Core JavaScript Guide

Download [HTML ZIP]

Index **Next**

Last Updated **September 28, 2000**

Symbols A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

zero-fill right shift (>>>) operator 1, 2

Previous **Contents**

Last Updated **September 28, 2000**

About this Book

JavaScript is Netscape's cross-platform, object-based scripting language. This book explains everything you need to know about using core JavaScript.

This preface contains the following sections:

New Features in this Release

For a summary of JavaScript 1.5 features, see "New Features in this Release" on page 18. 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 4.06-4.7x
JavaScript 1.4	
JavaScript 1.5	Navigator 6.0 Mozilla (open source browser)

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 Abbreviations of 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 core JavaScript documentation includes the following books:

-
- The *Core JavaScript Guide* (this book) provides information about the core JavaScript language and its objects.
- The *Core JavaScript Reference* provides reference material for the core JavaScript language.

If you are new to JavaScript, start with the *Core JavaScript Guide*. Once you have a firm grasp of the fundamentals, you can use the *Core JavaScript Reference* to get more details on individual objects and statements.

Document Conventions

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.

Last Updated **September 28, 2000**

Chapter 1 JavaScript Overview

This chapter introduces JavaScript, discusses some of its fundamental concepts, and describes the new features in the 1.5 release.

This chapter contains the following sections:

What Is JavaScript?

JavaScript is Netscape's cross-platform, object-oriented scripting language. JavaScript is a small, lightweight language; it is not useful as a standalone language, but is designed for easy embedding in other products and applications, such as web browsers. Inside a host environment, JavaScript can be connected to the objects of its environment to provide programmatic control over them.

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.

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.

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
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.
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."

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 the mozilla Web site. You can also find the specification on the ECMA Web site. The ECMA specification does not describe the Document Object Model (DOM), which is 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, Edition 1 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, Edition 1, 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, Edition 1. 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.

JavaScript 1.4	JavaScript 1.4 is fully compatible with ECMA-262, Edition 1. The third version of the ECMA specification was not finalized when JavaScript 1.4 was released.
JavaScript 1.5	JavaScript 1.5 is fully compatible with ECMA-262, Edition 3.

Note: ECMA-262, Edition 2 consisted of minor editorial changes and bug fixes to the Edition 1 specification. The TC39 working group of ECMA is currently working on ECMAScript Edition 4, which will correspond to a future release of JavaScript, JavaScript 2.0.

The *Core 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).

New Features in this Release

JavaScript version 1.5 provides the following new features and enhancements:

-
- **Runtime errors.** Runtime errors are now reported as exceptions.
- **Number formatting enhancements.** Number formatting has been enhanced to include `Number.prototype.toExponential`, `Number.protoytpe.toFixed` and `Number.prototype.toPrecision` methods. See page 109.
- **Regular expression enhancements**. The following regular expression enhancements have been added:
  -
  - Quantifiers — `+`, `*`, `?` and `{}`— can now be followed by a `?` to force them to be non-greedy. See the entry for `?` on page 56.
  - Non-capturing parentheses, `(?:x)` can be used instead of capturing parentheses `(x)`. When non-capturing parentheses are used, matched subexpressions are not available as back-references. See the entry for `(?:x)` on page 56.
  - Positive and negative lookahead assertions are supported. Both assert a match depending on what follows the string being matched. See the entries for `x(?=y)` and `x(?!y)` on page 56.
  - The `m` flag has been added to specify that the regular expression should match over multiple lines. See page 63.
- **Conditional function declarations.** Functions can now be declared inside an `if` clause. See page 84.

- Function expressions. Functions can now be declared inside an expression. See page 85.

- **Multiple `catch` clauses.** Multiple `catch` clauses in a `try...catch` statement are supported. See page 80.

- **Getters and Setters.** JavaScript writers can now add getters and setters to their objects. This feature is available only in the C implementation of JavaScript. See page 98.

- **Constants.** Read only named constants are supported. This feature is available only in the C implementation of JavaScript. See page 27.

Last Updated **September 28, 2000**

Part 1 Core Language Features

Chapter 7 Working with Objects

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

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.

Last Updated **September 28, 2000**

Chapter 2 Values, Variables, and Literals

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

This chapter contains the following sections:

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. The names of variables, called *identifiers*, conform to certain rules.

A JavaScript identifier must start with a letter, underscore (_), or dollar sign ($); 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).

Starting with JavaScript 1.5, you can use ISO 8859-1 or Unicode letters such as å and ü in identifiers. You can also use the \uXXXX Unicode escape sequences listed on page 34 as characters in identifiers.

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`.

Constants

You can create a read-only, named constant with the `const` keyword. The syntax of a constant identifier is the same as for a variable identifier: it must start with a letter or underscore and can contain alphabetic, numeric, or underscore characters.

```
const prefix = '212';
```

A constant cannot change value through assignment or be re-declared while the script is running.

The scope rules for constants are the same as those for variables, except that the `const` keyword is always required, even for global constants. If the keyword is omitted, the identifier is assumed to represent a `var`.

You cannot declare a constant at the same scope as a function or variable with the same name as the function or variable. For example:

```
//THIS WILL CAUSE AN ERROR
function f{};
const f = 5;

//THIS WILL CAUSE AN ERROR ALSO
function f{
const g=5;
var g;

//statements

}
```

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 93.

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 100 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]` and `myList[2]` are missing.

```
myList = [ , 'home', , 'school'];
```

In the following example, the length of the array is four, and `myList[1]`and `myList[3]` are 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 103 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.

Octal integer literals are deprecated and have been removed from the ECMA-262, Edition 3 standard. JavaScript 1.5 still supports them for backward compatibility.

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 or not behave as you expect, because the { will be interpreted as the beginning of a block.

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 (`CarTypes("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 a numeric or string literal for the name of a property or nest an object inside another. The following example uses these options.

```
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 110 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
`\v`	Vertical tab

`\'`	Apostrophe or single quote
`\"`	Double quote
`\\`	Backslash character (\).
`\`*XXX*	The character with the Latin-1 encoding specified by up to three octal digits *XXX* between 0 and 377. For example, \251 is the octal sequence for the copyright symbol.
`\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.

Escaping Characters

For characters not listed in Table 2.1, a preceding backslash is ignored, but this usage is deprecated and should be avoided.

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 a 7-bit character set, Unicode uses a 16-bit value for each character. It allows for tens of thousands of characters. It also supports an extension mechanism, 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, string literals, identifiers, and regular expressions of JavaScript.

Unicode Escape Sequences

You can use the Unicode escape sequence in string literals, regular expressions, and identifiers. 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	\u0008	Backspace	<BS>
	\u0009	Horizontal Tab	<HT>
	\u0022	Double Quote	"
	\u0027	Single Quote	'
	\u005C	Backslash	\

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

Displaying Characters with Unicode

You can use Unicode to display the characters in different languages or technical symbols. For characters to be displayed properly, a client such as Netscape Navigator 4.x or Netscape 6 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. Sometimes, the only way to input Unicode characters is by using Unicode escape sequences.

For more information on Unicode, see the Unicode Consortium Web site and The

Unicode Standard, Version 2.0, published by Addison-Wesley, 1996.

Last Updated **September 28, 2000**

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

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
- Object: evaluates to an object

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, string, logical, or object 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[1]

Equal (==)	Returns true if the operands are equal. If the two operands are not of the same type, JavaScript attempts to convert the operands to an appropriate type for the comparison.	`3 == var1`
		`"3" == var1`
		`3 == '3'`

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

[1] These examples assume that `var1` has been assigned the value 3 and `var2` has been assigned the value 4.

Arithmetic Operators

Arithmetic operators take numerical values (either literals or variables) as their operands and return a single numerical value. The standard arithmetic operators are addition (+), subtraction (-), multiplication (*), and division (/). These operators work as they do in most other programming languages, except the / operator returns a floating-point division in JavaScript, not a truncated division as it does in languages such as C or Java. For example:

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

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

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

- in
- instanceof
- 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*10 +j])
```

Note that two-dimensional arrays are not yet supported. This example emulates a two-dimensional array using a one-dimensional array.

delete

The delete operator deletes an object, an object's property, or an element at a specified index in an array. The 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 if declared
implicitly)
```

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
}
```

in

The `in` operator returns true if the specified property is in the specified object. The syntax is:

`propNameOrNumber` `in` *`objectName`*

where `propNameOrNumber` is a string or numeric expression representing a property name or array index, and `objectName` is the name of an object.

The following examples show some uses of the `in` operator.

```
// Arrays
trees=new Array("redwood","bay","cedar","oak","maple")
0 in trees        // returns true
3 in trees        // returns true
6 in trees        // returns false
"bay" in trees    // returns false (you must specify the index
number,
                  // not the value at that index)
```

```
"length" in trees // returns true (length is an Array property)

// Predefined objects
"PI" in Math          // returns true
myString=new String("coral")
"length" in myString  // returns true

// Custom objects
mycar = {make:"Honda",model:"Accord",year:1998}
"make" in mycar  // returns true
"model" in mycar // returns true
```

instanceof

The `instanceof` operator returns true if the specified object is of the specified object type. The syntax is:

`objectName` `instanceof` *`objectType`*

where `objectName` is the name of the object to compare to `objectType`, and `objectType` is an object type, such as Date or Array.

Use `instanceof` when you need to confirm the type of an object at runtime. For example, when catching exceptions, you can branch to different exception-handling code depending on the type of exception thrown.

For example, the following code uses `instanceof` to determine whether `theDay` is a `Date` object. Because `theDay` is a `Date` object, the statements in the `if` statement execute.

```
theDay=new Date(1995, 12, 17)
if (theDay instanceof Date) {
   // statements to execute
}
```

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 93.

See new in the *Core 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 function
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 undefined, which 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	`< <= > >= in instanceof`
bitwise shift	`<< >> >>>`
addition/subtraction	`+ -`
multiply/divide	`* / %`
negation/increment	`! ~ - + ++ -- typeof void delete`
call / create instance	`() new`
member	`. []`

Last Updated **September 28, 2000**

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

You construct a regular expression in one of two ways:

- Using a regular expression literal, as follows:

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

  Regular expression literals provide compilation of the regular expression when the script is evaluated. When the regular expression will remain constant, use this for better performance.

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

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

  Using the constructor function provides runtime compilation of the regular expression. Use the constructor function when you know the regular expression pattern will be changing, or you don't know the pattern and are getting it from another source, such as user input.

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.

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 item) 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 item 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. If the multiline flag is set to true, also matches immediately after a line break character. For example, `/^A/` does not match the 'A' in "an A", but does match the first 'A' in "An A".
$	Matches end of input. If the multiline flag is set to true, also matches immediately before a line break character. For example, `/t$/` does not match the 't' in "eater", but does match it in "eat".
*	Matches the preceding character 0 or more times. For example, `/bo*/` matches 'boooo' in "A ghost booooed" and 'b' in "A bird warbled", but nothing in "A goat grunted".
+	Matches the preceding character 1 or more times. Equivalent to `{1,}`. For example, `/a+/` matches the 'a' in "candy" and all the a's in "caaaaaaandy".

?	Matches the preceding character 0 or 1 time. For example, `/e?le?/` matches the 'el' in "angel" and the 'le' in "angle." If used immediately after any of the quantifiers `*`, `+`, `?`, or `{}`, makes the quantifier non-greedy (matching the minimum number of times), as opposed to the default, which is greedy (matching the maximum number of times). Also used in lookahead assertions, described under x(?=y) and x(?!y) in this table.
.	(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. These are called capturing parentheses. For example, `/(foo)/` matches and remembers 'foo' in "foo bar." The matched substring can be recalled from the resulting array's elements `[1]`, ..., `[n]`.
`(?:x)`	Matches 'x' but does not remember the match. These are called non-capturing parentheses. The matched substring can not be recalled from the resulting array's elements `[1]`, ..., `[n]`.
`x(?=y)`	Matches 'x' only if 'x' is followed by 'y'. For example, `/Jack(?=Sprat)/` matches 'Jack' only if it is followed by 'Sprat'. `/Jack(?=Sprat|Frost)/` matches 'Jack' only if it is followed by 'Sprat' or 'Frost'. However, neither 'Sprat' nor 'Frost' is part of the match results.
`x(?!y)`	Matches 'x' only if 'x' is not followed by 'y'. For example, `/\d+(?!\.)/` matches a number only if it is not followed by a decimal point. The regular expression `/\d+(?!\.)/.exec("3.141")` matches 141 but not 3.141.

`x|y` Matches either 'x' or 'y'.

For example, `/green|red/` matches 'green' in "green apple" and 'red' in "red apple."

`{n}` Where n is a positive integer. Matches exactly n occurrences of the preceding character.

For example, `/a{2}/` doesn't match the 'a' in "candy," but it matches all of the a's in "caandy," and the first two a's in "caaandy."

`{n,}` Where n is a positive integer. Matches at least n occurrences of the preceding character.

For example, `/a{2,}` doesn't match the 'a' in "candy", but matches all of the a's in "caandy" and in "caaaaaaandy."

`{n,m}` Where n and `m` are positive integers. Matches at least n and at most `m` occurrences of the preceding character.

For example, `/a{1,3}/` matches nothing in "cndy", the 'a' in "candy," the first two a's in "caandy," and the first three a's in "caaaaaaandy" Notice that when matching "caaaaaaandy", the match is "aaa", even though the original string had more a's in it.

`[xyz]` A character set. Matches any one of the enclosed characters. You can specify a range of characters by using a hyphen.

For example, `[abcd]` is the same as `[a-d]`. They match the 'b' in "brisket" and the 'c' in "ache".

`[^xyz]` A negated or complemented character set. That is, it matches anything that is not enclosed in the brackets. You can specify a range of characters by using a hyphen.

For example, `[^abc]` is the same as `[^a-c]`. They initially match 'r' in "brisket" and 'h' in "chop."

`[\b]` Matches a backspace. (Not to be confused with `\b`.)

`\b` Matches a word boundary, such as a space or a newline character. (Not to be confused with `[\b]`.)

For example, `/\bn\w/` matches the 'no' in "noonday";`/\wy\b/` matches the 'ly' in "possibly yesterday."

`\B` Matches a non-word boundary.

For example, `/\w\Bn/` matches 'on' in "noonday", and `/y\B\w/` matches 'ye' in "possibly yesterday."

`\c`*X* Where *X* is a control character. Matches a control character in a string.

For example, `/\cM/` matches control-M in a string.

`\d` Matches a digit character. Equivalent to `[0-9]`.

For example, `/\d/` or `/[0-9]/` matches '2' in "B2 is the suite number."

`\D` Matches any non-digit character. Equivalent to `[^0-9]`.

For example, `/\D/` or `/[^0-9]/` matches 'B' in "B2 is the suite number."

`\f` Matches a form-feed.

`\n` Matches a linefeed.

`\r` Matches a carriage return.

`\s` Matches a single white space character, including space, tab, form feed, line feed. Equivalent to `[ \f\n\r\t\v\u00A0\u2028\u2029]`.

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\u00A0\u2028\u2029]`.

For example, `/\S\w*/` matches 'foo' in "foo bar."

`\t` Matches a tab.

`\v` Matches a vertical tab.

`\w` Matches any alphanumeric character including the underscore. Equivalent to `[A-Za-z0-9_]`.

For example, `/\w/` matches 'a' in "apple," '5' in "$5.28," and '3' in "3D."

`\W` Matches any non-word character. Equivalent to `[^A-Za-z0-9_]`.

For example, `/\W/` or `/[^$A-Za-z0-9_]/` matches '%' in "50%."

`\n` Where *n* is a positive integer. A back reference to the last substring matching the *n* parenthetical in the regular expression (counting left parentheses).

For example, `/apple(,)\sorange\1/` matches 'apple, orange,' in "apple, orange, cherry, peach."

`\0` Matches a NUL character. Do not follow this with another digit.

`\xhh`	Matches the character with the code hh (two hexadecimal digits)
`\uhhhh`	Matches the character with code hhhh (four hexadecimal digits).

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.

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'.

To match a substring without causing the matched part to be remembered, within the parentheses preface the pattern with `?:`. For example, `(?:\d+)` matches one or numeric characters but does not remember the matched characters.

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 *Core 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 construct the regular expression from a string, 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, Ignoring Case, and Considering Multiline Input.)	5
	source	The text of the pattern.Updated at the time that the regular expression is created, not executed.	"d(b+)d"

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 `Array` elements `[1]`, ..., `[n]`.

The number of possible parenthesized substrings is unlimited. 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 `$1` and `$2` in the replacement to denote the first string and second parenthesized substring match.

```
<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, called using the () shortcut notation, uses the value of `RegExp.input` as its argument.

```
<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, Ignoring Case, and Considering Multiline Input

Regular expressions have three 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. To indicate a multi-line search, use the `m` flag. These flags can be used separately or together in any order, and are included as part of the regular expression.

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

```
re = /pattern/flags
re = new RegExp("pattern", ["flags"])
```

Note that the flags 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.

The `m` flag is used to specify that a multiline input string should be treated as multiple

lines. If the `m` flag is used, `^` and `$` match at the start or end of any line within the input string instead of the start or end of the entire string.

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:

-
- Block 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
- Exception Handling Statements: `try...catch` and `throw`

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 *Core JavaScript Reference* for details about the statements in this chapter.

Block Statement

A block statement is used to group statements. The block is delimited by a pair of curly brackets:

```
{statement1 statement2 . . .statementn}
```

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, {}.

You should not use simple assignments in a conditional statement. For example, do not use the following code:

```
if(x = y)
{
   /* do the right thing */
}
```

If you need to use an assignment in a conditional statement, put additional parentheses around the assignment. For example, use `if( (x = y) )`.

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, null, zero, NaN, or the empty string, 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`, and `while` loop statements, as well as label (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.

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.
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. This expression can also declare variables.

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. If the `condition` expression is omitted entirely, the condition is assumed to be true.

3. The `statements` execute.

4. The update expression `incrementExpression`, if there is one, 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` is true, the statement executes again. At the end of every execution, the condition is checked. When the condition is 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 statement.

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` without a label, it terminates the innermost enclosing `while`, `do-while`, `for`, or `switch` immediately and transfers control to the following statement.

- When you use `break` with a label, it terminates the specified labeled statement.

The syntax of the `break` statement looks like this:

```
1. break
2. break label
```

The first form of the syntax terminates the innermost enclosing loop or `switch`; 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.

-
- When you use `continue` without a label, it terminates the current iteration of the innermost enclosing `while`, `do-while` or `for` statement 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`.

- When you use `continue` with a label, it applies to the looping statement identified with that label.

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);
}
```

Note: Using a `with` statement can significantly slow down your program.

Comments

Comments are author notations that explain what a script does. Comments are ignored by the interpreter. JavaScript supports Java and C++-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. */
```

Exception Handling Statements

You can throw exceptions using the `throw` statement and handle them using the `try...catch` statements.

You also use the `try...catch` statement to handle Java exceptions. See "Handling Java Exceptions in JavaScript" on page 141 and "Handling JavaScript Exceptions in Java" on page 144 for information.

The throw Statement

Use the `throw` statement to throw an exception. When you throw an exception, you specify an expression containing the value of the exception:

```
throw expression
```

The following code throws several exceptions.

```
throw "Error2";    // generates an exception with a string value
```

```
throw 42;          // generates an exception with the value 42
throw true;        // generates an exception with the value true
```

You can specify an object when you throw an exception. You can then reference the object's properties in the `catch` block. The following example creates an object `myUserException` of type `UserException` and uses it in a `throw` statement.

```
// Create an object type UserException
function UserException (message) {
   this.message=message;
   this.name="UserException";
}
// Create an instance of the object type and throw it
myUserException=new UserException("Value too high");
throw myUserException;
```

The try...catch Statement

The `try...catch` statement marks a block of statements to try, and specifies one or more responses should an exception be thrown. If an exception is thrown, the `try...catch` statement catches it.

The `try...catch` statement consists of a `try` block, which contains one or more statements, and zero or more `catch` blocks, containing statements that specify what to do if an exception is thrown in the `try` block. That is, you want the `try` block to succeed, and if it does not succeed, you want control to pass to the `catch` block. If any statement within the `try` block (or in a function called from within the `try` block) throws an exception, control immediately shifts to the `catch` block. If no exception is thrown in the `try` block succeed, the `catch` block is skipped. The `finally` block executes after the `try` and `catch` blocks execute but before the statements following the `try...catch` statement.

The following example uses a `try...catch` statement. The example calls a function that retrieves a month name from an array based on the value passed to the function. If the value does not correspond to a month number (1-12), an exception is thrown with the value "InvalidMonthNo" and the statements in the `catch` block set the `monthName` variable to "unknown".

```
function getMonthName (mo) {
   mo=mo-1; // Adjust month number for array index (1=Jan, 12=Dec)
   var months=new Array("Jan","Feb","Mar","Apr","May","Jun","Jul",
      "Aug","Sep","Oct","Nov","Dec");
   if (months[mo] != null) {
      return months[mo]
```

```
   } else {
      throw "InvalidMonthNo"
   }
}

try {
   // statements to try
   monthName=getMonthName(myMonth) // function could throw
exception
}
catch (e) {
   monthName="unknown"
      logMyErrors(e) // pass exception object to error handler
}
```

The catch Block

You can use a single `catch` block to handle all exceptions that may be generated in the `try` block, or you can use separate `catch` blocks each of which handles a particular type of exception.

Single catch Block

Use a single `try...catch` statement's `catch` block (*recovery block*) to execute error-handling code for any exceptions thrown in the `try` block.

A single `catch` block has the following syntax:

```
catch (catchID) {
   statements
}
```

The `catch` block specifies an identifier (`catchID` in the preceding syntax) that holds the value specified by the `throw` statement; you can use this identifier to get information about the exception that was thrown. JavaScript creates this identifier when the `catch` block is entered; the identifier lasts only for the duration of the `catch` block; after the `catch` block finishes executing, the identifier is no longer available.

For example, the following code throws an exception. When the exception occurs, control transfers to the `catch` block.

```
try {
   throw "myException" // generates an exception
}
catch (e) {
   // statements to handle any exceptions
```

```
   logMyErrors(e) // pass exception object to error handler
}
```

Multiple catch Blocks

A single `try` statement can contain multiple conditional catch blocks, each of which handles a specific type of exception. In this case, the appropriate conditional `catch` block is entered only when the exception specified for that block is thrown. You can also include an optional catch-all `catch` block for all unspecified exceptions as the final `catch` block in the statement.

For example, the following function invokes three other functions (declared elsewhere), which validate its arguments. If a validation function determines that the component that it is checking is invalid, it returns 0, causing the caller to throw a particular exception.

```
function getCustInfo(name, id, email)
{
   var n, i, e;

   if (!validate_name(name))
      throw "InvalidNameException"
   else
      n = name;
   if (!validate_id(id))
      throw "InvalidIdException"
   else
      i = id;
   if (!validate_email(email))
      throw "InvalidEmailException"
   else
      e = email;
   cust = (n + " " + i + " " + e);
   return (cust);
}
```

The conditional catch blocks route control to the appropriate exception handler.

```
try {

// function could throw three exceptions
   getCustInfo("Lee", 1234, "lee@netscape.com")
}

catch (e if e == "InvalidNameException") {
// call handler for invalid names
```

```
bad_name_handler(e)
}

catch (e if e == "InvalidIdException") {
// call handler for invalid ids
bad_id_handler(e)
}

catch (e if e == "InvalidEmailException") {
// call handler for invalid email addresses
bad_email_handler(e)
}

catch (e){
// don't know what to do, but log it
logError(e)
}
```

The finally Block

The `finally` block contains statements to execute after the `try` and `catch` blocks execute but before the statements following the `try...catch` statement. The `finally` block executes whether or not an exception is thrown. If an exception is thrown, the statements in the `finally` block execute even if no `catch` block handles the exception.

You can use the `finally` block to make your script fail gracefully when an exception occurs; for example, you may need to release a resource that your script has tied up. The following example opens a file and then executes statements that use the file (server-side JavaScript allows you to access files). If an exception is thrown while the file is open, the `finally` block closes the file before the script fails.

```
openMyFile();
try {
   writeMyFile(theData)
}
finally {
   closeMyFile() // always close the resource
}
```

Nesting try...catch Statements

You can nest one or more `try...catch` statements. If an inner `try...catch` statement does not have a `catch` block, the enclosing `try...catch` statement's

`catch` block is checked for a match.

Last Updated **September 28, 2000**

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

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.

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)
```

A function can be defined based on a condition. For example, given the following function definition:

```
if (num == 0)
{
   function myFunc(theObject) {
   theObject.make="Toyota"
   }
}
```

the `myFunc` function is only defined if the variable `num` equals 0. If `num` does not equal 0, the function is not defined, and any attempt to execute it will fail.

In addition to defining functions as described here, you can also define `Function` objects, as described in "Function Object" on page 106.

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

A function can also be defined inside an expression. This is called a function expression. Typically such a function is anonymous; it does not have to have a name. For example, the function square could have been defined as:

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

This is convenient when passing a function as an argument to another function. The following example shows the map function being defined and then called with an anonymous function as its first parameter:

```
function map(f,a) {
   var result=new Array;
   for (var i = 0; i != a.length; i++)
      result[i] = f(a[i]);
   return result;
}
```

The call

```
map(function(x) {return x * x * x}, [0, 1, 2, 5, 10];
```

returns [0, 1, 8, 125, 1000].

Calling Functions

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 91) 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 {
      var 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 arguments passed to it as follows:

`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) {
   var 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 *Core JavaScript Reference* for more information.

JavaScript 1.3 and earlier versions. The `arguments` array is a property of the `Function` object and can be preceded by the function name, as follows:

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

Predefined Functions

JavaScript has several top-level predefined functions:

-
- `eval`
- `isFinite`
- `isNaN`

- `parseInt` and `parseFloat`
- `Number` and `String`
- `encodeURI, decodeURI, encodeURIComponent, and decodeURIComponent` (all available with Javascript 1.5 and later).

The following sections introduce these functions. See the *Core 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.

The `escape` and `unescape` functions do not work properly for non-ASCII characters and have been deprecated. In JavaScript 1.5 and later, use `encodeURI`, `decodeURI`, `encodeURIComponent`, and `decodeURIComponent`.

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

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

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 and later, 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 `property` *`I`* is an identifier (either a name, a number, or a string literal), and each `value` *`I`* is an expression whose value is assigned to the `property` *`I`*. 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 28.

JavaScript 1.1 and earlier. 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.

Using a Constructor Function

Alternatively, you can create an object with these two steps:

1.
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 *Core 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

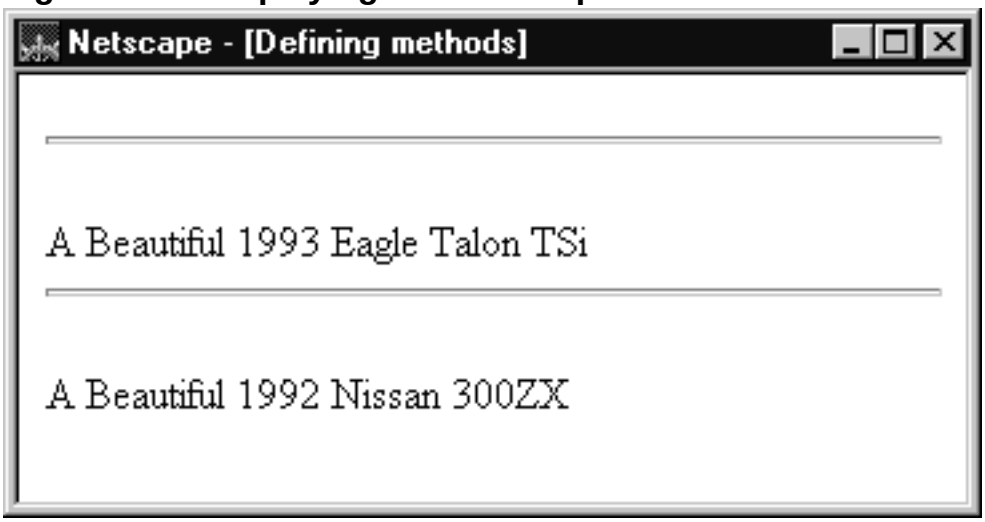
```
function car(make, model, year, owner) {
   this.make = make;
   this.model = model;
   this.year = year;
   this.owner = owner;
   this.displayCar = displayCar;
}
```

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>
```

Defining Getters and Setters

A getter is a method that gets the value of a specific property. A setter is a method that sets the value of a specific property. You can define getters and setters on any predefined core object or user-defined object that supports the addition of new properties. The syntax for defining getters and setters uses the object literal syntax.

The following JS shell session illustrates how getters and setters could work for a user-defined object o. The JS shell is an application that allows developers to test JavaScript code in batch mode or interactively.

The o object's properties are:

-
- o.a - a number

- `o.b` - a getter that returns `o.a` plus 1
- `o.c` - a setter that sets the value of `o.a` to half of its value

```
js> o = new Object;
[object Object]
js> o = {a:7, get b() {return this.a+1; }, set c(x) {this.a
= x/2}};
[object Object]
js> o.a
7
js> o.b
8
js> o.c = 50
js> o.a
25
js>
```

This JavaScript shell session illustrates how getters and setters can extend the `Date` prototype to add a year property to all instances of the predefined `Date` class. It uses the `Date` class's existing `getFullYear` and `setFullYear` methods to support the `year` property's getter and setter.

These statements define a getter and setter for the `year` property.:

```
js> var d = Date.prototype;
js> d.year getter= function() { return this.getFullYear(); };

js> d.year setter= function(y) { return this.setFullYear(y); };
```

These statements use the getter and setter in a Date object:

```
js> var now = new Date;
js> print(now.year);
2000
js> now.year=2001;
987617605170
js> print(now);
Wed Apr 18 11:13:25 GMT-0700 (Pacific Daylight Time) 2001
```

Deleting Properties

You can remove a property by using the `delete` operator. The following code shows how to remove a property.

```
//Creates a new property, myobj, with two properties, a and b.
myobj = new Object;
myobj.a=5;
myobj.b=12;

//Removes the a property, leaving myobj with only the b property.
delete myobj.a;
```

You can also use delete to delete a global variable if the **var** keyword was not used to declare the variable:

```
g = 17;
delete g;
```

See "delete" on page 46 for more information.

Predefined Core Objects

This section describes the predefined objects in core JavaScript: `Array`, `Boolean`, `Date`, `Function`, `Math`, `Number`, `RegExp`, and `String`.

Array Object

JavaScript does not have an explicit array data type. However, you can use the predefined `Array` object and its methods to work with arrays in your applications. The `Array` object has methods for manipulating arrays in various ways, such as joining, reversing, 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 28 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+"]"
   }
}
```

This example creates an array with the following rows:

```
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.split`. 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` , `null`, `0`, `NaN`, or the empty string , including a `Boolean` object whose value is false, evaluates to true when passed to a conditional statement. See "if...else Statement" on page 68 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. 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 1995.

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)
   if (hour == 0)
      temp = "12";
   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` - 12), otherwise simply `hour`, unless hour is 0, in which case it becomes 12.

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 and the function expression. See the *Core 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'
}
```

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

-
- When you assign a function to a variable using `var setBGColor = new Function("...")`, `setBGColor` is a variable for which the current value is a reference to the function created with `new Function()`.

- When you create a function using `function setBGColor() {...}`, `setBGColor` is not a variable, it is the name of a function.

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, atan2`	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
`random`	Returns a random number between 0 and 1.
`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.

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

Property	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

The Number prototype provides methods for retrieving information from Number objects

in various formats. The following table summarizes the methods of `Number.prototype`.

Table 7.3 Methods of Number.prototype

Method	Description
`toExponential`	Returns a string representing the number in exponential notation.
`toFixed`	Returns a string representing the number in fixed-point notation.
`toPrecision`	Returns a string representing the number to a specified precision in fixed-point notation.
`toSource`	Returns an object literal representing the specified Number object; you can use this value to create a new object. Overrides the `Object.toSource` method.
`toString`	Returns a string representing the specified object. Overrides the `Object.toString` method.
`valueOf`	Returns the primitive value of the specified object. Overrides the `Object.valueOf` method.

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 *Core 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 7.4 Methods of String Instances

Method	Description
`anchor`	Creates HTML named anchor.
`big, blink, bold, fixed, italics, small, strike, sub, sup`	Create HTML formatted string.
`charAt, charCodeAt`	Return the character or character code at the specified position in string.
`indexOf, lastIndexOf`	Return 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 Unicode values. This is a method of the String class, not a String instance.
`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`	Return the specified subset of the string, either by specifying the start and end indexes or the start index and a length.
`match, replace, search`	Work with regular expressions.
`toLowerCase, toUpperCase`	Return the string in all lowercase or all uppercase, respectively.

Last Updated **September 28, 2000**

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

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

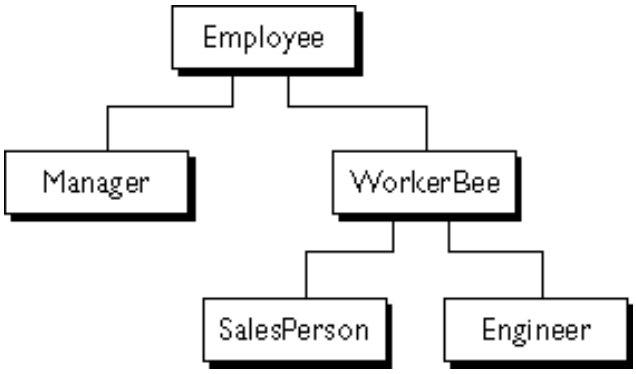
Class-based (Java)	Prototype-based (JavaScript)

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".

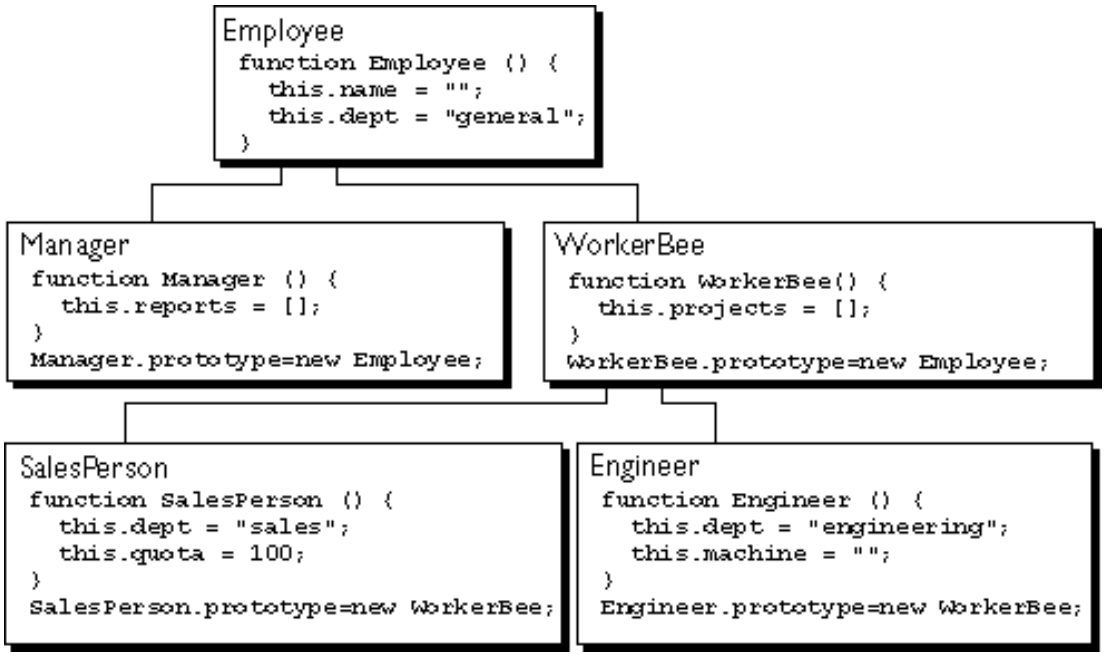
Creating the Hierarchy

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 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.

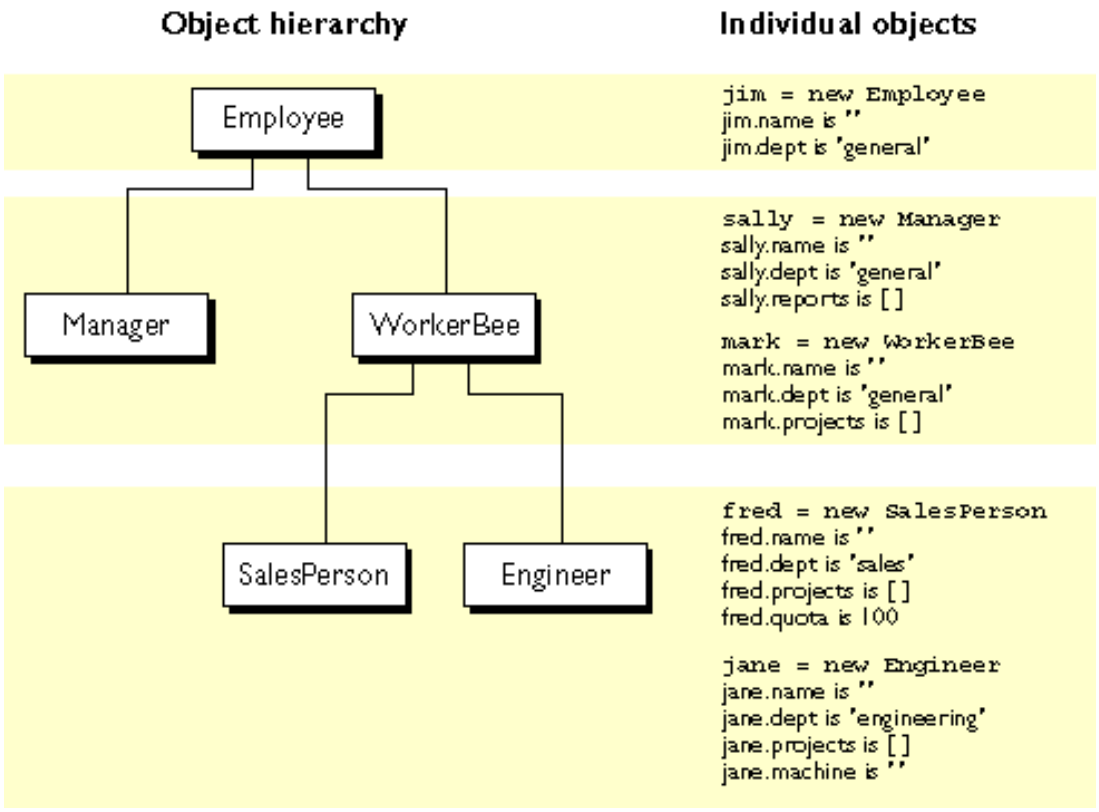
JavaScript	Java

`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.

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;
```

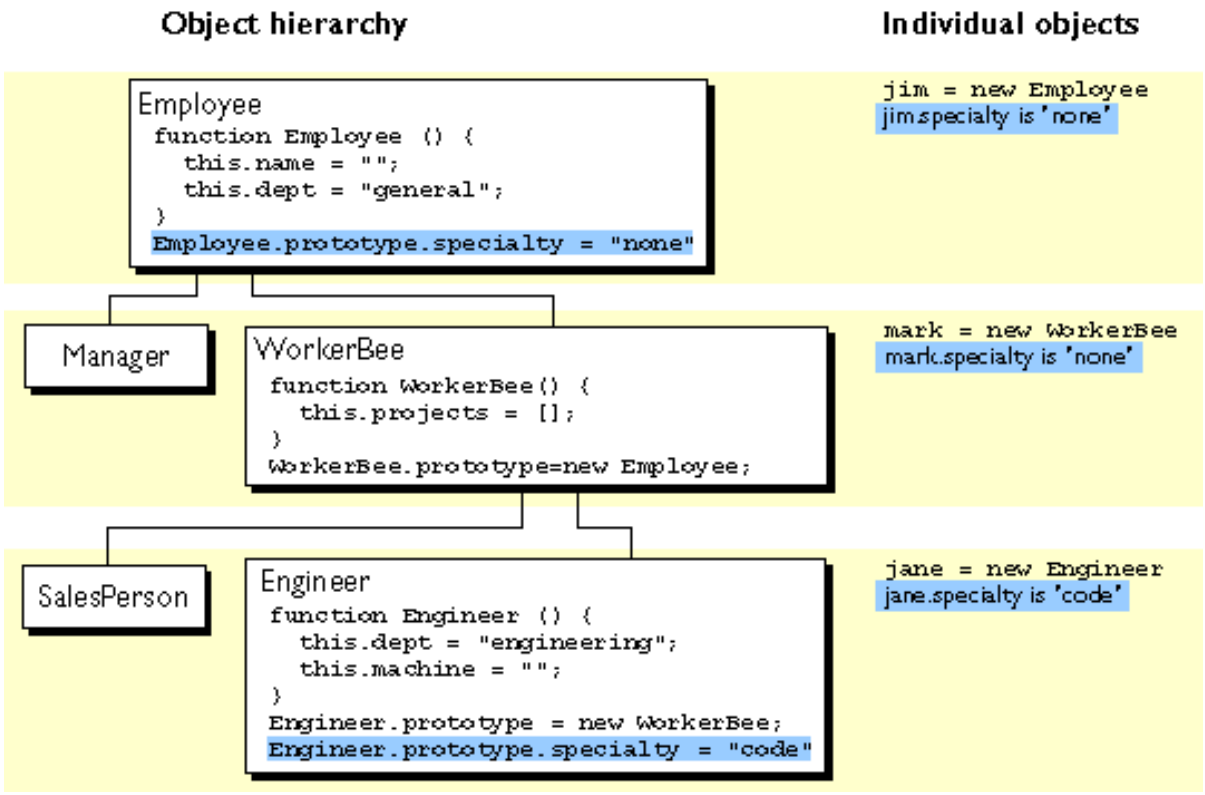
Now, the `mark` object has a `bonus` property, but no other `WorkerBee` has this property.

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.

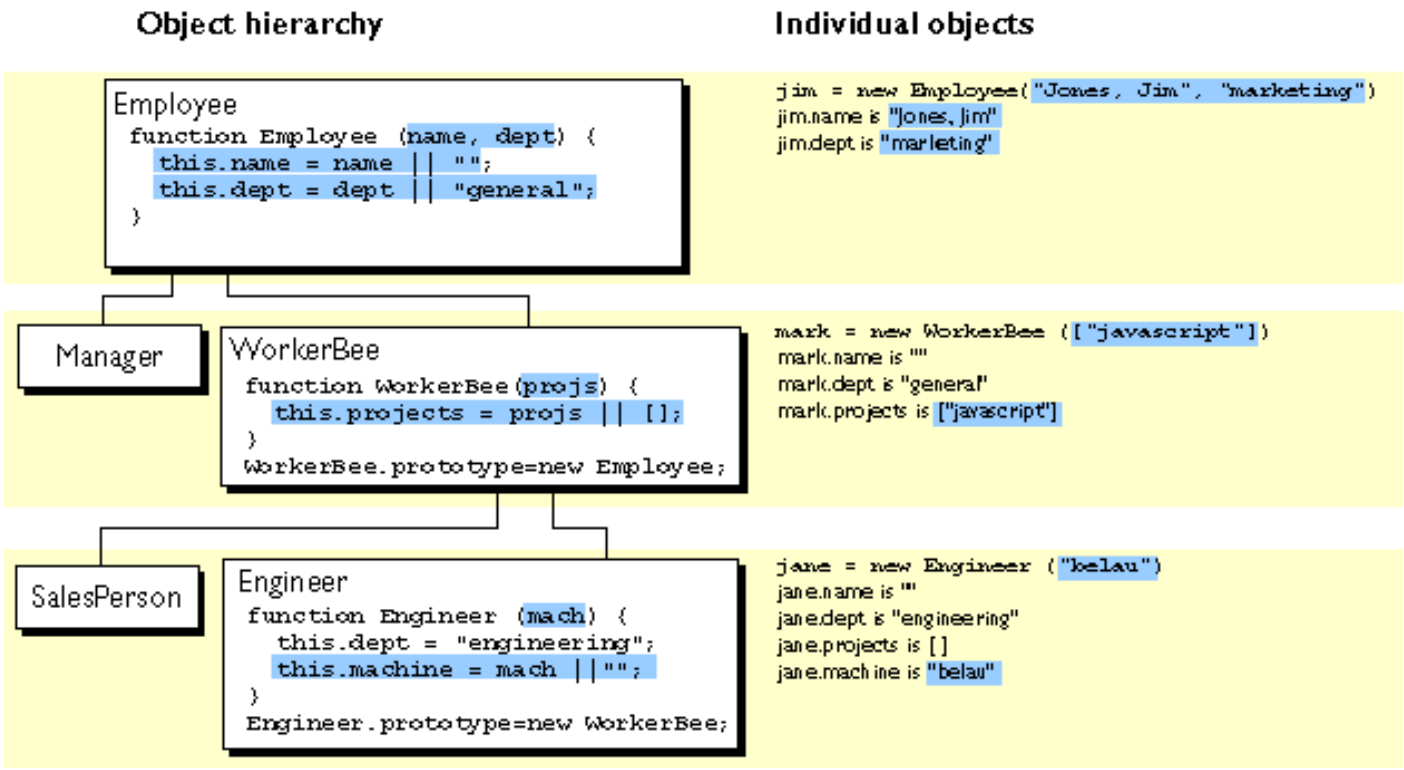
Figure 8.4 Adding properties



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");
```

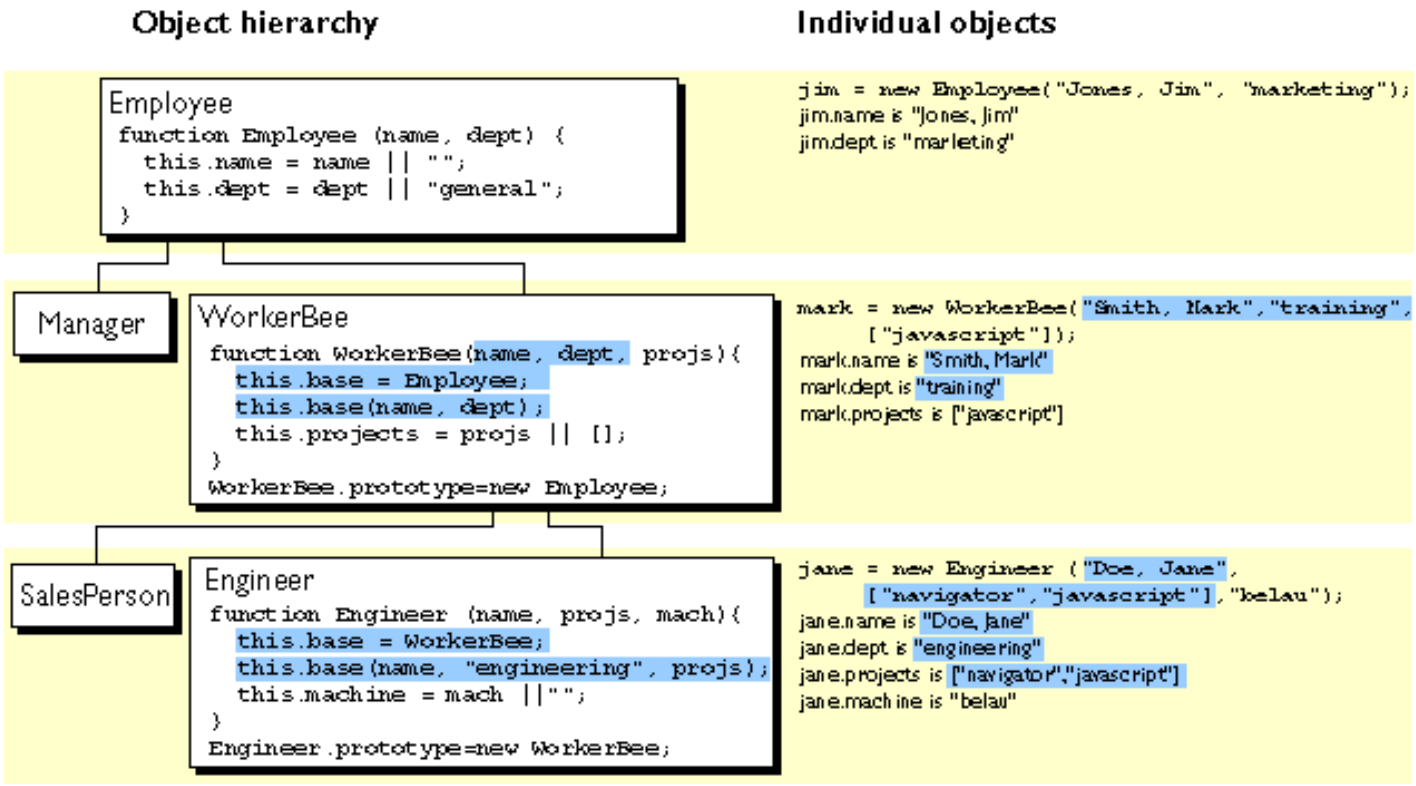
Jane's properties are now:

```
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.
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".

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.
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.

Starting with JavaScript 1.4, JavaScript provides an `instanceof` operator to test the protoytpe chain. This operator works exactly like the `instanceof` function discussed below.

As discussed in Inheriting Properties, 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.

Last Updated **September 28, 2000**

Part 2 Working with LiveConnect

Chapter 9 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.

Last Updated **September 28, 2000**

Chapter 9 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:

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

Working with Wrappers

In JavaScript, a *wrapper* is an object of the target language data type that encloses an object of the source language. When programming in JavaScript, 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 9.1 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 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:

```
x = java.lang.reflect.Array.newInstance(java.lang.Integer, 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.

In JavaScript 1.3 and earlier, `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.

```
// JavaScript 1.3
theClass = java.lang.Class.forName("java.lang.String")
theArray = java.lang.reflect.Array.newInstance(theClass, 5)
```

In JavaScript 1.4 and later, you can pass a `JavaClass` object directly to a method, as shown in the following example:

```
// JavaScript 1.4
theArray = java.lang.reflect.Array.newInstance(java.lang.String,
5)
```

Arguments of Type char

In JavaScript 1.4 and later, you can pass a one-character string to a Java method which requires an argument of type `char`. For example, you can pass the string "H" to the `Character` constructor as follows:

```
c = new java.lang.Character("H")
```

In JavaScript 1.3 and earlier, you must pass such methods an integer which corresponds to the Unicode value of the character. For example, the following code also assigns the value "H" to the variable `c`:

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

Handling Java Exceptions in JavaScript

When Java code fails at run time, it throws an exception. If your JavaScript code accesses a Java data member or method and fails, the Java exception is passed on to JavaScript for you to handle. Beginning with JavaScript 1.4, you can catch this exception in a `try...catch` statement.

For example, suppose you are using the Java `forName` method to assign the name of a Java class to a variable called `theClass`. The `forName` method throws an exception if the value you pass it does not evaluate to the name of a Java class. Place the `forName` assignment statement in a `try` block to handle the exception, as follows:

```
function getClass(javaClassName) {
   try {
      var theClass = java.lang.Class.forName(javaClassName);
   } catch (e) {
      return ("The Java exception is " + e);
   }
   return theClass
}
```

In this example, if `javaClassName` evaluates to a legal class name, such as "java.lang.String", the assignment succeeds. If `javaClassName` evaluates to an invalid class name, such as "String", the `getClass` function catches the exception and returns something similar to the following:

```
The Java exception is java.lang.ClassNotFoundException: String
```

See "Exception Handling Statements" on page 78 for more information about JavaScript exceptions.

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.

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 *Core 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.

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 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 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.

When you are evaluating JavaScript code in Java, the following situations can cause run-time errors:

-
- The JavaScript code is not evaluated, either due to a JavaScript compilation error or to some other error that occurred at run time.
  The JavaScript interpreter generates an error message that is converted into an instance of `JSException`.
- Java successfully evaluates the JavaScript code, but the JavaScript code executes an unhandled `throw` statement.
  JavaScript throws an exception that is wrapped as an instance of `JSException`. Use the `getWrappedException` method of `JSException` to unwrap this exception in Java.

For example, suppose the Java object `eTest` evaluates the string `jsCode` that you pass to it. You can respond to either type of run-time error the evaluation causes by

implementing an exception handler such as the following:

```
import netscape.javascript.JSObject;
import netscape.javascript.JSException;

public class eTest {
    public static Object doit(JSObject obj, String jsCode) {
        try {
            obj.eval(jsCode);
        } catch (JSException e) {
            if (e.getWrappedException()==null)
                return e;
            return e.getWrappedException();
        }
        return null;
    }
}
```

In this example, the code in the `try` block attempts to evaluate the string `jsCode` that you pass to it. Let's say you pass the string "`myFunction()`" as the value of `jsCode`. If `myFunction` is not defined as a JavaScript function, the JavaScript interpreter cannot evaluate `jsCode`. The interpreter generates an error message, the Java handler catches the message, and the `doit` method returns an instance of `netscape.javascript.JSException`.

However, suppose `myFunction` is defined in JavaScript as follows:

```
function myFunction() {
   try {
      if (theCondition == true) {
         return "Everything's ok";
      } else {
         throw "JavaScript error occurred" ;
      }
   } catch (e) {
      if (canHandle == true) {
         handleIt();
      } else {
         throw e;
      }
   }
}
```

If `theCondition` is false, the function throws an exception. The exception is caught in the JavaScript code, and if `canHandle` is true, JavaScript handles it. If `canHandle` is

false, the exception is rethrown, the Java handler catches it, and the `doit` method returns a Java string:

```
JavaScript error occurred
```

See "Exception Handling Statements" on page 78 for complete information about JavaScript exceptions.

Backward Compatibility

In JavaScript 1.3 and earlier versions, the `JSException` class had three public constructors which optionally took a string argument, specifying the detail message or other information for the exception. The `getWrappedException` method was not available.

Use a `try...catch` statement such as the following to handle LiveConnect exceptions in JavaScript 1.3 and earlier versions:

```
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.

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.

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. • `NaN` is converted to `NaN`.
`java.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 too large or small to be represented are rounded to +infinity or -infinity. • `NaN` is converted to `NaN`.
byte char int long short	• • Values are rounded using round-to-negative-infinity mode. • Values which are too large or small to be represented result in a run-time error. • `NaN` can not be converted and results in a run-time error.

`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 `equals()` method 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`	JavaScript 1.4: • • A JavaScript string is converted to an instance of `java.lang.String` with a Unicode value. JavaScript 1.3 and earlier: • • 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. The JavaScript string value is converted to a number according to the rules described in ECMA-262
char	JavaScript 1.4: • • One-character strings are converted to Unicode characters. • All other values are converted to numbers. JavaScript 1.3 and earlier: • • 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 and later versions 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`.
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	In JavaScript 1.3 and later versions, 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	In JavaScript 1.3 and later versions, 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 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	In JavaScript 1.3 and later versions, 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 and later versions 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 + "";
```

Last Updated **September 28, 2000**

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.

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.

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.

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

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**