later in the page, it will not change the value of document.title, affect the appearance of the page, or generate an error.
document.title = "The New Improved JavaScript Page"
There are some important exceptions to this rule: you can update the value of form elements dynamically. For example, the following script defines a text field that initially displays the string "Starting Value." Each time you click the button, you add the text "...Updated!" to the value.

```
<FORM NAME="demoForm">

<INPUT TYPE="text" NAME="mytext" SIZE="40" VALUE="Starting
Value">

<P><INPUT TYPE="button" VALUE="Click to Update Text Field"

   onClick="document.demoForm.mytext.value += '...Updated!' ">

</FORM>
```

This is a simple example of updating a form element after layout.
Using event handlers, you can also change a few other properties after layout has completed, such as document.bgColor.

Key Navigator Objects

This section describes some of the most useful Navigator objects: window, Frame, document, Form, location, history, and navigator.

window and Frame Objects

The window object is the "parent" object for all other objects in Navigator. You can create multiple windows in a JavaScript application. A Frame object is defined by the FRAME tag in a FRAMESET document. Frame objects have the same properties and methods as window objects and differ only in the way they are displayed.
The window object has numerous useful methods, including the following:

- open and close: Opens and closes a browser window; you can specify the size of the window, its content, and whether it has a button bar, location field, and other "chrome" attributes.
- alert: Displays an Alert dialog box with a message.
- confirm: Displays a Confirm dialog box with OK and Cancel buttons.
- prompt: Displays a Prompt dialog box with a text field for entering a value.

- blur and focus: Removes focus from, or gives focus to a window.
- scrollTo: Scrolls a window to a specified coordinate.
- setInterval: Evaluates an expression or calls a function each time the specified period elapses.
- setTimeout: Evaluates an expression or calls a function once after the specified period elapses.

window also has several properties you can set, such as location and status.
You can set location to redirect the client to another URL. For example, the following statement redirects the client to the Netscape home page, as if the user had clicked a hyperlink or otherwise loaded the URL:
location = "http://home.netscape.com"
You can use the status property to set the message in the status bar at the bottom of the client window; for more information.

document Object

Each page has one document object.
Because its write and writeln methods generate HTML, the document object is one of the most useful Navigator objects. For information on write and writeln,

The document object has a number of properties that reflect the colors of the background, text, and links in the page: bgColor, fgColor, linkColor, alinkColor, and vlinkColor. Other useful document properties include lastModified, the date the document was last modified, referrer, the previous URL the client visited, and URL, the URL of the document. The cookie property enables you to get and set cookie values

The document object is the ancestor for all the Anchor, Applet, Area, Form, Image, Layer, Link, and Plugin objects in the page.
Users can print and save generated HTML by using the commands on the Navigator File menu (JavaScript 1.1 and later).

Form Object

Each form in a document creates a Form object. Because a document can contain more than one form, Form objects are stored in an array called forms. The first form (topmost in the page) is forms[0], the second forms[1], and so on. In addition to referring to each form by name, you can refer to the first form in a document as
document.forms[0]
Likewise, the elements in a form, such as text fields, radio buttons, and so on, are stored in an elements array. You could refer to the first element (regardless of what it is) in the first form as
document.forms[0].elements[0]
Each form element has a form property that is a reference to the element's parent form. This property is especially useful in event handlers, where you might need to refer to another element on the current 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>
```

location Object

The location object has properties based on the current URL. For example, the hostname property is the server and domain name of the server hosting the document.
The location object has two methods:

- reload forces a reload of the window's current document.
- replace loads the specified URL over the current history entry.

history Object

The history object contains a list of strings representing the URLs the client has visited. You can access the current, next, and previous history entries by using the history object's current, next, and previous properties. You can access the other history values using the history array. This array contains an entry for each history entry in source order; each array entry is a string containing a URL.
You can also redirect the client to any history entry by using the go method. For example, the following code loads the URL that is two entries back in the client's history list.
history.go(-2)
The following code reloads the current page:
history.go(0)
The history list is displayed in the Navigator Go menu.

navigator Object

The navigator object contains information about the version of Navigator in use. For example, the appName property specifies the name of the browser, and the appVersion property specifies version information for the Navigator.
The navigator object has three methods:

- javaEnabled specifies whether Java is enabled
- preference lets you use a signed script to get or set various user preferences (JavaScript 1.2 and later)
- taintEnabled specifies whether data tainting is enabled (JavaScript 1.1 only)

Navigator Object Arrays

Some Navigator objects have properties whose values are themselves arrays. These arrays are used to store information when you don't know ahead of time how many values there will be. The following table shows which properties of which objects have arrays as values.

Predefined JavaScript arrays

Object	**Property**	**Description**
document	anchors	Reflects a document's <A> tags that contain a NAME attribute in source order
	applets	Reflects a document's <APPLET> tags in source order
	embeds	Reflects a document's <EMBED> tags in source order
	forms	Reflects a document's <FORM> tags in source order
	images	Reflects a document's <IMG> tags in source order (images created with the Image() constructor are not included in the images array)
	layers	Reflects a document's <LAYER> and <ILAYER> tags in source order
	links	Reflects a document's <AREA HREF="..."> tags, <A HREF=""> tags, and Link objects created with the link method in source order
Form	elements	Reflects a form's elements (such as Checkbox, Radio, and Text objects) in source order
Function	arguments	Reflects the arguments to a function
navigator	mimeTypes	Reflects all the MIME types supported by the client (either internally, via helper applications, or by plug-ins)
	plugins	Reflects all the plug-ins installed on the client in source order
select	options	Reflects the options in a Select object (<OPTION> tags) in source order

window	frames	Reflects all the <FRAME> tags in a window containing a <FRAMESET> tag in source order
	history	Reflects a window's history entries

You can index arrays by either their ordinal number 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.
For example, suppose the following form element is defined:
<INPUT TYPE="text" NAME="Comments">
If you need to refer to this form element by name, you can specify document.forms["Comments"].
All predefined JavaScript arrays have a length property that indicates the number of elements in the array. For example, to obtain the number of forms in a document, use its length property: document.forms.length.
JavaScript 1.0. You must index arrays by their ordinal number, for example document.forms[0].

Using the write Method

The write method of document displays output in the Navigator. "Big deal," you say, "HTML already does that." But in a script you can do all kinds of things you can't do with ordinary HTML. For example, you can display text conditionally or based on variable arguments. For these reasons, write is one of the most often-used JavaScript methods.
The write method takes any number of string arguments that can be string literals or variables. You can also use the string concatenation operator (+) to create one string from several when using write.
Consider the following script, which generates dynamic HTML with JavaScript:

```
<HEAD>

<SCRIPT>

<!--- Hide script from old browsers

// This function displays a horizontal bar of specified width

function bar(widthPct) {

   document.write("<HR ALIGN='left' WIDTH=" + widthPct + "%>");

}
// This function displays a heading of specified level and some
text
```

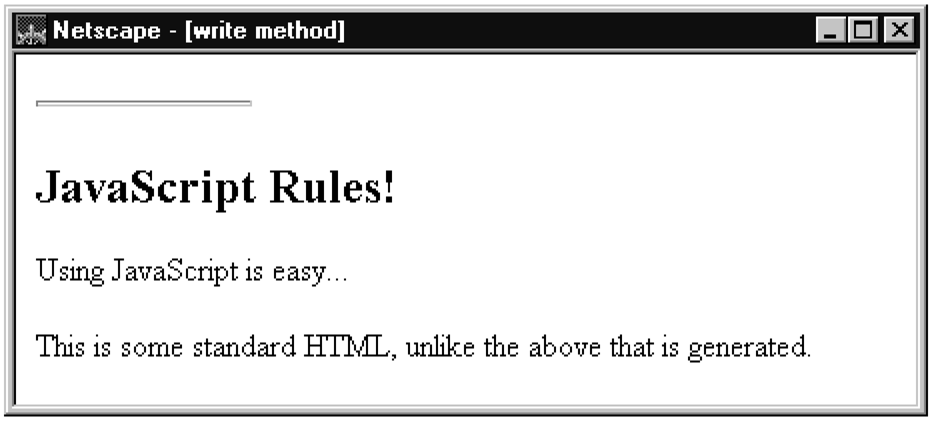
```
function output(headLevel, headText, text) {

   document.write("<H", headLevel, ">", headText, "</H",

      headLevel, "><P>", text)

}

// end script hiding from old browsers -->

</SCRIPT>

</HEAD>
<BODY>

<SCRIPT>

<!--- hide script from old browsers

bar(25)

output(2, "JavaScript Rules!", "Using JavaScript is easy...")

// end script hiding from old browsers -->

</SCRIPT>

<P> This is some standard HTML, unlike the above that is
generated.

</BODY>
```

The HEAD of this document defines two functions:

- bar, which displays an HTML horizontal rule of a width specified by the function's argument.
- output, which displays an HTML heading of the level specified by the first argument, heading text specified by the second argument, and paragraph text specified by the third argument.

The document BODY then calls the two functions to produce the display shown in the following figure.

Display created using JavaScript functions



JavaScript Rules!

Using JavaScript is easy...

This is some standard HTML, unlike the above that is generated.

The following line creates the output of the bar function:

```
document.write("<HR ALIGN='left' WIDTH=", widthPct, "%>")
```

Notice that the definition of bar uses single quotation marks inside double quotation marks. You must do this whenever you want to indicate a quoted string inside a string literal. Then the call to bar with an argument of 25 produces output equivalent to the following HTML:

```
<HR ALIGN="left" WIDTH=25%>
```

write has a companion method, writeln, which adds a newline sequence (a carriage return or a carriage return and linefeed, depending on the platform) at the end of its output. Because HTML generally ignores new lines, there is no difference between write and writeln except inside tags such as PRE, which respect carriage returns.

Printing Output

Navigator versions 3.0 and later print output created with JavaScript. To print output, the user chooses Print from the Navigator File menu. Navigator 2.0 does not print output created with JavaScript.

If you print a page that contains layers (Navigator 4.0 and later), each layer is printed separately on the same page. For example, if three layers overlap each other in the browser, the printed page shows each layers separately.

If you choose Page Source from the Navigator View menu or View Frame Source from the right-click menu, the web browser displays the content of the HTML file with the generated HTML. If you instead want to view the HTML source showing the scripts which generate HTML (with the document.write and document.writeln methods), do not use the Page Source and View Frame Source menu items. In this situation, use the view-source: protocol. For example, assume the file file://c|/test.html contains this text:

```
<HTML>

<BODY>

Hello,
<SCRIPT>document.write(" there.")</SCRIPT>

</BODY>

</HTML>
```

If you load this URL into the web browser, it displays the following:

Hello, there.
If you choose Page Source from the View menu, the browser displays the following:

```
<HTML>

<BODY>Hello,  there.</BODY>

</HTML>
```

If you load `view-source:file://c|/test.html`, the browser displays the following:

```
<HTML>

<BODY>Hello, <SCRIPT>document.write(" there.")</SCRIPT>

</BODY>

</HTML>
```

Displaying Output

JavaScript in Navigator generates its results from the top of the page down. Once text has been displayed, you cannot change it without reloading the page. In general, you cannot update part of a page without updating the entire page. However, you can update the following:

- A layer's contents.
- A "subwindow" in a frame separately. For more information, see Chapter 12, "Using Windows and Frames."
- Form elements without reloading the page; see "Example: Using an Event Handler" on page 160.

Using Windows and Frames

JavaScript lets you create and manipulate windows and frames for presenting HTML content. The window object is the top-level object in the JavaScript client hierarchy; Frame objects are similar to window objects, but correspond to "sub-windows" created with the FRAME tag in a FRAMESET document.
This chapter contains the following sections:

- Opening and Closing Windows
- Using Frames
- Referring to Windows and Frames

- Navigating Among Windows and Frames

Opening and Closing Windows

A window is created automatically when you launch Navigator; you can open another window by choosing New then Navigator Window from the File menu. You can close a window by choosing either Close or Exit from the File menu. You can also open and close windows programmatically with JavaScript.

Opening a Window

You can create a window with the open method. The following statement creates a window called msgWindow that displays the contents of the file sesame.html:

```
msgWindow=window.open("sesame.html")
```

The following statement creates a window called homeWindow that displays the Netscape home page:

```
homeWindow=window.open("http://home.netscape.com")
```

Windows can have two names. The following statement creates a window with two names. The first name, msgWindow, is a variable that refers to the window object. This object has information on the window's properties, methods, and containership. When you create the window, you can also supply a second name, in this case displayWindow, to refer to that window as the target of a form submit or hypertext link.

```
msgWindow=window.open("sesame.html","displayWindow")
```

A window name is not required when you create a window. But the window must have a name if you want to refer to it from another window.

When you open a window, you can specify attributes such as the window's height and width and whether the window contains a toolbar, location field, or scrollbars. The following statement creates a window without a toolbar but with scrollbars:

```
msgWindow=window.open

   ("sesame.html","displayWindow","toolbar=no,scrollbars=yes")
```

Closing a Window

You can close a window with the close method. You cannot close a frame without closing the entire parent window.

Each of the following statements closes the current window:

```
window.close()
self.close()
close()
```

Do not use the third form, close(), in an event handler. Because of how JavaScript figures out what object a method call refers to, inside an event handler it will get the wrong object.

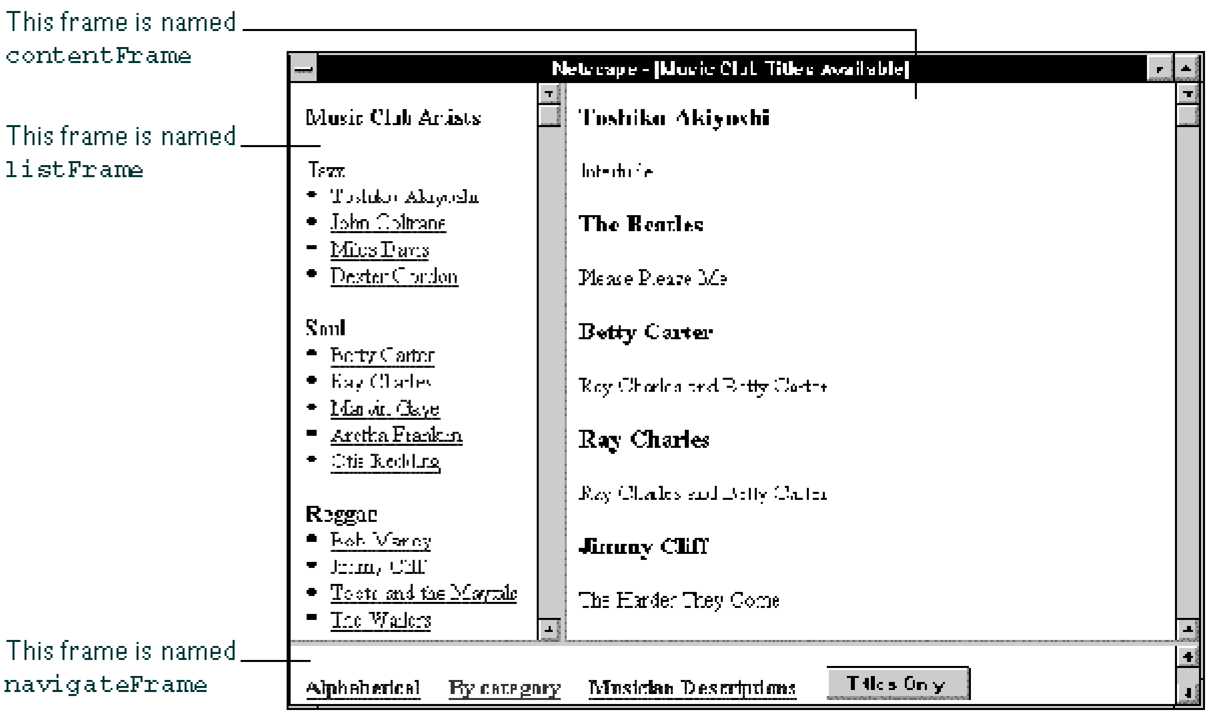
The following statement closes a window called msgWindow:

```
msgWindow.close()
```

Using Frames

A **frameset** is a special type of window that can display multiple, independently scrollable **frames** on a single screen, each with its own distinct URL. The frames in a frameset can point to different URLs and be targeted by other URLs, all within the same window. The series of frames in a frameset make up an HTML page.
The following figure depicts a window containing three frames. The frame on the left is named listFrame; the frame on the right is named contentFrame; the frame on the bottom is named navigateFrame.

A page with frames


This frame is named contentFrame
This frame is named listFrame
This frame is named navigateFrame
Netscape - [Music Club Titles Available]
Music Club Artists
Alphabetical
By category
Musician Descriptions
Titles Only

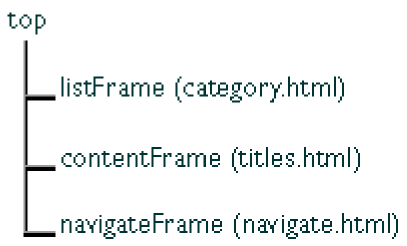
Creating a Frame

You create a frame by using the FRAMESET tag in an HTML document; this tag's sole purpose is to define the frames in a page.
Example 1. The following statement creates the frameset shown previously:

```
<FRAMESET ROWS="90%,10%">

   <FRAMESET COLS="30%,70%">

      <FRAME SRC=category.html NAME="listFrame">

      <FRAME SRC=titles.html NAME="contentFrame">

   </FRAMESET>

   <FRAME SRC=navigate.html NAME="navigateFrame">

</FRAMESET>
```

The following figure shows the hierarchy of the frames. All three frames have the same parent, even though two of the frames are defined within a separate frameset. This is because a frame's parent is its parent window, and a frame, not a frameset, defines a window.

An example frame hierarchy


top
listFrame (category.html)
contentFrame (titles.html)
navigateFrame (navigate.html)

You can refer to the previous frames using the frames array as follows. (For information on the frames array, see the window object in the Client-Side JavaScript Reference.)

- listFrame is top.frames[0]
- contentFrame is top.frames[1]
- navigateFrame is top.frames[2]

Example 2. Alternatively, you could create a window like the previous one but in which the top two frames have a parent separate from navigateFrame. The top-level frameset would be defined as follows:

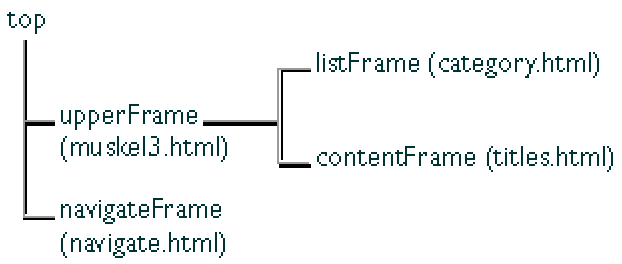
```
<FRAMESET ROWS="90%,10%">

   <FRAME SRC=muskel3.html NAME="upperFrame">

   <FRAME SRC=navigate.html NAME="navigateFrame">

</FRAMESET>
```

The file muskel3.html contains the skeleton for the upper frames and defines the following frameset:

```
<FRAMESET COLS="30%,70%">

   <FRAME SRC=category.html NAME="listFrame">

   <FRAME SRC=titles.html NAME="contentFrame">

</FRAMESET>
```

The following figure shows the hierarchy of the frames. upperFrame and navigateFrame share a parent: the top window. listFrame and contentFrame share a parent: upperFrame.

Another example frame hierarchy



You can refer to the previous frames using the frames array as follows. (For information on the frames array, see the window object in the Client-Side JavaScript Reference.)

- upperFrame is top.frames[0]
- navigateFrame is top.frames[1]
- listFrame is upperFrame.frames[0] or top.frames[0].frames[0]
- contentFrame is upperFrame.frames[1] or top.frames[0].frames[1]

Updating a Frame

You can update the contents of a frame by using the location property to set the URL, as long as you specify the frame hierarchy.
For example, suppose you are using the frameset described in Example 2 in the previous section. If you want users to be able to close the frame containing the alphabetical or categorical list of artists (in the frame listFrame) and view only the music titles sorted by musician (currently in the frame contentFrame), you could add the following button to navigateFrame:

```
<INPUT TYPE="button" VALUE="Titles Only"
   onClick="top.frames[0].location='artists.html'">
When a user clicks this button, the file artists.html is loaded
into the frame upperFrame; the frames listFrame and contentFrame
close and no longer exist.
```

Referring To and Navigating Among Frames

Because frames are a type of window, you refer to frames and navigate among them as you do windows. See "Referring to Windows and Frames" on page 197 and "Navigating Among Windows and Frames" on page 200.

Creating and Updating Frames: an Example

If you designed the frameset in the previous section to present the available titles for a music club, the frames and their HTML files could have the following content:

- category.html, in the frame listFrame, contains a list of musicians sorted by category.
- titles.html, in the frame contentFrame, contains an alphabetical list of musicians and the titles available for each.
- navigate.html, in the frame navigateFrame, contains hypertext links the user can click to choose how the musicians are displayed in listFrame: alphabetically or by category. This file also defines a hypertext link users can click to display a description of each musician.

- An additional file, alphabet.html, contains a list of musicians sorted alphabetically. This file is displayed in listFrame when the user clicks the link for an alphabetical list.

The file category.html (the categorical list) contains code similar to the following:

```
<B>Music Club Artists</B>

<P><B>Jazz</B>

<LI><A HREF=titles.html#0001 TARGET="contentFrame">Toshiko
Akiyoshi</A>

<LI><A HREF=titles.html#0006 TARGET="contentFrame">John
Coltrane</A>

<LI><A HREF=titles.html#0007 TARGET="contentFrame">Miles
Davis</A>

<LI><A HREF=titles.html#0010 TARGET="contentFrame">Dexter
Gordon</A>
<P><B>Soul</B>

<LI><A HREF=titles.html#0003 TARGET="contentFrame">Betty
Carter</A>

<LI><A HREF=titles.html#0004 TARGET="contentFrame">Ray
Charles</A>

...
The file alphabet.html (the alphabetical list) contains code
similar to the following:
<B>Music Club Artists</B>

<LI><A HREF=titles.html#0001 TARGET="contentFrame">Toshiko
Akiyoshi</A>

<LI><A HREF=titles.html#0002 TARGET="contentFrame">The
Beatles</A>

<LI><A HREF=titles.html#0003 TARGET="contentFrame">Betty
Carter</A>

<LI><A HREF=titles.html#0004 TARGET="contentFrame">Ray
Charles</A>

...
The file navigate.html (the navigational links at the bottom of
the screen) contains code similar to the following. Notice that
the target for artists.html is "_parent." When the user clicks
this link, the entire window is overwritten, because the top
window is the parent of navigateFrame.
```

```
<A HREF=alphabet.html TARGET="listFrame"><B>Alphabetical</B></A>

&nbsp&nbsp&nbsp

<A HREF=category.html TARGET="listFrame"><B>By category</B></A>

&nbsp&nbsp&nbsp

<A HREF="artists.html" TARGET="_parent">

   <B>Musician Descriptions</B></A>
The file titles.html (the main file, displayed in the frame on
the right) contains code similar to the following:
<A NAME="0001"><H3>Toshiko Akiyoshi</H3></A>

<P>Interlude
<A NAME="0002"><H3>The Beatles</H3></A>

<P>Please Please Me
<A NAME="0003"><H3>Betty Carter</H3></A>

<P>Ray Charles and Betty Carter

...
```

Referring to Windows and Frames

The name you use to refer to a window depends on whether you are referring to a window's properties, methods, and event handlers or to the window as the target of a form submit or a hypertext link.
Because the window object is the top-level object in the JavaScript client hierarchy, the window is essential for specifying the containership of objects in any window.

Referring to Properties, Methods, and Event Handlers

You can refer to the properties, methods, and event handlers of the current window or another window (if the other window is named) using any of the following techniques:

- self or window: self and window are synonyms for the current window, and you can use them optionally to refer to the current window. For example, you can close the current window by calling either window.close() or self.close().
- top or parent: top and parent are also synonyms that you can use in place of the window name. top can be used for any window; it refers to the topmost Navigator window. parent can be used for a frame; it refers to the frameset window that contains that frame. For example, for the frame frame1, the statement parent.frame2.document.bgColor="teal" changes the background color of the frame named frame2 to teal, where frame2 is a sibling frame in the current frameset.

- The name of a window variable: The window variable is the variable specified when a window is opened. For example, msgWindow.close() closes a window called msgWindow.
- Omit the window name: Because the existence of the current window is assumed, you do not have to refer to the name of the window when you call its methods and assign its properties. For example, close() closes the current window. However, when you open or close a window within an event handler, you must specify window.open() or window.close() instead of simply using open() or close(). Because of the scoping of static objects in JavaScript, a call to close() without specifying an object name is equivalent to document.close().

For more information on these techniques for referring to windows, see the window object in the Client-Side JavaScript Reference.
Example 1: refer to the current window. The following statement refers to a form named musicForm in the current window. The statement displays an alert if a checkbox is checked.

```
if (document.musicForm.checkbox1.checked) {

   alert('The checkbox on the musicForm is checked!')}
```

Example 2: refer to another window. The following statements refer to a form named musicForm in a window named checkboxWin. The statements determine if a checkbox is checked, check the checkbox, determine if the second option of a Select object is selected, and select the second option of the Select object. Even though object values are changed in another window (checkboxWin), the current window remains active: checking the checkbox and selecting the selection option do not give focus to the window.

```
// Determine if a checkbox is checked

if (checkboxWin.document.musicForm.checkbox2.checked) {

   alert('The checkbox on the musicForm in checkboxWin is
checked!')}
// Check the checkbox

checkboxWin.document.musicForm.checkbox2.checked=true
// Determine if an option in a Select object is selected

if
(checkboxWin.document.musicForm.musicTypes.options[1].selected)

   {alert('Option 1 is selected!')}
// Select an option in a Select object

checkboxWin.document.musicForm.musicTypes.selectedIndex=1
```

Example 3: refer to a frame in another window. The following statement refers to a frame named frame2 that is in a window named window2. The statement changes the background color of frame2 to violet. The frame name, frame2, must be specified in the FRAMESET tag that creates the frameset.
window2.frame2.document.bgColor="violet"

Referring to a Window in a Form Submit or Hypertext Link

You use a window's name (not the window variable) when referring to a window as the target of a form submit or hypertext link (the TARGET attribute of a FORM or A tag). The window you specify is the window into which the link is loaded or, for a form, the window in which server responses are displayed.
The following example creates a hypertext link to a second window. The example has a button that opens an empty window named window2, then a link that loads the file doc2.html into the newly opened window, and then a button that closes the window.

```
<FORM>

<INPUT TYPE="button" VALUE="Open Second Window"

   onClick="msgWindow=window.open('','window2',

   'resizable=no,width=200,height=200')">

<P>

<A HREF="doc2.html" TARGET="window2"> Load a file into
window2</A>

<P>

<INPUT TYPE="button" VALUE="Close Second Window"

   onClick="msgWindow.close()">

</FORM>
```

If the user selects the Open Second Window button first and then the link, Communicator opens the small window specified in the button and then loads doc2.html into it.
On the other hand, if the user selects the link before creating window2 with the button, then Communicator creates window2 with the default parameters and loads doc2.html into that window. If the user later clicks the Open Second Window button, Communicator changes the parameters of the already open window to match those specified in the event handler.

Navigating Among Windows and Frames

Many Navigator windows can be open at the same time. The user can move among these windows by clicking them to make them active, or give them focus. When a window has focus, it moves to the front and changes visually in some way. For example, the color of the window's title bar might change. The visual cue varies depending on which platform you are using.
You can give focus to a window programmatically by giving focus to an object in the window or by specifying the window as the target of a hypertext link. Although you can change an object's values in a second window, that does not make the second window active: the current window remains active.
You navigate among frames the same way as you navigate among windows.

Example 1: give focus to an object in another window. The following statement gives focus to a Text object named city in a window named checkboxWin. Because the Text object is gaining focus, the window also gains focus and becomes active. The example also shows the statement that creates checkboxWin.

```
checkboxWin=window.open("doc2.html")

...

checkboxWin.document.musicForm.city.focus()
```

Example 2: give focus to another window using a hypertext link. The following statement specifies window2 as the target of a hypertext link. When the user clicks the link, focus switches to window2. If window2 does not exist, it is created.

```
<A HREF="doc2.html" TARGET="window2"> Load a file into
window2</A>
```

Additional Topics

- Using JavaScript URLs
- Using Client-Side Image Maps
- Using Server-Side Image Maps
- Using the Status Bar
- Using Cookies
- Determining Installed Plug-ins

Using JavaScript URLs

You are probably familiar with the standard types of URLs: http:, ftp:, file:, and so on. With Navigator, you can also use URLs of type javascript: to execute JavaScript statements instead of loading a document. You simply use a string beginning with javascript: as the value for the HREF attribute of anchor tags. For example, you can define the following hyperlink to reload the current page when the user clicks it:

<A HREF="javascript:history.go(0)">Reload Now</A>
In general, you can put any statements or function calls after the javascript: URL prefix.
You can use JavaScript URLs in many ways to add functionality to your applications. For example, you could increment a counter p1 in a parent frame whenever a user clicks a link, using the following function:

```
function countJumps() {

   parent.p1++

   window.location=page1

}
```

To call the function, use a JavaScript URL in a standard HTML hyperlink:

```
<A HREF="javascript:countJumps()">Page 1</A>
```

This example assumes page1 is a string representing a URL.
If the value of the expression following a javascript: URL prefix evaluates to undefined, no new document is loaded. If the expression evaluates to a defined type, the value is converted to a string that specifies the source of the document to load.

Using Client-Side Image Maps

A client-side image map is defined with the MAP tag. You can define areas within the image that are hyperlinks to distinct URLs; the areas can be rectangles, circles, or polygons.
Instead of standard URLs, you can also use JavaScript URLs in client-side image maps, for example,

```
<MAP NAME="buttonbar">

<AREA SHAPE="RECT" COORDS="0,0,16,14"

   HREF ="javascript:top.close(); window.location = newnav.html">

<AREA SHAPE="RECT" COORDS="0,0,85,46"

   HREF="contents.html" target="javascript:alert(`Loading

   Contents.'); top.location = contents.html">

</MAP>
```

Using Server-Side Image Maps

Client-side image maps provide functionality to perform most tasks, but standard (sometimes called server-side) image maps provide even more flexibility. You specify a standard image map with the ISMAP attribute of an IMG tag that is a hyperlink. For example,

```
<A HREF="img.html"><IMG SRC="about:logo" BORDER=0 ISMAP></A>
```

When you click an image with the ISMAP attribute, Navigator requests a URL of the form
URL?x,y
where URL is the document specified by the value of the HREF attribute, and x and y are the horizontal and vertical coordinates of the mouse pointer (in pixels from the top-left of the image) when you clicked. (The "about:logo" image is built in to Navigator and displays the Netscape logo.)
Traditionally, image-map requests are sent to servers, and a CGI program performs a database lookup function. With client-side JavaScript, however, you can perform the lookup on the client. You can use the search property of the location object to parse the x and y coordinates and perform an action accordingly. For example, suppose you have a file named img.html with the following content:

```
<H1>Click on the image</H1>

<P>

<A HREF="img.html"><IMG SRC="about:logo" BORDER=0 ISMAP></A>

<SCRIPT>

str = location.search

if (str == "")

   document.write("<P>No coordinates specified.")

else {

   commaloc = str.indexOf(",") // the location of the comma

   document.write("<P>The x value is " + str.substring(1,
commaloc))

   document.write("<P>The y value is " +

      str.substring(commaloc+1, str.length))

}

</SCRIPT>
```

When you click a part of the image, Navigator reloads the page (because the HREF attribute specifies the same document), adding the x and y coordinates of the mouse click to the URL. The statements in the else clause then display the x and y coordinates. In practice, you could redirect to another page (by setting location) or perform some other action based on the values of x and y.

Using the Status Bar

You can use two window properties, status and defaultStatus, to display messages in the Navigator status bar at the bottom of the window. Navigator normally uses the status bar to display such messages as "Contacting Host..." and "Document: Done." The defaultStatus message appears when nothing else is in the status bar. The status property displays a transient message in the status bar, such as when the user moves the mouse pointer over a link.
You can set these properties to display custom messages. For example, to display a custom message after the document has finished loading, simply set defaultStatus. For example,
defaultStatus = "Some rise, some fall, some climb...to get to Terrapin"

Creating Hints with onMouseOver and onMouseOut

By default, when you move the mouse pointer over a hyperlink, the status bar displays the destination URL of the link. You can set status in the onMouseOut and onMouseOver event handlers of a hyperlink or image area to display hints in the status bar instead. The event handler must return true to set status. For example,

```
<A HREF="contents.html"

   onMouseOver="window.status='Click to display contents';return
true">

Contents

</A>
```

This example displays the hint "Click to display contents" in the status bar when you move the mouse pointer over the link.

Using Cookies

Netscape cookies are a mechanism for storing persistent data on the client in a file called cookies.txt. Because HyperText Transport Protocol (HTTP) is a stateless protocol, cookies provide a way to maintain information between client requests. This section discusses basic uses of cookies and illustrates with a simple example. For a complete description of cookies, see the Client-Side JavaScript Reference.
Each cookie is a small item of information with an optional expiration date and is added to the cookie file in the following format:
name=value;expires=expDate;
name is the name of the datum being stored, and value is its value. If name and value contain any semicolon, comma, or blank (space) characters, you must use the escape function to encode them and the unescape function to decode them.
expDate is the expiration date, in GMT date format:
Wdy, DD-Mon-YY HH:MM:SS GMT
Although it's slightly different from this format, the date string returned by the Date method toGMTString can be used to set cookie expiration dates.
The expiration date is an optional parameter indicating how long to maintain the cookie. If expDate is not specified, the cookie expires when the user exits the current Navigator

session. Navigator maintains and retrieves a cookie only if its expiration date has not yet passed.
For more information on escape and unescape, see the Client-Side JavaScript Reference.

Limitations

Cookies have these limitations:

- 300 total cookies in the cookie file.
- 4 Kbytes per cookie, for the sum of both the cookie's name and value.
- 20 cookies per server or domain (completely specified hosts and domains are treated as separate entities and have a 20-cookie limitation for each, not combined).

Cookies can be associated with one or more directories. If your files are all in one directory, then you need not worry about this. If your files are in multiple directories, you may need to use an additional path parameter for each cookie. For more information, see the Client-Side JavaScript Reference.

Using Cookies with JavaScript

The document.cookie property is a string that contains all the names and values of Navigator cookies. You can use this property to work with cookies in JavaScript.
Here are some basic things you can do with cookies:

- Set a cookie value, optionally specifying an expiration date.
- Get a cookie value, given the cookie name.

It is convenient to define functions to perform these tasks. Here, for example, is a function that sets cookie values and expiration:

```
// Sets cookie values. Expiration date is optional

//

function setCookie(name, value, expire) {

   document.cookie = name + "=" + escape(value)

   + ((expire == null) ? "" : ("; expires=" +
expire.toGMTString()))

}
Notice the use of escape to encode special characters
(semicolons, commas, spaces) in the value string. This function
assumes that cookie names do not have any special characters.
The following function returns a cookie value, given the name of
the cookie:
function getCookie(Name) {
   var search = Name + "="

   if (document.cookie.length > 0) { // if there are any cookies

      offset = document.cookie.indexOf(search)

      if (offset != -1) { // if cookie exists

         offset += search.length

         // set index of beginning of value

         end = document.cookie.indexOf(";", offset)

         // set index of end of cookie value

         if (end == -1)

            end = document.cookie.length

         return unescape(document.cookie.substring(offset, end))

      }

   }

}
```

Notice the use of unescape to decode special characters in the cookie value.

Using Cookies: an Example

Using the cookie functions defined in the previous section, you can create a simple page users can fill in to "register" when they visit your page. If they return to your page within a year, they will see a personal greeting.
You need to define one additional function in the HEAD of the document. This function, register, creates a cookie with the name TheCoolJavaScriptPage and the value passed to it as an argument.

```
function register(name) {

   var today = new Date()

   var expires = new Date()

   expires.setTime(today.getTime() + 1000*60*60*24*365)

   setCookie("TheCoolJavaScriptPage", name, expires)

}
```

The BODY of the document uses getCookie (defined in the previous section) to check whether the cookie for TheCoolJavaScriptPage exists and displays a greeting if it does. Then there is a form that calls register to add a cookie. The onClick event handler also calls history.go(0) to redraw the page.

```
<BODY>

<H1>Register Your Name with the Cookie-Meister</H1>

<P>

<SCRIPT>

var yourname = getCookie("TheCoolJavaScriptPage")

if (yourname != null)

   document.write("<P>Welcome Back, ", yourname)

else

   document.write("<P>You haven't been here in the last year...")

</SCRIPT>
<P> Enter your name. When you return to this page within a year,
you will be greeted with a personalized greeting.

<BR>

<FORM onSubmit="return false">

Enter your name: <INPUT TYPE="text" NAME="username" SIZE= 10><BR>

<INPUT TYPE="button" value="Register"

   onClick="register(this.form.username.value); history.go(0)">

</FORM>
```

Determining Installed Plug-ins

You can use JavaScript to determine whether a user has installed a particular plug-in; you can then display embedded plug-in data if the plug-in is installed, or display some alternative information (for example, an image or text) if it is not. You can also determine whether a client is capable of handling a particular MIME (Multipart Internet Mail Extension) type. This section introduces the objects and properties needed for handling

plug-ins and MIME types. For more detailed information on these objects and properties, see the Client-Side JavaScript Reference.
The navigator object has two properties for checking installed plug-ins: the mimeTypes array and the plugins array.

mimeTypes Array

mimeTypes is an array of all MIME types supported by the client (either internally, via helper applications, or by plug-ins). Each element of the array is a MimeType object, which has properties for its type, description, file extensions, and enabled plug-ins.
For example, the following table summarizes the values for displaying JPEG images.
Table 13.1 MimeType property values for JPEG images

Expression	**Value**
navigator.mimeTypes["image/jpeg"].type	image/jpeg
navigator.mimeTypes["image/jpeg"].description	JPEG Image
navigator.mimeTypes["image/jpeg"].suffixes	jpeg, jpg, jpe, jfif, pjpeg, pjp
navigator.mimeTypes["image/jpeg"].enabledPlugin	null

The following script checks to see whether the client is capable of displaying QuickTime movies.

```
var myMimetype = navigator.mimeTypes["video/quicktime"]

if (myMimetype)

   document.writeln("Click <A HREF='movie.qt'>here</A> to see a "
+

      myMimetype.description)

else

   document.writeln("Too bad, can't show you any movies.")
```

plugins Array

plugins is an array of all plug-ins currently installed on the client. Each element of the array is a Plugin object, which has properties for its name, file name, and description as well as an array of MimeType objects for the MIME types supported by that plug-in. The user can obtain a list of installed plug-ins by choosing About Plug-ins from the Help menu. For example, the following table summarizes the values for the LiveAudio plug-in.

Plugin property values for the LiveAudio plug-in

Expression	Value
navigator.plugins['LiveAudio'].name	LiveAudio
navigator.plugins['LiveAudio'].description	LiveAudio - Netscape Navigator sound playing component
navigator.plugins['LiveAudio'].filename	d:\nettools\netscape\nav30\ Program\plugins\NPAUDIO.DLL
navigator.plugins['LiveAudio'].length	7

In Table 13.2, the value of the length property indicates that navigator.plugins['LiveAudio'] has an array of MimeType objects containing seven elements. The property values for the second element of this array are as shown in the following table.

MimeType values for the LiveAudio plug-in

Expression	Value
navigator.plugins['LiveAudio'][1].type	audio/x-aiff
navigator.plugins['LiveAudio'][1].description	AIFF
navigator.plugins['LiveAudio'][1].suffixes	aif, aiff
navigator.plugins['LiveAudio'][1].enabledPlugin.name	LiveAudio

The following script checks to see whether the Shockwave plug-in is installed and displays an embedded Shockwave movie if it is:

```
var myPlugin = navigator.plugins["Shockwave"]

if (myPlugin)

   document.writeln("<EMBED SRC='Movie.dir' HEIGHT=100
WIDTH=100>")

else

   document.writeln("You don't have Shockwave installed!")
```

Unicode

Unicode is a universal character-coding standard for the interchange and display of principal written languages. It covers the languages of Americas, Europe, Middle East, Africa, India, Asia, and Pacifica, as well as historic scripts and technical symbols. Unicode allows for the exchange, processing, and display of multilingual texts, as well as the use of common technical and mathematical symbols. It hopes to resolve internationalization problems of multilingual computing, such as different national character standards. Not all modern or archaic scripts, however, are currently supported.
The Unicode character set can be used for all known encoding. Unicode is modeled after the ASCII (American Standard Code for Information Interchange) character set. It uses a numerical value and name for each character. The character encoding specifies the identity of the character and its numeric value (code position), as well as the representation of this value in bits. The 16-bit numeric value (code value) is defined by a hexadecimal number and a prefix U, for example, U+0041 represents A. The unique name for this value is LATIN CAPITAL LETTER A.
JavaScript versions prior to 1.3. Unicode is not supported in versions of JavaScript prior to 1.3.

Unicode Compatibility with ASCII and ISO

Unicode is compatible with ASCII characters and is supported by many programs. The first 128 Unicode characters correspond to the ASCII characters and have the same byte value. The Unicode characters U+0020 through U+007E are equivalent to the ASCII characters 0x20 through 0x7E. Unlike ASCII, which supports the Latin alphabet and uses 7-bit character set, Unicode uses a 16-bit value for each character. It allows for tens of thousands of characters. Unicode version 2.0 contains 38,885 characters. It also supports an extension mechanism, Transformation Format (UTF), named UTF-16, that allows for the encoding of one million more characters by using 16-bit character pairs. UTF turns the encoding to actual bits.
Unicode is fully compatible with the International Standard ISO/IEC 10646-1; 1993, which is a subset of ISO 10646, and supports the ISO UCS-2 (Universal Character Set) that uses two-octets (two bytes or 16 bits).
JavaScript and Navigator support for Unicode means you can use non-Latin, international, and localized characters, plus special technical symbols in JavaScript programs. Unicode provides a standard way to encode multilingual text. Since Unicode is compatible with ASCII, programs can use ASCII characters. You can use non-ASCII Unicode characters in the comments and string literals of JavaScript.

Unicode Escape Sequences

You can use the Unicode escape sequence in string literals. The escape sequence consists of six ASCII characters: \u and a four-digit hexadecimal number. For example, \u00A9 represents the copyright symbol. Every Unicode escape sequence in JavaScript is interpreted as one character.
The following code returns the copyright symbol and the string "Netscape Communications".
```
x="\u00A9 Netscape Communications"
```
The following table lists frequently used special characters and their Unicode value.

Unicode values for special characters

Category	**Unicode value**	**Name**	**Format name**
White space values	\u0009	Tab	<TAB>
	\u000B	Vertical Tab	<VT>
	\u000C	Form Feed	<FF>
	\u0020	Space	<SP>
Line terminator values	\u000A	Line Feed	<LF>
	\u000D	Carriage Return	<CR>
Additional Unicode escape sequence values	\u000b	Backspace	<BS>
	\u0009	Horizontal Tab	<HT>
	\u0022	Double Quote	"
	\u0027	Single Quote	'
	\u005C	Backslash	\

The JavaScript use of the Unicode escape sequence is different from Java. In JavaScript, the escape sequence is never interpreted as a special character first. For example, a line terminator escape sequence inside a string does not terminate the string before it is interpreted by the function. JavaScript ignores any escape sequence if it is used in comments. In Java, if an escape sequence is used in a single comment line, it is interpreted as an Unicode character. For a string literal, the Java compiler interprets the escape sequences first. For example, if a line terminator escape character (\u000A) is used in Java, it terminates the string literal. In Java, this leads to an error, because line terminators are not allowed in string literals. You must use \n for a line feed in a string literal. In JavaScript, the escape sequence works the same way as \n.

Displaying Characters with Unicode

You can use Unicode to display the characters in different languages or technical symbols. For characters to be displayed properly, a client such as Netscape Navigator 4.x needs to support Unicode. Moreover, an appropriate Unicode font must be available to the client, and the client platform must support Unicode. Often, Unicode fonts do not display all the Unicode characters. Some platforms, such as Windows 95, provide a partial support for Unicode.

To receive non-ASCII character input, the client needs to send the input as Unicode. Using a standard enhanced keyboard, the client cannot easily input the additional characters supported by Unicode. Often, the only way to input Unicode characters is by using Unicode escape sequences. The Unicode specification, however, does not require the use of escape sequences. Unicode delineates a method for rendering special Unicode characters using a composite character. It specifies the order of characters that can be used to create a composite character, where the base character comes first, followed by one or more non-spacing marks. Common implementations of Unicode, including the JavaScript implementation, however, do not support this option. JavaScript does not attempt the representation of the Unicode combining sequences. In other words, an input of a and ' does not produce à. JavaScript interprets a' as two distinct 16-bit Unicode characters. You must use a Unicode escape sequence or a literal Unicode character for à.