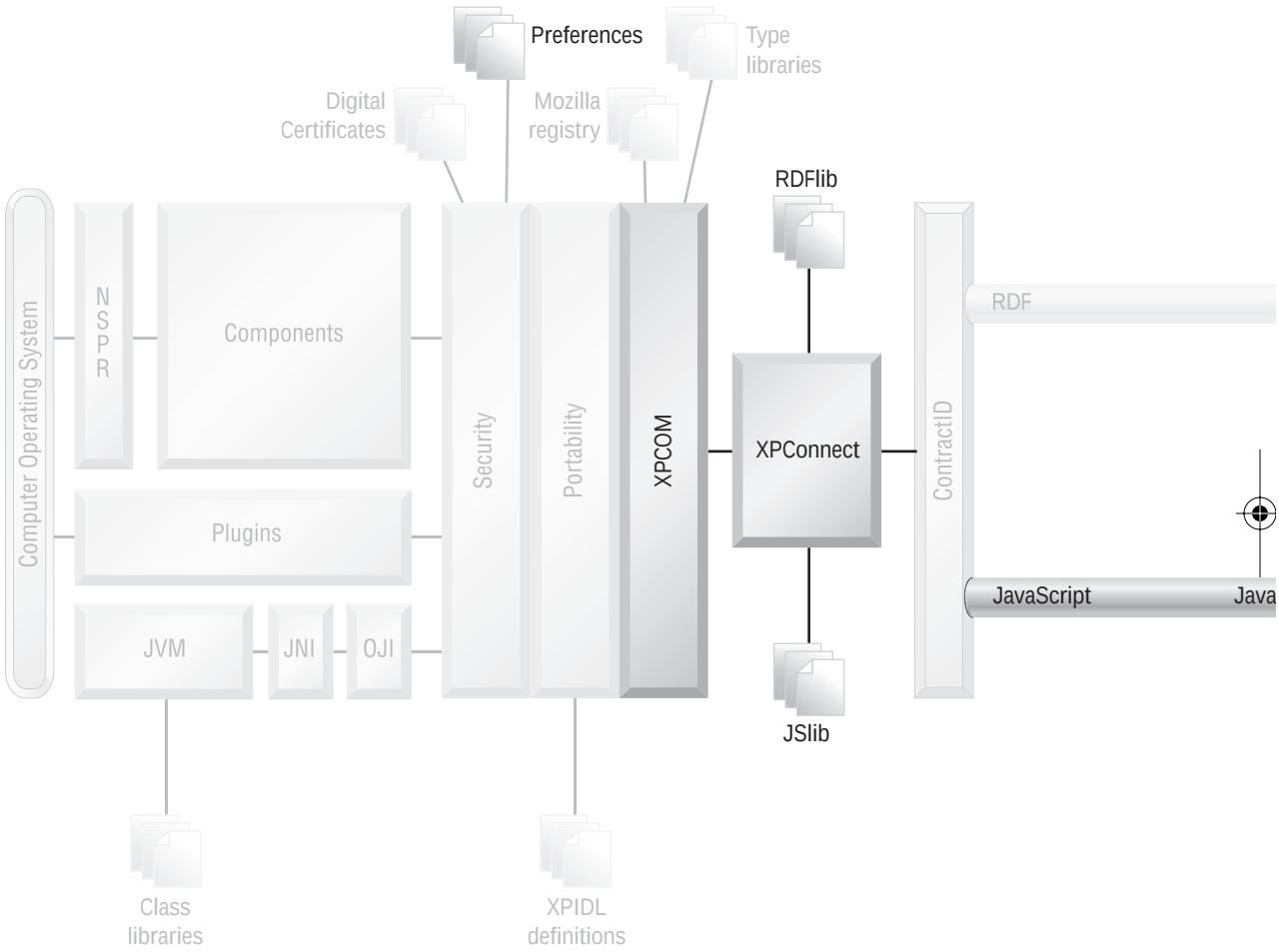
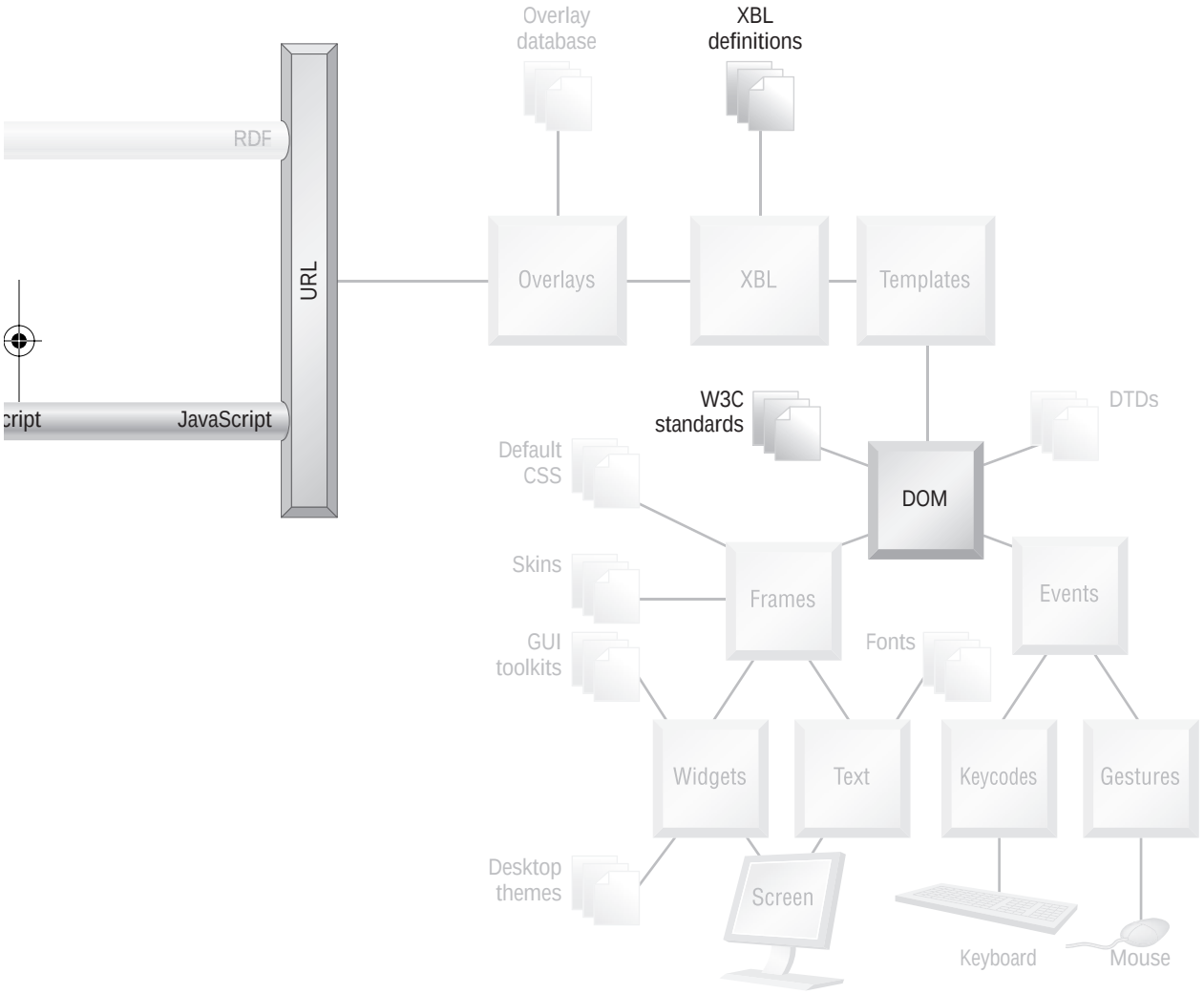

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
Plugins
JavaScript
Java
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 5

Scripting



JavaScript is a lightweight programming language with C-like syntax that is an essential part of the Mozilla Platform. JavaScript programs, or program fragments, are called scripts. By adding scripts to XUL, read-only documents become dynamic interfaces that can do something in response to the user's commands. Writing scripts is a task for a programmer, not for a Web page author or a content provider. Mozilla applications can be developed only by a programmer.

This chapter describes the JavaScript language itself. It also provides an overview of the many services of the Mozilla Platform that are exposed to JavaScript scripts. These features and services are examined in more detail throughout the rest of this book. Whole books have been written on JavaScript. This chapter is a complete description, but it is also brief.

JavaScript scripts that are part of a Mozilla application might follow one or more traditional programming styles, depending on how ambitious the application is.

Lightweight Mozilla applications contain scripts similar to the scripts used in Web pages. In Web pages, these scripts are often added to HTML content as an afterthought. When such scripts become larger, they are sometimes called Dynamic HTML. Even so, such scripts do little more than rearrange HTML content in a pleasing way. Simple macros used in products like Microsoft Word are similarly lightweight.

Middleweight Mozilla applications use scripting more systematically than lightweight applications. Other environments like this are 4GL tools such as Sybase's PowerBuilder or Oracle's SQL*Forms. In such cases, scripts are responsible for much of the basic activity put into the software. Use of Python in the Zope application server is an Open Source example.

Heavyweight Mozilla applications, like ActiveState's Komodo development environment, have so much scripting that the scripts overwhelm the basic browser-like features of the platform. In such cases, JavaScript can act like other standalone scripting environments such as Visual Basic, Perl, and Tcl/Tk, or perhaps emacs' elisp. In such heavily engineered programming, scripts drive the behavior of the Mozilla Platform from beginning to end.

It is the middleweight approach, however, that application programmers use the most. XUL, JavaScript, XPCOM, and default platform processing are combined into a final application. JavaScript is the glue that holds the other technologies together.

The NPA diagram at the start of this chapter shows the bits of Mozilla that are most responsible for scripting support. From the diagram, JavaScript is both away from the user and away from the computer's operating system. This is because JavaScript and its scripts are embedded technologies that hide within other software. Scripts rarely drive Mozilla from the outside. The two most important boxes, XPConnect and the DOM, are deep inside the Mozilla Platform. They have a large number of APIs (Application Programming Interfaces), which are all available to JavaScript scripts. The scripted use of these interfaces is the main construction task when developing a Mozilla-based

application. Being able to manipulate these interface script saves the programmer from having to use more laborious languages like C and C++.

The simplest use of JavaScript involves XML and the `<script>` tag. For both XUL and HTML documents, the following content will change the words in the title bar of a document's window:

```
<script> window.title = "Scripted Title"; </script>
```

Scripts like this can change any part of a displayed Mozilla window, including any part of the content. Such scripts can also interact indirectly with the wider world, including the Internet. The content of the `<script>` tag itself is highly meaningful and needs to be specially processed, just as the CSS2 content of the `<style>` tag is specially processed.

The Mozilla Platform itself is partly made out of JavaScript scripts. Such scripts are mostly found in the chrome, although a few outside exceptions, like preference files, also exist. Since the chrome is designed to contain Mozilla applications, it is no surprise that many scripts are stored there.

Before plunging into the language syntax, it's worth asking: Why pick JavaScript? That question is answered next.

5.1 JavaScript's Role as a Language

JavaScript is a member of the C family of programming languages. The most visible members of this family are C and ANSI C, Objective-C, C++, Java, JavaScript, PHP, C#, and awk.

The members of this family share syntax and some structure. These languages are all third-generation procedural languages, meaning that programs are specified as a series of ordered steps. They all contain the `if` keyword. Some of these languages are advanced enough to include support for objects. Objects provide structure on top of procedural steps.

JavaScript's essential qualities make it a (nearly) unique member of this group. It is designed to be the most accessible and easiest to use language in the family. It is also designed to be a take-anywhere language. It is highly portable, has tiny resource requirements, and doesn't need the support of a traditional compiler. This ease of use gives it wide appeal.

JavaScript code runs inside an interpreter, a virtual machine like Java's JVM, but one that is very small by comparison with that technology. It can be used in embedded devices, although the device must support 32-bit shifts and floating-point operations. Because JavaScript code is interpreted, variables are late-bound and weakly typed. This makes for an environment where it is very easy to get small programs working quickly, but very hard to get the last bugs out of big programs.

Because it is so small, JavaScript is heavily dependent on other software (called host software) before it can do anything meaningful. This is very similar to C, which can't do much without access to its companion `stdio` libraries,

or a `stdio` equivalent. The Mozilla Platform, in the form of a set of libraries and an executable, provides a host for the JavaScript interpreter.

Because the host is typically large, most of the time spent scripting in JavaScript is spent exploring what the host has to offer. Inside Mozilla, JavaScript has a role similar to Visual Basic for Applications (VBA) as used inside Microsoft Word and Microsoft Excel. This is also the kind of use with which Web developers are familiar.

5.2 STANDARDS, BROWSERS, AND <SCRIPT>

Mozilla supports the ECMA-262 standard, specifically ECMAScript version 1, edition 3. This standard is also named ISO-16262. ISO is the international standards organization (but ISO is not an acronym). ECMA is the European Computer Manufacturers' Association. Web addresses are `www.iso.org` and `www.ecma.ch`. ECMAScript standards in PDF form can be downloaded for free. With a little effort, the standard is useable as an everyday language reference. There are two other ECMAScript standards. Mozilla does not support either of them.

ECMA-327 "ECMAScript Compact Profile" is a near-identical version of ECMAScript intended for embedding in tiny devices. It removes a few features considered too complex for tiny implementations. It can be viewed as an attempt to compete against the WAPScript language, and other languages designed for embedded use.

ECMA-290 "ECMAScript Components" specifies how to create modular JavaScript programs using XML-based module files. It is the basis of Microsoft's Windows Scripting Components technology. Mozilla uses XBL instead.

ECMAScript is the official name for JavaScript, because Sun Microsystems owns the Java trademark. Trademark protection includes words that extend an existing mark or derive from one. Long ago, JavaScript's pre-release name was LiveScript. Mozilla's implementation of ECMA-262 edition 3 is called JS 1.5 and is nicknamed SpiderMonkey. It is an interpreter implemented as a C library.

SpiderMonkey also supports old versions of JavaScript, including the somewhat unusual version 1.2. That version contained a variety of new features, some of which failed to become popular. Support for early versions can be turned on if required, following the approach of older Netscape browsers. In a XUL application, the latest version should always be used. In other markups, such as HTML, any version may be chosen. The `<script>` tag, supported in both HTML and XUL, is the way to make this choice.

The correct way to include JavaScript code in an XML document is like this:

```
<script type="application/x-javascript" src="code.js/>
```

This next method is deprecated, so avoid it, even though it will still work:

```
<script type="text/javascript" src="code.js/>
```

Another method can be used to choose specific versions of JavaScript:

```
<script type="JavaScript1.2" src="code.js/>
```

This final method assumes the language is JavaScript and defaults to the latest version:

```
<script src="code.js/>
```

For a new Mozilla application, the first syntax is ideal. The final syntax is a useful alternative, but to be correct and precise, the `type` attribute should always be added. All these examples have the default encoding of `encoding="UTF-8"`.

Chapter 2, XUL Layout, under "Good Coding Practices," explains why JavaScript code should always be stored outside an XML document in a separate file. The section "Using XPCOM Components" later in this chapter shows how to include a JavaScript file from another JavaScript file.

The Mozilla organization has a second JavaScript interpreter, one written in Java rather than in C. This version, called Rhino, is not used or packaged with the Mozilla Platform; however, it is available to download. It is also ECMA-262 version 1 edition 3 compliant.

5.3 ECMAScript Edition 3

This section describes the features of the JavaScript language that come from the ECMAScript standard. Enhancements are discussed in the next section. Objects provided by the host software rather than by the standards are discussed after that.

5.3.1 Syntax

Here we look at the text of the JavaScript language. Because the ECMAScript standard is nearly readable by itself, this chapter goes over that syntax only briefly. Where it is possible, an attempt is made to explore consequences of the syntax.

Mozilla JavaScript scripts are stored in plain text files. The content of those files must be encoded in UTF-8 format. This means that characters from the ASCII character set are stored as a single byte. This is normal for English-speaking places, and no special preparation is required. Just use a plain text editor for scripts.

Some computers use the unused ASCII values from 128 to 255, perhaps for European characters such as the *é* in *résumé*. In the past, this tradition has been a handy way to create text in other languages. Such practices do not fol-

low the UTF-8 encoding rules, which demand two or more bytes for all non-ASCII characters. Embedding such 8-bit European characters won't work in scripts. "Ã©" is the correct UTF-8 encoding for *é*, or at least that is what the correct encoding looks like when viewed with a simple-minded text editor.

Even correctly specified, multibyte UTF-8-encoded characters have restricted use. They can only appear inside JavaScript string literals, inside comments, and sometimes in variable names. In general, stick to ASCII characters for your code; don't use other characters except as data. If you are European and want to use Latin1 characters for variable names, then in general they are safe if correctly expressed in UTF-8. For the fine points of Unicode support, see section 7.6 of ECMA-262 edition 3 and section 5.16 of the Unicode 3.0 standard.

Working with arbitrary Unicode characters is still possible in strings. See the section entitled "Data Types" for details.

5.3.1.1 Text Layout JavaScript is a free-format language like XML and C. Statements aren't restricted to a single line. Recognized whitespace characters include space and tab; recognized end-of-line characters include linefeed (hex 0A) and carriage-return (hex 0D). Beware that Windows, Macintosh MacOS 9, and UNIX all have a different concept of end-of-line. This doesn't affect the interpretation of scripts, but it can make editing them on the wrong computer harder.

Comments in JavaScript are written C-style. Single line comments are supported:

```
// Single line comment
```

as are parenthetic comments:

```
/* comment that can span
   multiple lines */
```

All combinations are possible except that multiline comments cannot appear inside other multiline comments. JavaScript does not have a special comment used for documentation. Scripts are not preprocessed before they are interpreted. There is no preprocessor like C's `cpp`.

5.3.1.2 Statements The basic unit of work in JavaScript is the statement. A statement is terminated by a semicolon, but that semicolon is optional. The one place that it is not optional is between the three expressions in a `for(;;)` statement. In all other cases, the JavaScript interpreter will automatically assume that a semicolon is present if it is left off. Three equivalent statements are

```
x = 5;
x = 5
x = 5   // same as previous line, even with this comment
```

This semicolon feature is designed to allow Visual-Basic style developers to feel comfortable with JavaScript. It is recommended that semicolons always be used. Not only is it clearer, but future versions of JavaScript will insist that one be present.

Scripts do not require a `main()` or any other kind of structure. Like Perl, statements can appear outside functions and outside objects from line 1 onward. JavaScript also supports the do-nothing statement:

```
;
```

JavaScript supports compound statements using the brace characters { and }, but they are different from compound statements in C (see the section entitled "Scope Rules." Bare compound statements aren't that useful in JavaScript, even though they are supported:

```
{ x = 5; y= 5; }
```

In this chapter, *statement* means either a single statement with trailing semicolon or a compound statement without a trailing semicolon.

5.3.1.3 Data Types

JavaScript has the following native data types:

```
Undefined Null Boolean Number String Object
```

There is also a hidden data type with no name that is a 32-bit signed integer.

In JavaScript, types are associated with data items, not with structures that hold data, like variables. There is an ancient analogy between programming variables and shoeboxes. In this analogy, a piece of data is a shoe, and a structure for holding it (a variable) is a shoebox. In JavaScript the type of information is attached to the shoe; it cannot be found in the shoebox. Variables holding types `Boolean`, `Number`, or `String` imply a shoebox with a single shoe of one kind in it. Type `Object` refers to a shoebox containing many shoes tied together into a single bundle. Type `Null` refers to a shoebox with nothing in it, and type `Undefined` refers to a shoebox whose contents aren't yet specified.

The `typeof` operator can be used to identify types. It will return one of the following strings for normal JavaScript data, or a custom string if the data tested come from the host software:

```
"undefined" "boolean" "number" "string" "object" "function"
```

"`object`" is returned if the data is of the `Null` type. Let's consider each type in turn.

The `Undefined` type has just one value (undefined) and no literal values. The global object (see later) has a single property named `undefined` that has this value. If necessary, the undefined value can be generated with the `void` operator, or with `undefined`:

```
x = void 0;
X = undefined;
```

The `Null` type has just one value: null. `null` is also a literal and can be used to empty a variable or in a comparison.

```
x = null;
```

The `Boolean` type has two values, true and false. `true` and `false` are literals for these values. False does not equal 0 (zero) as it does in C, but the conversion process between false and 0 is so seamless that any subtle difference can usually be ignored.

```
x = true;
```

The `Number` value stores a 64-bit double-precision floating-point number, as described in the IEEE 754 standard. That standard is not free, but a near identical draft can be had from `http://www.validlab.com/754R/`.

Floating point is an inexact attempt at representing a real number and is accurate to at least 15 digits, unless mathematical operations introduce further error. The IEEE 754 standard allows for Not-a-Number values (possibly resulting from dividing zero by zero, or taking the inverse sine of 2), and for Infinite values (possibly caused by overflow). The `isNaN()` and `isFinite()` methods can be used to test for these conditions. JavaScript has no literals for these values, but the global object has an `NaN` and an `Infinity` property, and the `Math` object has several handy properties:

```
POSITIVE_INFINITY NEGATIVE_INFINITY NaN MAX_VALUE MIN_VALUE
```

These properties can be used for comparisons. Floating-point literals support exponential notation up to about +/- 10^{300}:

```
x = -3.141592654;  y = 1.0e+23;   z = 234.555555E-100;
```

Number literals can also be specified in hexadecimal by using a 0x or 0X prefix, followed by the digits 0–9 and A–F in upper- or lowercase.

```
x = 0xFEFF;
```

JavaScript's method of comparing NaN values matches the recommendations in IEEE 754, but the unique identity of different IEEE 754 NaN values is not preserved by the ECMAScript language.

Because floating-point numbers are inexact and subject to error, they are poor choices for programming counters and indices. Integers are a better solution for such common tasks. Inside Mozilla's JavaScript interpreter, `Number` data are actually stored as 31-bit signed integers until there is a clear need for floating-point accuracy. The end result is that by avoiding division and by keeping numbers below about 46,000 (the square root of 2^{31}), most simple calculations in JavaScript are exact integer arithmetic without any floating-point error. These whole integers are also big enough to store any Unicode value or any CSS2 RGB (Red-Green-Blue) color value.

Several situations can cause a value to be stored as a true floating-point number. Some examples are: if a number literal has a decimal point; if division

occurs where a remainder would result; if a function that has a real result (like `sin()`) is applied; or if mathematics results in a number bigger than 2^{31}. At all other times, `Numbers` are stored as integers.

If a number is converted from integer to floating-point representation, floating-point errors do not automatically occur. The IEEE 754 floating-point representation has 54 bits of precision, which is enough accuracy for all but the most intensive and repetitive calculations.

The `String` type represents a Unicode sequence of characters stored in a UTF-16 encoding (i.e., two bytes per character). Each string appears immutable as in Java; strings cannot be worked on "in place" as they can in C. String literals are delimited by a pair of single or double quotes. Special notation can be used for common nonprintable characters. This notation is inspired by special characters used in C strings. The JavaScript versions are

```
\b \t \n \v \f \r \" \' \\ and \x and \u
```

These characters are backspace, tab, newline, vertical tab, formfeed, carriage return, double quote, single quote, backslash, and the byte and Unicode leader characters. The byte leader character must be followed by two hexadecimal digits and specify a Unicode character whose code-point (character index) is between 0 and 255. This includes the ASCII characters and the nonbreaking whitespace character (0xA0). It also includes ISO 8859 (Latin 1 - European) characters. The Unicode leader character must be followed by four hexadecimal digits and specifies any Unicode character you can think of. A trivial example is

```
str = "hello, world\n";
```

The `Object` type will be discussed in its own section shortly.

JavaScript provides automatic type conversion between most types. This means that a piece of data used in a context where a certain type is expected will be converted to that type before use. Such conversion is also discussed later. JavaScript does not have a casting system, but methods that can convert between types explicitly are available.

5.3.1.4 Variables JavaScript is a 3GL and therefore has user-defined variables. Variable names must start with an alphabetic letter or underscore or with $. $ should be avoided because it is rarely used in handwritten code. There is no limit on the length of a variable name. Alphanumeric characters, the dollar sign, and the underscore are the allowed characters. Variable names are case-sensitive.

```
my_variable x counter5 interCapName not$common _secret
```

Naming conventions recommend that all capitals be used for constants, with underscore as word delimiter (like Java); an initial capital be used for object constructors; and an initial underscore be used to indicate a variable is not intended for informal use.

Variables are undefined unless declared with the `var` keyword. If they are used without being declared, then that is a syntax error. If they are declared but nothing is assigned to them, then they are undefined. Variables are therefore defined when declared or on first use, whichever comes first. Initial assignment of variables in JavaScript is not restricted to constant expressions as is the case in C. Any expression may be used for initial assignment, and a variable can be declared anywhere in a script.

```
var x;
var y = 2, z = 3.45;
var product = x * y * z;
```

JavaScript variables store one item of data each. That item of data is either a simple value or a reference to an object. In the section entitled "Objects," it is explained that variables are also properties. JavaScript does not have pointers, and there is no syntax supporting references explicitly. Variables names may not be reserved words like `if`. The thing named `this` is a special variable that always refers to the current object.

5.3.1.5 Arrays JavaScript supports single-dimensioned arrays, like C, but their size can be specified by a nonconstant expression. Arrays are created using the `new` keyword, which is used to create objects of many kinds. There are several syntactic options for array creation:

```
var arr1 = new Array();             // zero-length array
var arr2 = new Array(5);            // array of 5 items
var arr3 = new Array(11,12,13);     // array of 3 items
var arr4 = new Array(2,"red",true); // array of 3 items
```

All array elements will be undefined unless content for the elements is specified at creation time. Each element of an array may contain data of any type. An array may also be created from a literal array, using the `[` and `]` bracket characters. These examples match the last ones, and are often preferred because the `Array()` method is a little ugly and confusing:

```
var arr1 = [];                // zero length array
var arr2 = [, , , , ,];       // array of 5 items
var arr3 = [11,12,13];        // array of 3 items
var arr4 = [2,"red",true];    // array of 3 items
```

Array literals can be nested so that array elements can themselves be arrays:

```
var arr5 = [ 6, ["red","blue"], 8, [], 10];
```

Array elements are referred to by their indices, which start at 0. The `length` property of an array is an integer one larger than the highest array index in the array. It is not equal to the number of elements in the array. It is kept up to date automatically:

```
a[0];          // first element of array a
b[2];          // third element of array b
c.length;      // one greater than highest index in c
c[c.length-1]; // last element of array c
d[1][4];       // see below
```

The last line in the preceding example is an array `d` whose second element `d[1]` is an array. Therefore, `d[1][4]` is the fifth element of the `d[1]` array.

Arrays are not fixed in size. The length property can be changed, either by assigning to it or by setting an element with a higher index.

```
arr1 = new Array(3);   // length is 3
arr1.length = 5;       // length is now 5;
arr1[8] = 42;          // length is now 9;
```

Arrays are sparse. In the preceding example, when the element with index 8 is set, the elements between index 5 and 8 are not created unless they are used in later statements. Because indices have the range of 32-bit unsigned integers, gaps between elements can be very large.

Looking ahead a little, arrays are also objects (of type `Array`). All objects support a little syntax that allows them to act like an array. When this is done, the object treated like an array does not gain a `length` property; it is merely allowed to use the square-bracket array notation. This array syntax can be used to find properties on all object-like data, but this flexibility doesn't benefit arrays, only other objects, as this example shows:

```
obj.prop_name == obj["prop_name"] // legal and always true
obj[1] != obj.1                   // illegal syntax
```

The syntax in the second half of the first line is useful when an object property needs to be created whose name is not a legal variable name. For example,

```
obj["A % overhead"] = 20;
```

A subtle trap with this array syntax support is caused by type conversion. Array element indices that are not integers are *not* rounded to the nearest integer. They are converted to strings instead:

```
obj[12.35] == obj["12.35"];
```

This example results in an object property being set rather than an array element because there are no floating-point indices. Array indices are typically stored in variables. If an index has been converted from an integer to a floating point as a result of some calculation, then this subtle type conversion can happen invisibly. It is difficult to spot because the property that is set will be used somewhat reliably, until the floating-point value is rounded or accumulates a fractional part due to calculation error. At that point, it will convert to a subtly different string that points to a different property. The value set also can't be recovered via a normal object property because the property name

12.35 is an illegal variable name. The moral is: Don't do complex mathematics on indices.

5.3.1.6 Expressions JavaScript expressions follow C, C++, and Java's expressions very closely and provide a means to do mathematics, bit operations, Boolean logic, and a few operations on objects. Expressions consist of variables, literals, and the operators noted in Table 5.1.

Table 5.1 JavaScript operators

Name	Binary?	Precedence	Symbol
Force highest precedence	Unary	0	( )
Array literal	Unary	0	[ ]
Object literal	Unary	0	{ }
Function call	Unary	0	( )
Property of		1	.
Element of	Binary	1	[ ]
Object literal	Unary	1	{ }
Create object	Unary	2	new
De-reference property	Unary	3	delete
Convert to undefined	Unary	3	void
Reveal type as a string	Unary	3	typeof
Pre- and postincrement	Unary	3	++
Pre- and postdecrement	Unary	3	--
Same sign	Unary	3	+
Opposite sign	Unary	3	-
32-bit bitwise NOT	Unary	3	~
Logical NOT	Unary	3	!
Multiplication		4	*
Division		4	/
Modulo		4	%
Addition, concatenation		5	+
Subtraction		5	-
32-bit left shift		6	<<
32-bit signed right shift		6	>>

Table 5.1 JavaScript operators (Continued)

Name	Binary?	Precedence	Symbol
32-bit unsigned right shift		6	>>>
Matches given type		7	instanceof
Matches object property		7	in
Ordinal comparisons		7	< > <= >=
Equality		7	== !=
Strict equality		7	=== !==
32-bit bitwise AND		8	&
32-bit bitwise XOR		9	^
32-bit bitwise OR		10	|
Logical AND		11	&&
Logical OR		12	||
Conditional	Ternary	13	?:
Simple assignment		14	=
Compound assignment		14	*= /= %= += -= <<= >>= >>>= &= ^= |=
List element delimiter		15	,

Precedence of 0 is the highest precedence. JavaScript roughly follows the left-to-right and right-to-left conventions of C for equal-precedence operators. It also supports short-circuit Boolean expressions, which means that an expression consisting of many `&&` and `||` operations is processed from left to right only until the final result is sure, not until the final term is evaluated.

One area where JavaScript Boolean logic is closer to Perl than C is in *multiplexed value sematics*. In this arrangement, `&&` and `||` used in expressions are evaluated as control-flow conditions similar to `?:` rather than as simple Boolean expressions. Thus in

```
var x = flag && y;
```

variable `x` evaluates to `y` if flag is `true`, and `false` otherwise, rather than evaluating to the Boolean result of "flag and y."

For mathematical expressions, a mixture of `Numbers` stored as integers and floating points results in all `Numbers` being promoted to floating points. If bitwise operations are attempted on `Numbers` stored as floating points, the floating-point numbers are first chopped down to 32 bits in a manner that is generally useless and unhelpful. Make sure that bit-operations only occur on `Numbers` stored as integer values.

5.3.1.7 Flow Control JavaScript supports C-style flow control. The standard forms are as follows, with the placeholder *statement* being either a single statement or a list of statements surrounded by { and }.

```
if (expression) statement
if (expression) statement else statement
while (expression) statement
do statement while (expression)
for (expression; expression; expression) statement
switch (expression) {
  case expression: statement; break;
  case expression: statement; break;   // as many as needed
  default: statement; break;
}
```

The argument to `switch()` can be anything, not just a variable. The `case` selectors are not restricted to literals either. The following two `if`s are the same:

```
if (a) statement else if (b) statement else statement
if (a) statement else {if (b) statement else statement}
```

As for many C-like languages, beware of the dangling `if` trap in which an `else` clause is attached to the last `if`, regardless of indentation; that trap is avoided by using the second, explicit syntax in this last example.

The `for` statement has a variant for stepping through the properties of a JavaScript object. Only the properties that are *not* `DontEnum` (see section 8.6.1 of ECMA-262) participate:

```
for ( variable-name in object ) statement
```

JavaScript has no `goto` statement. It does have labels, which are named as for variables but they are in a separate namespace. `continue` ends the current iteration of a loop; `break` leaves a loop or a switch statement permanently. A label can be used to break up more than one level when loops are nested several levels deep:

```
mylabel: statement;
break;
break label;
continue;
continue label;
```

JavaScript also has an exception system. It is not an optional add-on. It is part of the core language. It catches run-time errors and exceptions.

```
try { statement; }
catch (variable) { statement; }
finally { statement; }
```

There can be more than one `catch` block. The `finally` block is optional. Inside a `try` block, or anywhere, `throw` can be used to generate an exception:

```
throw expression;
```

The expression thrown can equate to any type of information from a simple number to a complex purpose-built object. To mimic the exceptions thrown by the XPConnect part of the platform, always throw a 32-bit integer, preferably including one of the values of the `Components.results` object.

Scripts are often written hastily with a quick purpose in mind, and exception handling is a less well-understood feature of 3GL languages. In reality, for a robust script, most processing should be contained in `try` blocks, as exceptions should never, ever reach the user. The simplest and most efficient way to ensure this is to wrap everything in a single, top-level `try` block

The `with` statement is discussed under "Scope."

5.3.1.8 Functions JavaScript supports functions. Functions are untyped and support a variable number of arguments, like C's `printf()`. Functions can also have no name, in which case they are anonymous. Listing 5.1 shows a typical function:

Listing 5.1 Ordinary JavaScript function syntax.

```
function sum(x, y)
{
  if (arguments.length != 2)
  {
    return void 0;
  }
  return x + y;
}

var a = sum(2,3);           // a = 5
var b = sum(1,2,3);         // b = undefined
var c = sum("red","blue"); // c = "redblue"
var d = sum(5, d);          // d = 5 + undefined = NaN
var e = sum;                // e is now a function
var f = e(3,4);             // f = 7
```

The `arguments` object acts like an Array object, except that it is static—if more elements are added to the array, its `length` property will not be updated. It contains all the arguments passed in to the function. Functions can be anonymous as well:

```
var plus = function (x,y) { return x + y; }
var a = plus(2,3);
```

The benefit of anonymous functions is that they don't automatically create an extra variable with the name of the function. Consequently, it is possible to set methods on objects without globally defined function variables hanging around. Globally defined function names can also be avoided by placing a named function's definition inside an expression:

```
var five = (function sum(a,b){return a+b;})(2,3);
```

If a function is called by itself, not as an object method, then any use of the keyword `this` is resolved by scoping rules.

5.3.1.9 Regular Expressions JavaScript supports Perl5 regular expressions, with some obscure and rarely seen differences. Obscure differences exist because regular expression syntax is subtle in detail and always evolving and being fixed. UNIX systems have *file*, *normal*, and *extended* variants of regular expressions. Perl and JavaScript support extended regular expressions that very, very roughly match `egrep(1)`, or the "Wildcards" feature of Microsoft Word's Find dialog box. The Perl `man(1)` manual page `perlre` is easier to understand than the ECMAScript definition, but not much. Look for an online tutorial.

All regular expression operations in JavaScript are methods of the String object or the `RegExp` object; they do not have standalone existence like Perl's `m//` operator:

```
match(re)             // "red".match(/e/) == ["e"];
replace(re,string)    // "red".replace(/e/,"o") == "rod";
replace(re,function)  // "red".replace(/e/,myfn);
search(re)            // "red".search(/e/) == 1;
split(re)             // "red".split(/e/) == ["r","d"];
```

`replace()` returns a string; `search()`, an integer; and `match()` and `split()` return an array of strings each.

Regular expressions have a literal syntax that can be typed into source code anywhere that a string literal might occur. It is converted immediately to a `RegExp` object. The syntax is

```
/pattern/flags
```

`pattern` is any of the convoluted syntax of regular expressions; flags is zero or more of `g` (replace everywhere), `i` (ignore case), and `m` (match lines of multiline targets separately).

5.3.2 Objects

JavaScript has objects, but at version 1.5, it is not fully object oriented. JavaScript's report card for the many descriptive terms that apply to object-like systems is shown in Table 5.2.

All Objects and their attributes are late-bound in JavaScript. Attempting to write fully object-oriented code is a technical feat that should generally be avoided. JavaScript is designed for simple manipulation of objects, not for creating complex (or any) class hierarchies. Most of the objects used in a JavaScript script come from the host software and are implemented in another language.

There are no class definitions in JavaScript 1.5; there are only run-time types. A normal object class system allows objects to be created using an abstract specification—a class. This is like using a sketch plan to carve a

Table 5.2 JavaScript object support

Object system concept	Syntax support	Ease of use
Aggregation	High	Easy
Containment	High	Easy
Delegation	High	Medium
Encapsulation	Low	Medium
Inheritance	Medium	Difficult
Information hiding	Low	Medium
Interfaces	None	Difficult
Late-binding	High	Easy
Object-based	High	Easy
Object-oriented	None	Difficult
Multiple inheritance	None	Difficult
Run-time type reflection	High	Easy
Templates	None	Difficult

statue. JavaScript uses a prototype system to create objects instead. This is like carving a statue by starting with another statue. All JavaScript objects, except Host objects, are created using another object as the initial ingredient. This is more flexible than a class-based system, but to do anything sophisticated with it in JavaScript results in somewhat messy syntax.

5.3.2.1 Plain Objects A piece of data of type `Object` in JavaScript has a set of properties that are the contents of the object. Some properties contain Function objects and therefore are also called object methods. The special global object's properties are also called variables and functions. Just about everything in JavaScript is a property of some other object. There is no information hiding: In C++ or Java terms, all properties are public.

All properties have attributes, but these attributes are a subtle feature of the language and have no syntax of their own. You can read about attributes in the ECMAScript standard, but you should really just forget them. They are different from XML attributes.

To create an object of your own, use either `new` or an object literal as in Listing 5.2.

Listing 5.2 Examples of object creation in JavaScript.

```
// explicit approach
var obj = new Object;
```

```
obj.foreground = "red";
obj.background = "blue";

// literal approach
var obj = { foreground:"red", background:"blue" }
```

To add a method to an object, just use a function or an anonymous function as a property's value. Anonymous functions can also appear in object literals as shown in Listing 5.3.

Listing 5.3 Examples of method creation in JavaScript.

```
function start_it() { this.run = true; }

// explicit approach
var obj = new Object;
obj.start = start_it;
obj.stop = function (){ this.run = false; }

// literal approach
var obj = { start: function (){ this.run = true; },
            stop: function (){ this.run = false; }
          }

// execute the methods
obj.start();
obj.stop();
```

The `this` operator refers to the object that the function using `this` is a property of.

Objects may contain other objects:

```
var obj = {
  mother:{name:"Jane", age:34},
  father:{name:"John", age:35}
};
var my_ma_name = obj.mother.name;
```

Containment and association are the same thing in JavaScript because there is no information hiding. It is possible to do some information hiding by messing around with the properties of `Function` objects in the prototype chain. This can create permanent properties that are in scope only when a function runs—effectively a private variable. That obscure technical trick is generally unnecessary. Only do it if you are supplying a library of precreated objects for someone else's consumption, and you want the library to be absolutely rock-solid. See also the section entitled "Language Enhancements" for information on property getters and setters.

If many objects with similar properties are to be created, then fully specifying each one by hand is a tedious solution. An object constructor is a better solution. This is a function called as an argument to `new`. Inside the function,

whatever standard properties the object needs are set. The constructor function can then be reused for each new object, as Listing 5.4 shows.

Listing 5.4 Examples of object construction using a constructor function().

```
function Parents(ma, pa)
{
  this.mother = { mother:ma; };
  this.father = { father:pa; };
  this.dog = "Spot";
}
var family1 = new Parents("Jane","John");
var family2 = new Parents("Joanne","Joe");
```

This system can be further improved. As soon as the constructor function exists, its prototype object can be modified. The prototype object for a constructor is an object that contributes default properties to the constructed object. The following lines could be added to Listing 5.4, immediately after the function definition for `Parents`:

```
Parents.prototype.lastname = "Smith";
Parents.prototype.ring = function (){ dial(123456789); };
```

When `family1` and `family2` get new parents, they will also have properties matching those in the prototype, for a total of five properties. Not only will they both have a dog named Spot, but they will both be Smith and both be able to ring on the same number. In fact, the `lastname` and `ring` properties are shared between the two objects. If one object updates its `lastname` property, that value will override the prototype's `lastname` property, and the object will cease to share the `lastname` property of the prototype. If the prototype object's `lastname` property is updated, then any objects sharing that property will see the change. This is not the case for the `dog` property, which is unique to each object created.

The purpose of this system is to allow an object to be extensively modeled once (by the prototype object) and then to permit that model to be reused by the constructor when a copy is created. The constructor can be restricted to dealing with construction parameters and running any initialization methods that might be required. Unfortunately, the prototypes system is somewhat unencapsulated because the prototype properties must be set outside the constructor.

See also the section entitled "Prototype Chains" in this chapter.

5.3.2.2 Host Objects Host objects exist outside JavaScript in the host software in which the interpreter is embedded. In Mozilla, such objects are typically written in C++. A piece of C code attached to the JavaScript interpreter finds them when they are asked for, constructs a simple internal interface that looks like a JavaScript object, hooks this up to the Host object, and thus exposes the Host object to scripting. The tricky part is finding the object in the

first place, and JavaScript needs assistance from the host software (the Mozilla Platform) for that. The `document` object is an example of a host object.

Host objects appear the same as plain JavaScript objects to programmers, unless you try to turn them back into code using `toString()`. Because functions and methods are also objects in JavaScript, this difference applies to single functions or methods as well. For example, this piece of code attempts to discover the function body of the `alert()` function (which in XUL is a method of a host object of type `ChromeWindow`):

```
var str = "" + alert;
```

The resulting string, however, shows that `alert()` has no JavaScript source:

```
"\nfunction alert() {\n    [native code]\n}"
```

5.3.2.3 Native Objects A JavaScript interpreter has its own range of objects. Their names and types are

```
Object Array Boolean Number String Math Date RegExp Function Error
```

Objects can be created automatically and on the fly by the JavaScript interpreter, or they can be created by hand, which requires a constructor. These object names are also the names of the matching object constructor objects. Therefore, construct an object of type `Boolean` with the `Boolean` object constructor object. That constructor object is named `Boolean`:

```
var flag = new Boolean();
```

Technically it makes a difference whether an object prototype object or an object constructor object is used with `new`, but in practical terms they are the same. The latter case requires function parenthesis, whereas the earlier case doesn't.

The `Object` and `Array` types have already been discussed. `Array` objects maintain a length property, whereas `Object` objects don't.

The `Boolean`, `Number`, and `String` objects match the basic data types of the same name. JavaScript freely and automatically converts between simple data and objects of the same kind, even for literals. This example shows automatic conversion of literals to objects, which then execute a single method.

```
"Test remark".charAt(3);   // result: "t"
1.2345.toFixed(2);         // result: 1.23
true.toString();           // result: "true"
```

The `Math` object provides numerous basic mathematical operations, such as `Math.sin()`.

The `Date` object stores dates and has many accessor methods. `Date` objects only support a version of the Western Gregorian calendar extended forwards and backwards in time. They support dates before the UNIX Epoch (1 January 1970) and are not 32-bit `time_t` values. They are IEEE double preci-

sion and reach backward and forward 280,000 years. Dates are accurate to one millisecond, provided that the computer has an accurate clock. The zero value for dates matches the UNIX Epoch, so all `time_t`'s are valid `Date` values. Do not use the `getYear()` method, which is old; use the `getFullYear()` method instead.

The `RegExp` object holds a regular expression pattern and flags. Some methods related to regular expressions also exist on the `String` object.

The `Function` object is the object that represents functions and methods. It has clumsy constructor syntax. Anonymous functions or the `function` keyword syntax are almost always preferred to using "`new Function`".

The `Error` object reports run-time errors and exceptions that aren't caught by `catch` or `finally`. It is little use to application programmers, who can just look at Mozilla's JavaScript Console for the same information.

When studying one of these objects, use the ECMA-262 standard as a reference. The properties and methods for object of type X are described in section 15 under "Properties of the X Prototype Object" and "Properties of X Instances." This rule applies for all cases except the `Math` object (see the next section).

5.3.2.4 Built-in Objects and Similar Effects When the JavaScript interpreter starts, some objects are made available without any user scripting. These are called *built-in* objects if they are native objects. It's also possible for host objects to appear before any scripting occurs. Such automated setup is always done for convenience reasons. The best examples are the `Global` object, the `Math` object, and the `document` object.

The global object sits at the top of an object containment hierarchy. It is the root object in JavaScript. It is not the property of any other object, and a global object cannot be created without creating a separate and independent run-time environment. In Mozilla, the Window object (for HTML) and ChromeWindow objects (for XUL) are global objects. Therefore, each Mozilla window is a separate run-time environment. These global objects are implemented so that they have a `window` property. That window property refers back at the global object (a loop). Programmers use this window object as the "top-level object" in scripts.

A `Math` object is also created every time a JavaScript instance starts. This is referenced by a property of the global object called `Math`. It allows the following shorthand syntax for mathematical operations:

```
var one = Math.sin(Math.PI/2);
```

If a document is loaded into a Mozilla window, then that loading process can automatically populate JavaScript with many additional objects. These objects are familiar to Web programmers as the Document Object Model, level 0. In HTML, these objects form a large containment hierarchy commonly used like this:

```
window.document.form3.username.value = "John";
```

The explicit use of a `window.` prefix is optional. Equivalent prefixes are `this` and `self`.

By comparison with HTML, XUL has a very limited set of precreated objects. It uses an XPCOM name service to find host objects that are not precreated.

5.3.3 Processing Concepts

Separate to visible syntax is the way the JavaScript interpreter crunches through your script. There are a number of novel concepts to the language.

5.3.3.1 Operator Precedence Precedence of operators is noted in Table 5.1. The left-to-right and right-to-left ordering that JavaScript uses is similar to that of C, C++, and Java.

5.3.3.2 Argument Passing All function and method arguments are passed by reference, except for Booleans, numbers, null, and undefined. Those few cases are passed by value (copied).

5.3.3.3 Type Conversion JavaScript automatically converts data between all simple types and the `Number`, `String`, and `Boolean` types. It forces type conversion so that expressions can be evaluated in all cases. Every object in JavaScript has a `toNumber()` method and a `toString()` method that are used to assist in this process. Casts are not required; it is done according to an extensive set of rules in the ECMA-262 standard. These rules can be boiled down to just two rules:

> Rule 1: Never assume that conversion will work when trying to change a string into a number.
>
> Rule 2: Don't use binary operators on objects whose types you aren't sure of.

Rule 1 exists because the contents of a string might be an invalid number literal. This code will cause the JavaScript interpreter to return `NaN` and, in worse cases, the interpreter may halt:

```
var str = "123stop45";
var x = str * 3;         // str isn't a number.
```

Halting can only occur if syntax errors or run-time errors occur. To guard against such things, use the `parseInt()` and `parseFloat()` functions explicitly instead:

```
var x = parseInt("123stop45") * 3;
```

Rule 2 exists because the comparison operators (<, ==, etc.) and + are overloaded for Strings and Numbers. The rules that decide whether to ultimately treat both operands as strings or numbers are not immediately obvi-

ous, and they have different senses for comparison and concatenation operators. If in doubt, see Rule 1.

5.3.3.4 Scope Scope is the process of deciding what variables, objects, and properties are available to use at what point in the code. Scope in JavaScript has two sides.

The first side is traditional variable scoping. This is the same as C and C++ where variables may be local to a function or global. In JavaScript, functions can have a local variable with the same name as a variable outside the function. When the function is interpreted, the local variable is used. When statements outside the function are interpreted, the global variable is used.

In C and C++, in addition to function and global scopes, every statement block has its own block scope. This means that the scope of a variable declared inside a set of statements surrounded by { and } is different from that of variables declared outside. In JavaScript, such a statement block does *not* create a new scope. It has no such effect.

In C and C++, variables declared partway through a scope are valid from that point onward in the scope. In JavaScript, variables declared partway through a function (or through a global section of code) are valid for the entire scope. This code, illegal in the equivalent C/C++, is valid JavaScript.

```
function f() {
  alert(x);    // reports "undefined"
  var x = 42;
  alert(x);    // reports "42";
}
f();
```

The second side to scope is JavaScript's unusual concept of scope chains. A scope chain is an ordered list of objects that has at one end the global object. When a function call occurs, the interpreter must first find the function matching the function call. It does so by looking at the objects in the scope chain. The first object found that has a method with the same name as the function will be executed as that function. This is the mechanism that allows event handlers in Web pages to call functions based on the window object, even though the current object is something else. A scope chain makes services from several objects available at the same time.

The `with` statement in JavaScript adds objects temporarily to the scope chain. In Listing 5.5, you can see that the `toString()` function is used repeatedly, and each time it is found in a different object. At the same time, the `myflag` variable is always found in the `window` object because none of the other objects has a `myflag` property.

Listing 5.5 Example of JavaScript scope chains at work.

```
// scope chain = window
var myflag = "Test String";
```

```
var x = toString();     // "[object ChromeWindow]"
with (document)
{
  // scope chain = document, window
  x = toString();       // "[object HTMLDocument]"
  x = new Object;
  with (x)
  {
    // scope chain = x, document, window
    var y = toString(); // "[object Object]"
    var x = myflag;      // window.myflag
  }
}
```

Scope chains are a distant cousin to the vtables inside C++ that implement inheritance, except that scope chains are not fixed sizes and do not implement classical object inheritence. For application programmers, scope chains are handy but usually require no thought.

5.3.3.5 Stacks and Garbage Collection C, C++, and Java use two kinds of memory for data: a stack and a pool (also called a heap). The stack is used automatically as required, and the pool is used whenever `new` or `malloc()` is called. JavaScript has no stack, or at least no stack with which the programmer can work. Everything goes into the memory pool. Neither array elements nor function arguments are guaranteed to be contiguous. Stacks are used to implement passing of function arguments in the Mozilla SpiderMonkey implementation, but that stack mechanism is not visible to JavaScript programmers.

Java is a pointer-free garbage-collected language, as is JavaScript. JavaScript has one garbage collector per run-time instance. It is possible for objects in one window to contain references to objects in other windows. When a window is destroyed, any user-defined JavaScript objects are marked for collection. JavaScript's garbage collector does not run in a separate thread as Java's does.

5.3.3.6 Run-time Evaluation JavaScript has the capability to interpret new code at run time. The most powerful and general way to do this is by using the global object's `eval()` method. `eval()` can be used to execute any legal JavaScript:

```
var x = 5;
var code = "x = x + 1;";
eval(code);                 // x = 6
x = x + 1;                  // x = 7
```

Other methods for evaluating code at run time are not specific to the interpreter itself. In Mozilla, they include the `setTimeout()` and `setInterval()` methods, the very limited `parseInt()` and `parseFloat()`, and the

`javascript:` URL. See the section entitled "Using XPCOM Components" for a further script-evaluation mechanism.

5.3.3.7 Closures JavaScript supports closures. This example illustrates:

```
function accumulate(x) {
  return function (y) { return x + y };
}
var subtotal = accumulate(2);
var total    = subtotal(3);  // total == 5
```

When the anonymous function is called, what will the return value be? If the `x` argument's state is cleaned up when the `accumulate()` function's scope ends, then it won't be available to contribute to the return value of the `subtotal()` call.

Closures are a solution to this problem. A closure is a collection of data that needs to live beyond the end of a function. It is basically a copy of scoped variables that refer to other objects. The copies are returned; therefore, the referred-to data doesn't lose its last reference when the originals are cleaned up. This prevents garbage collection of created objects when the scope ends. Closures are invisible to the programmer. Closure behavior makes sense for any language that supports run-time code evaluation.

5.3.3.8 Prototype Chains Perhaps the most complex feature of JavaScript is the prototype system. It is specified in a rather dense fashion in section 4.2.1 of the ECMAScript, 3rd Edition standard. Earlier we noted that every constructor has a prototype object, and that such a prototype object can be used to model the common parts of the objects that a constructor constructs.

In fact, every object in JavaScript has a prototype object, which is named `__proto__`, including objects used as prototype objects. This makes the following code perfectly valid:

```
MyConstructor.prototype.__proto__.value = 5;
```

Such a set of prototypes, whether explicitly stated or not, is called a prototype chain. The chain comes to an end with the `Object.prototype` object, which has no prototype of its own. All elements of the chain contribute properties to a final constructed object. The chain is an ordered list of its elements, so properties are contributed in that order. One effect of this order is *shadowing*, in which properties contributed later can override the values of properties with the same names contributed earlier.

Prototypes can be used to implement object-oriented inheritance. The simplest way to do this is to change a prototype to a new "base class," that is, to a new prototypical object. From Listing 5.4, this could be done as follows:

```
function Family() { };             // Some constructor.
Family.prototype = new Parents;    // New base "class".
```

The prototype property can also be set from within the constructor, providing that enough variations on shared, unshared, and cloned parts of the chain allow many tricks. Even multiple inheritances can be implemented, but they are applied rather awkwardly. Before attempting to work with multiple inheritance, read carefully section 4.2.1 of the standard. For most purposes, this complexity should be avoided. One base class as shown here is usually enough. Give it methods but no other properties.

Mozilla applications occasionally use the prototyping system to create custom objects. This is mostly done when more than one object of a given type is required. It is verbose to create explicitly several objects using object literals, so an object constructor is created once and used for each object needed instead.

5.4 LANGUAGE ENHANCEMENTS

Mozilla's JavaScript language has a few features beyond ECMA-262 Edition 3.

5.4.1 Mozilla SpiderMonkey Enhancements

Mozilla's interpreter, SpiderMonkey, is relaxed about what constitutes a valid statement:

```
x++ % y++;
```

This is standards-compliant behavior but other JavaScript implementations do not support it.

Perhaps the most useful standards extension is a feature that allows JavaScript `get` and `set` methods to be attached to an object property. In other words, when the property is read or written to, a whole function runs as a side-effect. The function must implement the normal action that occurs when a property is read or written as well. Listing 5.6 illustrates these extensions at work.

Listing 5.6 Creating an active property on a JavaScript object.

```
var obj = {
  get foo ()    { effect1(); return this._real_foo; },
  set foo (val) { effect2(); return this._real_foo = val;},
  _real_foo: "bar"
  }

var x = obj.foo;  // effect1() runs, and x = "bar"
obj.foo = "zip";   // effect2() runs, and _real_foo = "zip"
```

In this example, the `_real_foo` property stores the true state of the `foo` property, which really only exists as an interface. In ECMAScript standards

terminology, these special functions allow you to implement the `[[Get]]` and `[[Put]]` operations on a property with functions of your own design.

Mozilla also provides some management functions that can be used on the same getters and setters. These functions are

```
_defineGetter__("property-name", function-object);
_defineSetter__("property-name", function-object);
_lookupGetter__("property-name");
_lookupSetter__("property-name");
```

The first two functions have the same effect as the syntax noted in the previous example. The second two functions return the function installed as either the getter or setter.

Mozilla also supports a `__parent__` property that reflects the `[[Scope]]` internal property of function objects. Mozilla's JavaScript also provides this property on all native and host objects.

Finally, Mozilla supplies a `__proto__` property that can be used to set explicitly or read the internal `[[Prototype]]` property described in the standard.

5.4.2 ECMA-262 Edition 4 Enhancements

Edition 4 of ECMAScript, currently in draft, is an overhaul of the language with numerous goals. In general, it is an attempt to broaden the language to include full object-oriented support, packages, better interfaces to other languages, and numerous other small enhancements. It replaces the prototype system with classes and has a strict mode that requires all code to match Edition 4. It also has a nonstrict mode for backward compatibility.

Few of the proposed features of Edition 4 are present in SpiderMonkey 1.5, but the standard is close to being finished. A starting point for Edition 4 issues is `http://www.mozilla.org/js/language/`.

A simple Edition 4 extension is support for constants:

```
const varname = 5;
```

Mozilla has a `javascript.options.strict` preference, which can be set to `true`. This also appears in the Debug part of the preferences dialog box if a nightly build of the platform is used. This option is not the same as edition 4 strict mode; it is just additional checks against normal edition 3 code. For developers, it is a recommended preference.

SpiderMonkey 2.0 is planned to be equivalent to Edition 4 of the standard.

5.4.3 Security Enhancements

The Mozilla Platform runs interpreted scripts on the user's computer, which is a potential security problem. Features in the platform address security prob-

lems in a variety of ways, and a couple of these apply to JavaScript. For a more general discussion on security, see Chapter 16, XPCOM Objects.

A script will generate an error if processing exceeds a very large (4194304) number of backflow instructions. Roughly, one backflow instruction is any script processing that is a function return, a new loop iteration, or an aborted scope due to an exception. Such processing is seen as a denial-of-service attack that prevents user input from being processed. In a browser, such an error results in a popup warning to the user, who can then stop the script.

It is still possible to do lengthy, CPU-intensive processing in a script—just divide the work into chunks, and submit each chunk via the `setTimeout()` or `setInterval()` methods. This creates a scheduling opportunity between chunks that allows user input to be received and in turn prevents popup warnings.

Mozilla's general strategy on security is the "same origin" principle. A script loaded from one Web site (domain and path) may not interact with a document or resource from another Web site (less qualified path). This keeps Mozilla windows separate from each other. Attempts to set variables in windows where access is disallowed by security results in an exception (an error).

Scripts stored inside the chrome are released from security restrictions. The largest restriction is that the `Components` property of the `Window` or `ChromeWindow` object is fully accessible. This property is the entry point to all of Mozilla's XPCOM components. None of these components are directly accessible from outside the chrome. Some of these objects underlie other functionality that is independently made accessible. An example is the object that implements the `Form` object that is made available in HTML documents.

JavaScript scripts may be signed using digital certificate technology. In that case, the script may be granted capabilities equal to those of chrome scripts.

5.5 MOZILLA'S SCRIPTABLE INTERFACES

Like a hand in a glove, a JavaScript interpreter and the host software in which it is embedded go together. This topic describes the host software's contribution to that partnership.

As for C, the JavaScript language has no input or output operations. All input and output must be done via host objects. At least a trivial piece of host software is required to provide those host objects.

If basic input/output (I/O) features are provided, then a script can draw in other scripts from elsewhere. In theory, such scripts can be used to create very large programs. This is similar to Perl's module environment, but it is not common practice in JavaScript. In JavaScript, the expectation is that most of a program's functionality will be implemented by host objects. Because of this expectation, learning the language syntax and semantics is trivial compared to learning typically extensive object libraries.

Host objects are a JavaScript concept. Although such things appear as objects inside a script, the host software services behind those objects needn't be objects in turn. In Mozilla's case, scriptable functionality provided by the platform consists of sets of interfaces. Many of these interfaces are indistinguishable from objects, and many of these interfaces are implemented as C++ objects. They may also be implemented in pure JavaScript. Such things are called interfaces just to make the point that they formally and precisely expose features of the platform. The concept of interface in Mozilla follows Java's and Microsoft COM's concepts of interface—a selection of features that an object might provide, a selection that is not necessarily all of the object's functionality.

5.5.1 Interface Origins

All host objects are the same from a script's point of view, but host objects come from different places. Technologies that contribute host objects include:

- **World Wide Web Consortium standards.** The W3C DOM standards describe extensive programming interfaces to XML documents. These interfaces give scripts almost total control over document content.
- **Application and Browser Object Models** (AOM and BOM). In addition to DOM objects, the Mozilla Platform provides further objects that are available when an HTML or XUL document is displayed in a window. Some of these objects are supported by other Web browsers; some are not.
- **XPConnect and XPCOM.** This system is unique to the Mozilla Platform and provides access to all the components that make up that platform. Those components do much of the processing that goes on inside an application. Components also represent a library or toolbox of tools useful for standard programming problems, like access to sockets.
- **XBL bindings.** The XBL markup language, described in Chapter 15, XBL Bindings, combines an XML document, JavaScript, and a style rule to add scriptable interfaces to an XML document. Although these interfaces are specified in files outside the Mozilla Platform, logic in the platform is responsible for parsing them and tying them to scriptable objects.

These interfaces are all combined in the Mozilla environment. In particular, they can all be used inside a single window, which represents the root global variable of JavaScript's object system. Together they make for a rich programming environment.

Mozilla also has smaller scale JavaScript support. Some aspects of the platform, such as the preference system and the install system, have their own, independent JavaScript environments. These environments include a very small number of host objects dedicated to a specific purpose. The interpreted scripts that use them are cut off from the rest of the platform and its services. This chapter calls these environments *island interfaces.*

5.5.2 W3C DOM Standards

The Document Object Model standards are numbered starting from 0 (zero) and are designed to be independent of any particular piece of software. Mozilla supports all of DOM 0, all of DOM 1, most of DOM 2, a little of DOM 3, and numerous nonstandard extensions. Non-Mozilla Web browsers, like Internet Explorer 6 and Opera, support at least DOM 0 and DOM 1. There are many programming libraries that can be used to add DOM 1 support to an application that must process XML content.

Where Mozilla implements a DOM feature, that feature is exactly the same in Mozilla as in the standard. It is very straightforward.

The downloadable PDF (Portable Document Format) versions of these standards are good enough that they can be used as electronic help while programming. All versions from 1 onward are available at `www.w3.org`. To save not-yet-final drafts, which are typically available only in XHTML, use a recent version of Mozilla and save the file as "Web page, complete." Displaying such XHTML files saved locally might be slow if you are offline. If that happens, choose File | Work Offline for an immediate solution.

The appendices in these standards include a JavaScript binding for each interface, but those bindings are quite wordy. The IDL syntax used throughout those documents is close in syntax to JavaScript. It can be read and trivially translated into JavaScript scripts. That is the recommended way to go.

If you can understand the IDL syntax in these standards documents, then you are perfectly placed to understand the XPIDL files that describe Mozilla's XPCOM components. We will discuss those components in more detail shortly.

5.5.2.1 DOM 0 Mozilla supports DOM 0 in HTML documents only. Many objects available in XUL are inspired by equivalent DOM 0 objects, so DOM 0 should be used as a guide that says what to expect in XUL.

DOM version 0 has no W3C standard and represents early JavaScript support for HTML only. The contents of DOM 0 are roughly equivalent to the scriptable features of version 3 Web browsers. The best documentation on those features is available on Netscape's DevEdge Web site—`http://devedge.netscape.com`—where historical documents are still retained. Look for guides to early versions of Navigator under "archived information."

DOM 0 applies to HTML, not XML or XUL. As noted earlier, it provides a standard set of precreated objects that are made available to the scripter when an HTML document is loaded. The most famous example is the `Image` object that has been used extensively to implement the DHTML image rollover technique. See any book on intermediate Web page design.

DOM 0 objects often exactly match one HTML tag; for example, the `<input>` tag has an `InputElement` or `FormElement` object. In XUL, use the HTML tag nearest the XUL tag as a guide to the DOM 0 object for that XUL tag.

5.5.2.2 DOM 1 Mozilla supports DOM 1 completely.

Of the many features that the DOM 1 standard provides, the interfaces named `Node`, `NodeList`, `Element`, and `Document` provide 90% of the needs of scripts. The important properties and methods of these interfaces are listed in Table 5.3.

Table 5.3 Most useful features in the DOM 1 standards

Property or method	DOM interface	Use
parentNode	Node	The parent tag of the current tag
childNodes	Node	All the child tags of the current tag, as a NodeList
firstChild	Node	First child tag of the current tag
nextSibling	Node	Next child tag of the current tag's parent tag
Node insertBefore(aNode, existingNode)	Node	Add a tag or text before the stated child tag or text
Node removeChild(existingNode)	Node	Remove a tag or text from the immediate children of this tag
Node appendChild(aNode)	Node	Add a tag or text to the end of the children of this tag
String getAttribute(attString)	Element	Return the value of an existing attribute on this tag, or ""
void setAttribute(attString, value)	Element	Set attribute="value" text for this tag
void removeAttribute(attString)	Element	Remove the given attribute, if any, from this tag
Boolean hasAttribute(attString)	Element	Report existence of the stated attribute
Element createElement(tagString)	Document	Create a tag; the tag exists separate from the current document
Node createTextNode(value)	Document	Create text; the text exists separate from the current document

Table 5.3 Most useful features in the DOM 1 standards (Continued)

Property or method	DOM interface	Use
Element getElementById(idString)	Document	Get the tag with the specified id
NodeList getElementsByTagName(tagString)	Element, Document	Get all the tags with the specified tag name
Node item(i)	NodeList	Get the ith node in the list
length	NodeList	Return the number of nodes in the list

Because `Element` and `Document` interfaces are also `Nodes`, all the properties and methods noted against `Node` apply to those other interfaces.

The DOM1 standard ignores the history of DOM 0 and starts out afresh. It consists of two parts.

The first part is very general and applies to all XML documents, including MathML, SVG, and XUL. It also applies to XHTML and HTML. It breaks a text document up into a treelike structure, much as you might see in the "auto-indented tag view" of an HTML editor. Mozilla will display such a structure if you load an XML document whose XML application type it doesn't understand. This structure is a containment hierarchy.

Everything in DOM 1 part 1 is a Node object. Documents, tags, tag attributes, and tag contents are all objects that are subtypes of Node. The document tree is a tree of Nodes, with the top node being the Document object. The DOM 1 part 1 tree knows nothing about specific tags (such as XUL tags). So there is no `<button>` object. There is only a generic Element object, used for all tags. Processing is the same for XML, HTML, or XUL.

There are two ways to navigate such a tree. The first way uses a query system. You can use the DOM 1 equivalent of Microsoft's `document.all()`, or a Mozilla enhancement:

```
document.getElementsByTagName("button"); // all XML
document.getElementsByAttribute("class", "*"); // only XUL
```

The `"*"` parameter indicates that all values for the attribute are allowed. The other navigation technique is to step through the tree explicitly, using data structures and algorithms. For example, this code reaches down two levels into a document:

```
document.firstChild().firstChild().getAttribute("type");
```

DOM 1 part 1 also provides a complete set of features for inserting, updating, and deleting XML content, but this must be done by constructing objects, not by submitting fragments of XML content. Constructing objects can

be a lengthy and unwieldy process if the content changes are significant. Mozilla has a single enhancement to make these jobs easier: the `innerHTML` property. This improvement is inspired by Internet Explorer and allows XML content to be added directly to a tag's content. Despite the name, this works for XML and XUL as well as HTML:

```
tag_object.innerHTML = '<box><button value="On"/></box>';
old_content = tag_object.innerHTML;
```

Mozilla does not support the Internet Explorer extensions `innerText`, `outerHTML`, or `outerText`.

The second part of DOM 1 is intended to be HTML-specific. DOM 1 part 2 provides handy methods and attributes for a HTML document. If a tag is an HTML tag, then the matching node in the DOM tree will have properties matching the HTML attributes of that tag. So a `<FORM>` tag is a node in the tree with `action`, `enctype`, and `target` properties, among others.

DOM 1 part 2 provides two more ways to access tag collections:

```
document.getElementsByName('myform1');
document.getElementById('testid');
```

`getElementById()` is also supported for XUL and is used everywhere. `getElementsByName()` is not recommended, not even for HTML, because the name aspect of HTML is being dropped from the standards.

To summarize, DOM 1 breaks down a whole XML or HTML document into a big data structure that's fully modifiable. If the document is an HTML document, then attributes are reflected as object properties for known tags. In either case, collections of similar tags can be extracted from the data structure simply. You can call DOM 1 the "canonical" (authoritatively accepted) interface.

5.5.2.3 DOM 2 Mozilla almost fully supports DOM 2.

The DOM 2 standard fills in a number of gaps in the DOM 1 standard. It is a superset of DOM 1. DOM 1 is written in one document, but DOM 2 is written in six documents. They are:

1. DOM 2 Core
2. DOM 2 HTML
3. DOM 2 Events
4. DOM 2 Style
5. DOM 2 Traversal and Range
6. DOM 2 Views

The DOM 2 Core and HTML documents are the same as the DOM 1 document. Some small fixes are present, but recent editions of the DOM 1 standard also have these fixes. A minor enhancement remains. The DOM 2 Core/HTML standards call everything in DOM 1 the "fundamental interface." The "extended

interface" in DOM 2 provides several nice-to-have features that are allowed only in XML, not HTML. These features are interfaces that give access to

- Entity references like `&`
- Processing instructions implemented by languages such as XSLT
- The `<!ENTITY>` declarations in the document's defining DTD

The DOM 2 Events standard also provides the well-known Web browser events. In addition to the mouse events (`'click'`) and HTML-specific events (`'submit'`), there are general XML events (`'DOMFocusIn'`). There are also events that fire when the document tree is changed or "mutated." No key press events (`'keydown'`) are documented in DOM 2 Events. The DOM 2 Events standard also says how events travel around the document tree. Events are discussed in detail in Chapter 6, Events.

The DOM 2 Style standard compliments the CSS2 standard, giving JavaScript access to styles. In DOM 1, the only thing you can do is update the style attribute of a tag (an inline style). The DOM 2 standard lets you work with style rules as though they are objects, not just text content. It also lets you examine and change the fully cascaded style of any given tag. The following code

```
document.getElementById("greeting").style.color ="red";
```

changes a piece of text to red, regardless of what mix of external or inline styles exist. This interface, however, *is not supported in XUL*. It is supported in HTML only.

The DOM 2 Traversal and Range standard caters to user cut 'n' paste actions. This feature requires careful examination of the DOM tree to see what tags the user has selected. For this examination, better ways of stepping through the tree are needed. That is the Traversal part, where new objects called `Iterators` and `TreeWalkers` are specified. The Range part creates a collection of tag objects matching the tags selected with the mouse. In Mozilla, cut 'n' paste, where it exists, is done using this standard. The Range standard also does a little sophisticated processing of tags. It can chop tags in two, insert content between tags, and perform a few other conveniences. The ability to insert content more easily into an XML document while being guaranteed that the document will remain well formed is a powerful feature.

The DOM 2 Traversal and Range standard is mostly intended for building new software. This is even more true for the DOM 2 Views standard. Its purpose is to provide interfaces that let the DOM tree display in several windows at once. The most obvious use of this is in a Web page editor like HomeSite, where there is a user view, a source code view, and a document structure view, all visible at once. Updating the underlying model via one view updates the others. DOM 2 Views explains the interfaces needed so that you can build such an editor.

Of the six DOM 2 documents, two are from DOM 1, two are very useful (events and styles), and two are more obscure. You might call the DOM 2 stan-

dards the "user interaction" standards in the sense that they provide features that accommodate user input and page display.

5.5.2.4 DOM 3 Mozilla supports only a little of the DOM 3 standards.

The DOM 3 standards are very new, but they are near completion as this book goes to print. Much of the standardized functionality is not available in browsers yet. There are five parts to DOM 3:

1. DOM 3 Core
2. DOM 3 Events
3. DOM 3 Load and Save
4. DOM 3 Validation
5. DOM 3 XPath

DOM 3 Core makes minor changes to the DOM 2 Core. Two things are significant. This DOM 3 example

```
document.getInterface("SVG");
```

allows other standards implemented in the browser to be revealed to script use. This is equivalent to `use` in Perl. If SVG support in the browser provides an SVG interface to JavaScript, and it's not visible to you by default, this is the way to make it visible.

The second significant change is this example:

```
document.setNormalizationFeature("format-pretty-print");
```

which gives you additional control over an XML processor. The processor engine makes many decisions when loading a XUL or an HTML file, like how whitespace is handled and when errors are reported. As soon as DOM 3 Core is implemented, you will be able to control these decisions.

DOM 3 Events supplies the keypress events (now called text events) missing from DOM 2 Events. Most of the PC keys like Alt, Caps Lock, and F12 are specified. Multiple event handler listeners are also new. This feature allows two event handlers on the same tag to fire when just one event occurs. This is the only part of DOM 3 that Mozilla comes close to supporting.

The DOM 3 Abstract Schemas and Load and Save standard, which bears quite a lengthy name, is another two-part standard. The Abstract Schemas part provides full read-write access to the DTD of an XML document. This is more likely to be useful in an XML database than in a browser. The Load and Save part describes how to go from XML text to a DOM tree and back again. It is a serialization or encoding standard. This is the standards-based solution to the problem that the `innerHTML` property solves.

DOM 3 Validation adds a constraint system to document manipulation operations. Using the other DOM standards, it is possible to change a document so that it no longer conforms to its document type. For example, it is possible using the DOM 1 interfaces to add a `<circle>` tag to the content of an

HTML document, even though `<circle>` is not an HTML tag. The DOM 3 Validation standard checks the type definition of a document (either a DTD or an XSchema definition) every time one of the other DOM interfaces is used to modify the document. If the modification would break the rules implicit in the document type, the change is not allowed. It acts as a type of police officer.

Finally, DOM 3 XPath brings yet another method of stepping through the DOM tree to JavaScript, based on the W3C XPath standard. DOM 3 XPath exposes the XPath system to JavaScript.

You might call the DOM 3 standards the "heavy engineering" DOM standards. It will probably be a long time before the DOM 3 standards are ever used for publicly accessible XML pages. On the other hand, they are of vital interest to application developers working on high-end tools like WYSIWYG (What You See Is What You Get) document processors. Perhaps one day Mozilla will have full support.

It is unlikely that a DOM 4 set of standards will appear any time soon, if ever.

5.5.2.5 DOM Compatibility Ticks The DOM standards supply a `DOMImplementation` interface, which contains a `hasFeature()` method. This method can be called from JavaScript to reveal the standards that Mozilla claims to support. A suitable line of script is

```
f = document.implementation.hasFeature("XML","1.0");
```

Some past debate on `hasFeature` is recorded in Mozilla's bug database. Table 5.4 shows the results for Mozilla 1.0 and 1.4. "FALSE" is capitalized in the table only for ease of reading.

The results are the same for both XUL and HTML scripts, which makes reporting of the "HTML" feature incorrect when in an XUL document. XUL also has support for its own so-called `KeyEvents`, but that is not yet reported by `hasFeature()`.

A result of `false` for a particular feature is not a disaster. If a standard is completely implemented except for a single item, then `false` is the correct value to report. Finished parts of the implementation may still be used if they are known to exist. Knowing that they exist is a matter of reading and research.

If a feature has DOM support, the relevant DOM interfaces are available to script. Interfaces are used by programming languages. There are, however, other ways to interact with the services of a browser. Stylesheets and XML are two examples. A feature may have stylesheet support but not DOM support, or vice versa. `hasFeature` reports only on DOM support.

Some browser features may eventually make their way into the DOM standards. The Mozilla 1.0 features most likely to receive standards attention are drag-and-drop support, keystroke events, scrolling events, and key shortcuts.

Separate from HTML DOM 0 support is XUL DOM 0 support. XUL DOM 0 has no standard or central design document yet. It is undocumented except for remarks about specific tags in this book.

Table 5.4 DOM standards support for Mozilla 1.0 and 1.4

Feature string	Version string	hasFeature reports
HTML	1.0	true
XML	1.0	true
Core	2.0	true
HTML	2.0	true
XML	2.0	true
Views	2.0	true
StyleSheets	2.0	true
CSS	2.0	true
CSS2	2.0	true
Events	2.0	true
UIEvents	2.0	FALSE
MouseEvents	2.0	true
MouseScrollEvents	Not a W3C feature	true
MutationEvents	2.0	FALSE
HTMLEvents	2.0	true
Range	2.0	true
Traversal	2.0	FALSE
Xpath	3.0	true
All other DOM 3 features	3.0	FALSE

Mozilla specifies object interfaces using its own XPIDL language. The XPIDL files that programmers need when working with XPCOM objects are also used for DOM objects. Because XPIDL and IDL syntax are so similar, it is trivial to translate from one to the other. Mozilla's XPIDL definitions, however, located in the source directory `dom/public/idl`, are the official word on DOM and DOM-like interface support. They include restatements of both the DOM IDL definitions and the available XUL DOM-like interfaces. They also state Mozilla-specific enhancements to those standards. Beware that some implemented event handlers do not appear in these files.

5.5.3 Mozilla AOMs and BOMs

DOM objects are about XML documents. Some of these objects are made available when an XML document loads because they're handy to have. The `Document` object, and the document variable that points to it, is an obvious example.

There are other objects you might want to script that have nothing to do with documents. The window the document resides in is an obvious example. You might want to change the text of its titlebar, make a modal dialog box appear, or change the document that the window displays.

These nondocument objects are part of the Browser Object Model that exists in most Web browsers. This object model is small when compared with the DOM. It consists of a few objects only in a small hierarchy. The top of the hierarchy is a so-called `Window` object that represents the desktop window that the document is displayed in. The window variable refers to this object. The small BOM hierarchy is very familiar to Web developers and contains objects such as

```
navigator screen history location frames
```

This book doesn't dwell on these objects much (any Web-oriented JavaScript book covers them extensively). There is some further discussion on the window object in Chapter 10, Windows and Panes.

The Mozilla Platform is not just a Web browser but also a basis for applications. When XUL documents are being displayed, it makes more sense to talk about an Application Object Model rather than a BOM. XUL windows also start with a window variable that is the top-most object in the application hierarchy, but the matching object type is a chrome window, not a window. Figure 5.1 shows the AOM hierarchy for a chrome window.

Each word in Figure 5.1 is an object property that holds an object, although occasionally one may hold `null` or an empty string instead. The hierarchy that starts at window is always present. The hierarchy that starts with Element exists for every DOM 1 Element object. Dark lines exist for both XUL and HTML; lighter lines exist only for XUL. A dashed line means that the object is available only when security barriers are lifted. Where these terms match names in the Web browser BOM, the meanings are the same. The other terms are discussed throughout this book.

A brief summary of the XUL-specific AOM objects goes like this: `arguments` is passed in when a window is open; `content` points to the document held in a `<browser>` tag in the XUL page; `controllers` is a set of command controllers, either for the window or for one tag; `commandDispatcher` is also part of the command system; `focus` controls the focus ring; `boxObject` contains all the details of a single tag's frame, for both ordinary XUL tags and special XUL tags like `<scrollbox>`; `builder` exposes a tree builder for the `<tree>` tag; and `database` and `datasource` are part of the XUL template system.

In addition to these objects, Chapter 10, Windows and Panes, explores some of the less obviously exposed objects. Many of these objects are associated with the overall mechanics of displaying a XUL or HTML document.

The XUL window object is very similar to the HTML window object. It has a history property, for example. This arrangement is quite misleading because it implies that these properties do something. They don't do anything.

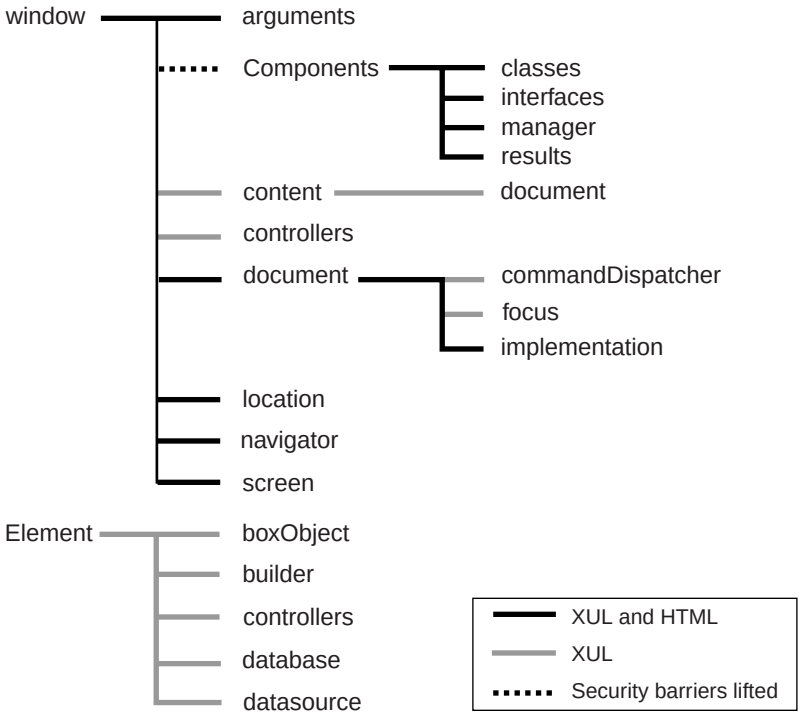


Fig. 5.1 Mozilla's XUL Application Object Model.

They exist because the many scripts and XUL documents that together make up a browser application expect them to be in place. Under normal conditions (a Classic Browser), these properties rely on extra scripting support to function. That scripting support isn't present in a plain XUL window, unless your XUL is designed to reimplement the features of a browser. Ignore these HTML-like properties.

As for HTML, all the implemented DOM interfaces are directly available to XUL. The properties and methods of the `window.document` object are the entry points to those interfaces.

Both HTML and XUL windows contain one special property: the `Components` property. That property is the starting point for the majority of Mozilla's unique interfaces. These interfaces are discussed next.

5.5.4 XPCOM and XPConnect

The DOM standards provide interfaces that make content accessible to scripts. By comparison, XPConnect and XPCOM provide interfaces that make the building blocks of the Mozilla Platform itself accessible to scripts.

XPCOM is entirely internal to the Mozilla Platform. XPConnect is the glue that turns XPCOM-compliant objects and their interfaces into JavaScript host objects. Chapter 16, XPCOM Objects, describes application-oriented use of this system extensively. This chapter just gives an overview of these two technologies.

From the application programmer's perspective, the XPCOM system appears mainly as the `window.Components` object (see Figure 5.1). The `classes` and `interfaces` properties of this object are lists of all the available XPCOM Contract IDs and interface names. The `manager` property is the XPCOM component manager. The `results` property is a list of all the possible exceptions that might be thrown and result values that might be returned. Results are 32-bit numbers.

These lists could in theory change dynamically, but that is not done in the standard platform. The standard platform registers all the known components at startup time, and a large part of that registration overhead is performed only once the first time the platform starts. The set of components registered can be recalculated by running the tool `regxpcom` (`regxpcom.exe` on Microsoft Windows). That tool is provided with any standard platform installation bundle. Using `regxpcom` is necessary only when new components (i.e., whole modules, not merely objects; see Chapter 16, XPCOM Objects) are created or installed.

From the application programmer's perspective, the XPConnect system is all but invisible, making the interfaces of C/C++-based components available to JavaScript. In order to do this, the XPConnect system relies on a set of type libraries (with `.xpt` extensions) that are stored in the `components` directory under the platform installation area. XPConnect also provides script access to browser plugins, to the DOM interfaces, and to Java and Java applets. Although these things may seem unrelated to XPCOM, in the end they are all XPCOM components.

All XPCOM components that provide more than one XPCOM interface should implement the `nsISupports` interface. This interface provides the `QueryInterface()` method, which can be used to find other interfaces supported by an object.

5.5.4.1 Finding XPCOM Components

The Mozilla Platform is large, and many parts of it have been separated into building blocks: Each is an XPCOM component that has some identity of its own. Not all of Mozilla is so separated; there are still bits that sit anonymously inside the platform.

There are at least a thousand XPCOM components, each with one or more interfaces. That is an overwhelming number and is equivalent to a huge class library. This large number presents problems for the newcomer. At first glance, Mozilla functionality seems like a fragmented mess. It is not. There is simply too much to learn in one sitting, and structure is not obvious. Finding documentation on all the components of Mozilla is also problematic. At two

pages per component, that's two thousand pages. Such huge books might exist one day, but they don't as this goes to print.

To find a suitable component, you need to look. The thing you ultimately want is an object that is used for a specific task. Not all objects need to be accessed directly via their XPCOM interfaces. Many objects exist in other forms. Before diving into XPCOM, ask yourself: Could the needed object appear in any of these other forms?

- As a standard JavaScript object (e.g., Date or Math). If so, look at the ECMAScript standard.
- As a piece of the DOM. If the object relates to any kind of XML document fragment, it will be covered in the W3C DOM standards.
- As a part of the BOM. A Web-based JavaScript book will tell you if the object is a standard feature of HTML scripts. The AOM in this chapter advises of standard XUL scriptable objects.
- As an XBL binding. Nearly all objects that have a matching XUL tag also have an XBL binding. The binding definition can be read from the file `toolkit.jar` in the chrome.
- As a floating object service. Some objects are separate from the DOM but still available for immediate use (e.g., Image and Option [HTML]; XMLHttpRequest and SOAPService [HTML and XUL]).

If there is no choice but to look for an XPCOM object, then here is how to proceed. An object is manufactured from an XPCOM interface and an XPCOM component that implements that interface. So you must locate these two things. Interfaces have names; components have Contract IDs (which are also names). To find these things, any of the following strategies will do:

- Look in the index of this book under a keyword that matches your problem area. Many component-interface pairs are recommended throughout this book.
- Download the summary reports from this book's Web site (at `www.nigelmcfarlane.com`), and look through those.
- Download the XPIDL bundles from this book's Web site (at `www.nigelmcfarlane.com`) or from the mozilla.org source, and look though those.
- Explore the chrome-based applications that come bundled with Classic Mozilla. That code contains many tested and proved examples.
- Search the mozilla.org organization's resources, like Web sites, newsgroups, and IRC. Well-stated questions posed in those fora often attract quality responses.

Of all these strategies, reading this book and acquiring a copy of the XPIDL files and other reports is the most immediate solution.

In the end, most, but not all, useful interfaces start with this prefix:

```
nsI
```

and all components' Contract IDs start with this prefix:

```
@mozilla.org/
```

As soon as you have a component-interface pair, the object can be created as follows.

5.5.4.2 Using XPCOM Components Here is a simple example of using a component. The Perl language has the `'require'` keyword, which allows one Perl script to load in the contents of another script. JavaScript has no such equivalent, but Mozilla has a component to do the job. Listing 5.7 illustrates.

Listing 5.7 Inclusion of JavaScript scripts using a Mozilla XPCOM component.

```
var comp_name = "@mozilla.org/moz/jssubscript-loader;1";
var comp_obj = Components.classes[comp_name];

var if_obj = Components.interfaces.mozIJSSubScriptLoader;

var final_obj = comp_obj.createInstance(if_obj);

final_obj.loadSubScript("file:///tmp/extras.js");
```

This script is equivalent to running `eval()` on the contents of the file `/tmp/extras.js`.

The two pieces of information needed to create an object are the component name (`comp_name`, a string) and the interface to use (`if_obj`, a property name). After the object is retrieved (`final_obj`), its methods can be put to use. In this case, the `loadSubScript()` method imports the contents of the file `extras.js` as though it were a string processed by `eval()`.

Typical of a Mozilla component, a line of documentation in the XPIDL definition for `mozIJSSubScriptLoader` states that the loaded JavaScript must be located on the local computer. Brief and sparse documentation is usual for Mozilla components.

5.5.4.3 Object Creation Alternatives In Listing 5.7, the `createInstance()` method was used to create the object. That is one of two main alternatives. The other main alternative is to use `getService()`. The `getService()` method is used when a given XPCOM component is designed to provide a single instance of an object only (the object is a singleton). Such objects may contain static or global information that cannot be coordinated across multiple object instances. In Listing 5.7, the equivalent line of code that uses `getService()` would be

```
var final_obj = comp_obj.getService(if_obj);
```

How do you tell which of `createInstance()` or `getService()` is required? If the Contract ID or the interface name contains the word *service* (any capitalization), then use `getService()`; otherwise, use `createInstance()`.

The `createInstance()` and `getService()` methods also have a zero argument form. If no argument is supplied, then the object created or returned has the `nsISupports` interface. That interface can then be used to acquire whatever other interface might be needed. We can modify Listing 5.6 to create an example of this use for `createInstance()`:

```
var anon_obj  = comp_obj.createInstance();
var final_obj = anon_obj.QueryInterface(if_obj);
```

If several objects of the same kind are to be created, then it is possible first to create a constructor, and then to use that constructor object to create the individual objects. This code replaces the first four lines of Listing 5.7:

```
var Includer = Components.Constructor(
                  "@mozilla.org/moz/jssubscript-loader;1",
                  "mozIJSSubScriptLoader",
                  null
                );
var final_obj = new Includer();
// var another_obj = new Includer()
```

Note that in this case the interface name is spelled out as a string. The third argument of the `Constructor()` method is an optional string. If stated, it is the name of a method to call as the initialization method when an object instance is created. Any arguments passed as arguments to the `Includer` object constructor will be passed to this initialization method.

Finally, it is possible to create objects with XPCOM interfaces directly in JavaScript. This can be done only for simple interfaces that have no special processing of their own. Continuing the variations on Listing 5.6, we can create an object using the code of Listing 5.8.

Listing 5.8 Creation of a JavaScript object with XPCOM interfaces.

```
var final_obj = {

  // interface nsISupports
  QueryInterface : function(iid) {
    var Ci = Components.interfaces;
    if ( !iid.equals(Ci.nsISupports) &&
         !iid.equals(Ci.mozIJsSubSCriptLoader )
      throw Components.results.NS_ERROR_NO_INTERFACE;
    return this;
  },

  // interface mozIJSSubScriptLoader
  loadSubScript : function (url) {
```

```
    // code to load a script goes here
    return;
  }
};
```

This object supports both the `nsISupports` and the `mozIJsSubScriptLoader` interfaces. If it were certain that no calls to `QueryInterface()` would ever be made on this object, then the `nsISupports` part of the object could be left off. Of course, this object has a major problem: How can it properly implement the second interface? It would need to contain a long piece of code that opens a file system file, reads the contents, and then calls `eval()` on that content. For that effort, you might as well use the existing implementation of the object. This by-hand construction of objects is sometimes highly useful. The most frequent use is in the creation of call-back objects used for observers and listeners—see Chapter 6, Events.

The `createInstance()` method is itself implemented by the `nsIComponentManager` interface; `getService()` is implemented by the `nsIServiceManager` interface.

5.5.4.4 Plugins and DOM Browser plugins have been scriptable since version 3.0 Web browsers. At that time, Netscape enhanced the version 1.1 NPAPI plugin standard to be scriptable, and hundreds of plugins were created to work with it. Mozilla still supports this standard, but very recent versions of both the platform and the plugins (especially Macromedia's Flash) are required if everything is to work without hiccups.

A series of scriptable XPIDL interfaces exists for plugins. Using these, individual plugins can be driven directly from JavaScript. The easiest way to handle plugins is still to use the HTML `<embed>` (deprecated) or `<object>` (recommended) tag and the DOM 0 plugins array. See Chapter 10, Windows and Panes, for a discussion on combining HTML and XUL.

A series of scriptable interfaces also exists for the DOM standards, so DOM interfaces can be retrieved using the `Components` object. There is little need for this, however, because those interfaces are automatically available on the window object, and on the DOM objects that are made and retrieved by the window object.

5.5.4.5 Java Web browsers have supported scripting Java applets since version 3.0; Mozilla also supports scripting Java. The architecture used in the Mozilla 1.x platform is different from that used in Netscape Communicator 4.x, but it is backwardly compatible for script writers.

In Netscape Communicator 4.x, the browser depended entirely on Java 1.1's security model. For that browser to make any security decisions, a request that Java examine a special class library provided by Netscape was required. Java and JavaScript were connected, and this was done by a technology called LiveConnect.

In Mozilla, this Java dependency is gone. Netscape manages its own security. No Java is required, and Java is no longer tightly integrated in the platform. There is no LiveConnect in the original sense; there is only XPConnect. Integration features provided by LiveConnect still exist, but they are buried in the code behind XPCOM. Mozilla's visible Java support now consists of a set of XPCOM interfaces like everything else, and Java support is considered to be similar in status to a plugin. The functionality of LiveConnect is still visible when Java and JavaScript objects are wrapped up for access in the language alternate to their origin.

Mozilla provides two pieces of technology that accommodate Java. The first is OJI (Open Java Interface). This is an API like the NPAPI (plugin) API that is supposed to work with any vendor's Java platform. XPCOM interfaces provide access to the OJI. The second piece of technology is a LiveConnect emulator for backward-compatibility. It also is an XPCOM interface, defined in the interface `nsIJRILLiveConnectPlugin`. This emulator handles Mozilla's C/C++ requests that were previously sent to Netscape 4.x's Java JVM.

As for plugins, the easiest way to use Java inside Mozilla is still to use an `<applet>` or `<object>` tag inside an HTML file. Again, see Chapter 10, Windows and Panes, for a discussion on combining HTML and XUL.

Some programmers come to the Web from a Java background. If you have a sophisticated Java environment and want to integrate it with Mozilla, tread cautiously. On the one hand, the most simple and obvious interactions between JavaScript and Java work fine. On the other hand, JavaScript, Java, and Mozilla are all quite complicated systems, and perfect interaction on every point is a large goal. Sophisticated interactions still have problems, and you're encouraged to study the remaining Java-related bugs in Mozilla's Bugzilla bug database carefully.

The best way forward is this: Stick to Sun Microsystem's Java implementation, and then use only version 1.4 or greater. A very recent version of Mozilla is recommended to keep the outstanding issues to a minimum.

Apart from some remarks in Chapter 10, Windows and Panes, that is all this book has to say on Java.

5.5.5 Scripting Libraries

Mozilla relies on the XPCOM system to draw in functionality from elsewhere, but a few small libraries written entirely in JavaScript exist. These libraries attempt to simplify particular aspects of the XPCOM system.

The most useful of these libraries are JSLib and RDFLib. RDFLib is really just a subset of JSLib. JSLib makes working with files and folders less painful. The RDFLib subpart makes working with RDF data sources less painful. This library creates JavaScript objects that simplify the XPCOM objects that they are based on.

This library can be downloaded from `http://jslib.mozdev.org`. Files in this library are written in ASCII, in UNIX format.

Beware that these libraries receive only occasional updates. Examine the headers when downloading them to see whether they have been recently maintained.

To use the library, include the top-level `jslib.js` file, which provides utility routines and constants. The most important of these routines is the `include()` function, which acts the same as Perl's `require` (or cpp's `#include`). Then include whatever other JSLib functionality is needed. The library contains nearly a dozen subdirectories, but the most significant ones are `io` (for file processing) and `rdf` (for RDF processing). Figure 5.2 shows the dependencies between these libraries.

Scripts with outgoing arrows require the targets of those arrows to be loaded first. Files with no dependencies are not shown. Clearly the library is divided into two main parts, which this book calls RDFLib and JSLib. There are two separate implementations of RDF file access: `rdf/rdfFile.js` and `io/RDF.js`. They present different interfaces for approximately the same task; `rdf/rdfFile.js` is the more extensive interface and uses subobjects.

The library uses the `window.Components.Constructor()` method to create object constructors for XPCOM components (see the section entitled "Object Creation Alternatives" for instructions describing how to do this). It also implements JavaScript object constructors using the JavaScript prototyping system. A prototype-based constructor is used to create a new JavaScript object. During the construction process, XPCOM object constructors are used to set properties on the new JavaScript object. Those properties hold one

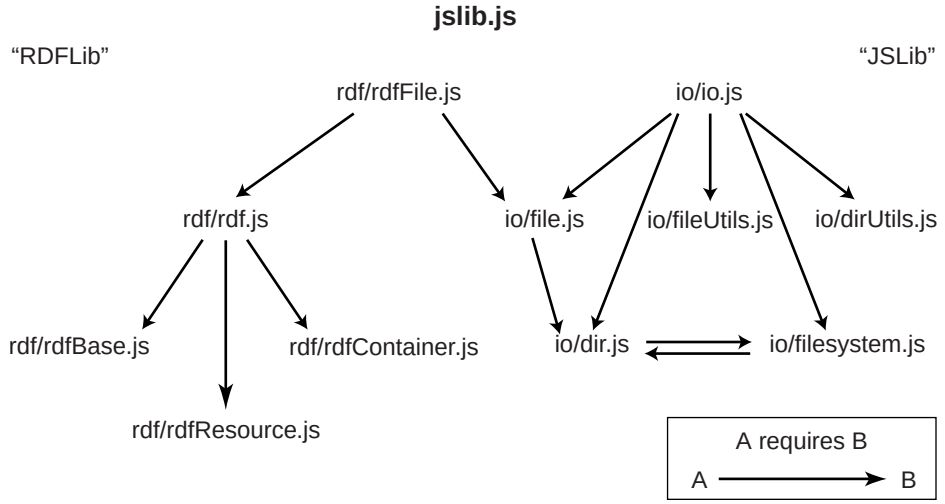


Fig. 5.2 JSLib and RDFlib file dependencies.

XPCOM object each. The final JavaScript object therefore contains (or uses) one or more XPCOM objects and is a wrapper or façade for those XPCOM objects. This is a simple and widely applicable technique.

Such new JavaScript objects contain error-checking, coordination, and translation code that makes the XPCOM components easier to use. They represent a simplification or specialist view of the XPCOM component system, intended for common tasks. Table 5.5 shows the objects created by this library.

In Table 5.5, a "data-source" argument is an `nsIRDFDataSource` object, not an `rdf:` URL. The JSLib library contains several other useful objects, but they are not as extensive or as complete as the objects in Table 5.5. The `Zip` object, for example, which is defined in the `zip/zip.js` file, performs only about half the work required to unzip a compressed file.

Most of the files in the JSLib have extensive documentation in the form of preamble comments. These preamble comments show how to use the objects supplied.

Table 5.5 Objects created by JSLIb and RDFLib

Constructor	Source file	Uses objects	Purpose
include() method	jslib.js		Include other .js files.
Dir(filePath)	io/dir.js	FileSystem()	Manage a folder on the local disk.
DirUtils()	io/dirUtils		Find install-specific directories.
File(filePath)	io/file.js	FileSystem()	Manage a file on the local disk.
FileSystem(filePath)	io/filesystem.js		Manage any file system entry on the local disk.
RDF(url, urn, xmlns, sync_flag)	io/rdf.js		Read and write a single special format RDF container in an existing RDF file.
RDFFile(file_url, urn, xmnls, sync_flag)	io/rdf.js	RDF()	Read and write a single special format RDF container in a local RDF file.
Socket()	network/socket.js		Set up and read/write a Domain socket.

Table 5.5 Objects created by JSLIb and RDFLib (Continued)

Constructor	Source file	Uses objects	Purpose
SocketListener()	network/socket.js		Handles asynchronous reads from a socket.
RDF(url, sync_flag)	rdf/rdf.js	RDFBase()	Read and write an RDF document fact-by-fact.
RDFBase(datasource)	rdf/rdfBase.js		Read and write an RDF document fact-by-fact, using an existing data source.
RDFContainer(type ownerURI, subjectURI, datasource)	rdf/rdfContainer.js	RDFResource()	Manage or create an RDF `<Seq>`, `<bag>`, or `<alt>` container using facts.
RDFResource(type, ownerURI, subjectURI, datasource)	rdf/rdfContainer.js	RDFBase()	Manage or create a URI that is useable as an RDF subject, predicate, or object.
Sound(url)	sound/sound.js		Play a sound from a URL.

5.5.6 XBL Bindings

JavaScript can also use interfaces supplied by XBL definitions. Those definitions are included in an XML file with the `-moz-binding` CSS2 style property. Such a binding attaches the definition to a tag or tags. Although XBL definitions can include content, it is also possible to create a definition that is just a set of functionality—an interface. Being able to add an arbitrary interface to an arbitrary tag is clearly a powerful feature.

If the DOM 1 standard is used to retrieve an `Element` object for a tag that is bound to an XBL definition, then the resulting JavaScript object will include the properties and methods implied by that definition. Those properties and methods can then be manipulated as for any host object. The platform automatically manages this task.

XBL definitions therefore make some DOM objects smarter by providing them with custom features. XBL definitions can also establish themselves as XPCOM components, so they are also accessible through XPConnect. The only good reason for doing this is if the XBL component needs to be accessed from C/C++ code.

Chapter 15, XBL Bindings, discusses XBL at length.

5.5.7 Island Interfaces

Mozilla has three scriptable island interfaces. These interfaces have their own JavaScript interpreter and their own global object. Because they are separate from the rest of Mozilla, they have their own AOM.

Mozilla's preferences system is the first of these islands. Its AOM consists of a single `PrefConfig` object that acts like a global object. It has only two properties, both of which are methods: `pref()` and `user_pref()`. The preference files, `prefs.js`, consist of repeated calls to these methods, but in actual fact, the full JavaScript language is available. Any use of this language availability is obscure at best. The preference system also presents two components to XPCOM. They are named

```
@mozilla.org/preferences-service;1
@mozilla.org/preferences;1
```

When these components are scripted from the normal Mozilla scripting environment, preferences can be changed, but the `prefs.js` files cannot be modified directly. Changes to these preferences do affect the regenerated `prefs.js` files that are written when the platform shuts down.

Mozilla's network-enabled component installation system is the second of these islands. Chapter 17, Deployment, discusses this environment and its AOM fully. This system also exposes XPCOM components to general scripting, but the interfaces provided are low-level and of little use to general tasks. They allow the XPInstall system to be notified when content is available from elsewhere (e.g., over the Internet) and accept that content for processing. These limited interfaces mean that XPInstall is good for installation tasks only.

`xpcshell` is Mozilla's third island interface. It is a program separate from the Mozilla Platform executable that isn't available in the downloadable releases. It is available in the nightly releases, or if you compile the software yourself (it requires the `--enable-debug` option). `xpcshell` is a standalone JavaScript interpreter that is used to test scripts, the SpiderMonkey interpreter, and the XPConnect/XPCOM system. Its AOM consists of a single global object and XPCOM-related objects. The most useful properties of the global object are listed in Table 5.6.

The `Components` array makes `xpcshell` a good place to test scripts that make heavy use of Mozilla components. Because `xpcshell` uses plain `stdin` and `stdout` file descriptors, it can easily be run in batch mode by automated testing systems. `xpcshell` is Mozilla's JavaScript equivalent of the Perl interpreter, except that it is little more than a test harness.

In theory, a script run through `xpcshell` could create and use enough components to build up a whole browser. In practice, the Mozilla executable sets up low-level initialization steps that the `xpcshell` doesn't use. Therefore, `xpcshell` is best left to simpler tasks.

Table 5.6 xpcshell global object properties

Property	Description
Components	The XPConnect/XPCOM components array, provides access to most components in the Mozilla Platform.
dump(arg)	Turns the sole argument into a Unicode string and sends it raw to stdout.
load(arg1,arg2,...)	Attempts to load and interpret the files (not URLs) specified.
print(arg1,args2,...)	Turn arguments to strings and sends them slightly formatted to stdout, using printf("%s ").
quit()	Stops xpcshell and exit.

5.6 HANDS ON: NOTETAKER DYNAMIC CONTENT

In this section, we'll add some scripts to the NoteTaker Edit dialog box. These scripts will modify the way the XUL document works after the platform displays it. In the process, we'll manipulate several types of objects discussed in this chapter.

5.6.1 Scripting <deck> via the DOM

The first NoteTaker change involves keyword content in the dialog box. This Edit dialog box actually supports two separate panes. Each panel is represented by one of the <toolbarbutton> tags, "Edit" and "Keywords." We'd like to display the dialog box with one or the other of the panes visible. Because the content of the keyword panel isn't yet decided, we'll just use a placeholder.

To support this two-panel system, we'll use a <deck> tag. The content for the Edit panel will appear as one card of the deck; the content for the Keyword panel will appear as the other card. Previously, the content underneath the two <toolbarbutton> tags was for the Edit button. Now, we'll make that content one card of a deck and make the other card like this:

```
<hbox flex="1">
  <description>Content to be added later.</description>
</hbox>
```

Figure 5.3 shows the structure of the XUL document before and after this change.

All we've done is add a <deck> tag and the new content at the right spots. This could have been done in Chapter 2, XUL Layout; however, it's not possible to swap between the cards of a plain <deck> without using a script. Now that we have JavaScript, we can do that swapping. Briefly sneaking ahead to Chapter 6, Events, we note that <toolbarbutton> tags support an

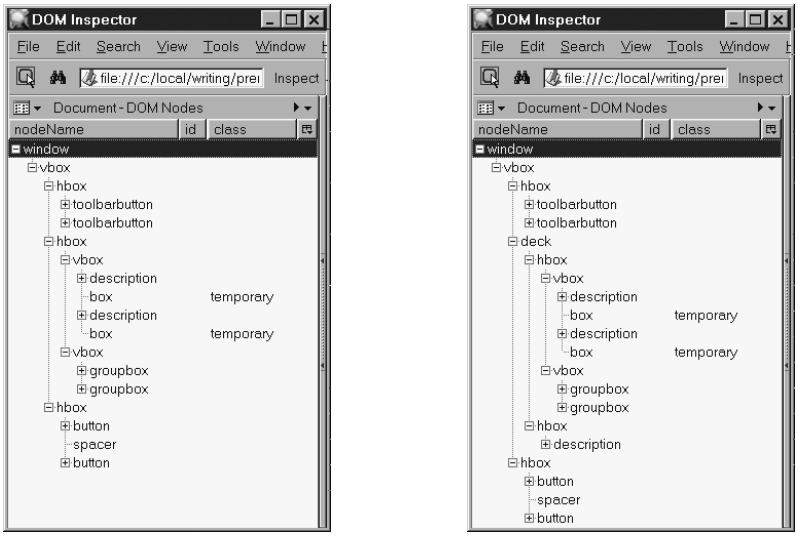


Fig. 5.3 Adding a <deck> to NoteTaker.

`onclick` event handler. We'll hook the script logic in there so that it's fun to play with.

There are numerous ways to make this work, but they all come down to adding some tag id attributes; writing a function, which we'll call `action()`; and adding an `onclick` handler. The `onclick` handlers look like this:

```
<toolbarbutton label="Edit" onclick="action('edit')"/>
<toolbarbutton label="Edit" onclick="action('keywords')"/>
```

The new ids look like this:

```
<deck flex="1" id="dialog.deck">
<hbox flex="1" id="dialog.edit">
<hbox flex="1" id="dialog.keywords">
```

The `action()` function looks like Listing 5.9.

Listing 5.9 Simple NoteTaker function that accepts commands as arguments.

```
function action(task)
{
  var card = document.getElementById("dialog." + task);

  if (!card || ( task != "edit" && task != "keywords") )
    throw("Unknown Edit Task");

  var deck  = card.parentNode;
  var index = 0;
```

```
  if ( task == "edit" ) index = 0;
  if ( task == "keywords") index = 1;

  deck.setAttribute("selectedIndex",index);
}
```

We want to get into the habit of passing command-like arguments to functions, since that's a very common practice for Mozilla applications. If we include this code in the `.XUL` file with a `<script>` tag, we'll have very obscure errors as described in the section entitled "Debug Corner" in this chapter, unless we're careful and use a `<![CDATA[]]>` section. It's better to put this script in a separate file right from the start and include that

```
<script src="editDialog.js"/>
```

The `action()` function calls the `window.document.getElementById()` method. This method is the most common starting point for script access to the DOM. It is passed the value of an XUL or HTML id attribute and returns an object for the tag with that id. You can then operate on that object via its properties and methods.

The remainder of the function navigates to the `<deck>` tag using the `parentNode` DOM property and sets the `selectedIndex` attribute there using the `setAttribute()` DOM property. Because this is a property meaningful to the `<deck>` tag, the XUL display system reacts to the change automatically, and the numbered card is displayed.

If command-like arguments are avoided, we can achieve the same effect with a quick-and-dirty solution. Equivalent `onclick` handlers that still use DOM interfaces look like these:

```
document.getElementById("dialog.deck").setAttribute("selectedIndex",0);
document.getElementById("dialog.deck").setAttribute("selectedIndex",1);
```

Whether structured or quick, just three interfaces of the DOM 1 Core standard are all we need to get most tasks done. These three are the `Document`, `Element`, and `Node` interfaces.

5.6.2 Alternative: Scripting Styles via the DOM

The use of `<deck>` is just one of many ways to script a XUL document. Another, equally valid alternative is just to modify CSS styles after the document has been loaded.

Starting from the `<deck>` example, remove the opening and closing `<deck>` tags. Now each set of content is enclosed in an `<hbox>` that can be treated like an HTML `<div>`. Listing 5.10 shows an alternate version of the `action()` method.

Listing 5.10 Simple NoteTaker function that accepts commands as arguments.

```
function action(task)
{
  var card = document.getElementById("dialog." + task);

  if (!card || ( task != "edit" && task != "keywords") )
    throw("Unknown Edit Task");

  var oldcard;  // the content to remove
  if (task == "edit")
    oldcard = document.getElementById("dialog.keywords");
  else
    oldcard = document.getElementById("dialog.edit");

  oldcard.setAttribute("style","visibility:collapse");
  card.removeAttribute("style");
}
```

This solution uses the same DOM interfaces as Listing 5.6 but modifies different attributes on the DOM hierarchy for the window. When the style rules for the content change, the rendering system inside the platform automatically updates the display. This use of the style attribute is a little clumsy. XUL provides a more useful collapsed attribute that can be set without affecting any other inline styles that scripts might add. The replacement lines follow:

```
oldcard.setAttribute("collapsed","true");
card.removeAttribute("collapsed");
```

The XUL `hidden` attribute shouldn't be dynamically updated because of the damage it does to XBL bindings (described in Chapter 15, XBL Bindings).

In addition to the `<deck>` and styled approaches, the DOM interfaces can be used to physically remove or add parts of the DOM hierarchy from a document. This approach is the third way to make content appear or disappear; however, it is an overly complex approach for simple tasks.

5.6.3 Alternative: Scripting `<deck>` via the AOM and XBL

When an XUL document is turned into a DOM hierarchy, more information is present than the W3C's standard DOM interfaces. There is also a set of interfaces resulting from Mozilla's XBL system. These extra interfaces add properties and methods to the DOM objects that are XUL-specific. Those properties and methods are extremely convenient and make the scripting job easier. Sometimes the pure DOM standards are a bit clumsy to use.

Let's see if the `<deck>` solution can be made easier using one of these interfaces. The tag we're most likely to script is the `<deck>` tag. We start by looking at the `xul.css` file. This file is stored in the `toolkit.jar` archive in the chrome. It's worthwhile keeping an unzipped copy of this JAR file somewhere handy. Viewing that file with a simple text editor, we look for "deck" and find:

```
deck {
  display: -moz-deck;
  -moz-binding: url("chrome://global/content/bindings/
     general.xml#deck");
}
```

The `-moz-binding` line tells us that `<deck>` does have a binding, so there are some goodies to look at. The binding is called deck, and it's in `general.xml`. So we view that file, and look for a line like

```
<binding id="deck">
```

Sure enough, it's there. Part of the binding reads

```
<binding id="deck">
  <implementation>
    <property name="selectedIndex" ...
```

That's all we need. Property names in XBL and attribute names in XUL generally match each other, and those names also generally match the names used for HTML objects. The `selectedIndex` property matches the `selectedIndex` attribute of the `<deck>` tag. We'll use that. In the `action()` method, replace this

```
deck.setAttribute("selectedIndex",index);
```

with this

```
deck.selectedIndex = index;
```

That's a trivial change, but it shortens the code enough that we can throw away the index variable and save a few lines. If the deck binding had a `setCard()` method, for example, then we could use that instead of writing `action()`. Perhaps it will arrive one day. The last few lines of the `action()` method can be collapsed to

```
var deck  = card.parentNode;

if ( task == "edit" )    deck.selectedIndex = 0;
if ( task == "keywords") deck.selectedIndex = 1;
```

5.6.4 Reading String Bundles via XPCOM

The second change we'll make to NoteTaker is to put external data into the displayed window from outside the XUL document. In short, we'll load content that appears inside the boxed areas. Over the course of this book, we'll change the way this information is loaded, saved, and displayed several times.

To get access to the outside, we don't need to use XPCOM components. We could submit a URL, which is explained in Chapter 7, Forms and Menus. Here we'll use a string bundle.

We can use string bundles, a.k.a. properties files from XUL or from JavaScript. If we do it the XUL way, it's the same as scripting `<deck>`—we add

some tags, and look for an XBL definition that supplies useful interfaces. Here, we'll work with a string bundle in raw JavaScript. Because it uses XPCOM, we must store our files inside the chrome.

We need an object that acts on properties files. In other words, we need a useful XPCOM interface and a component that implements that interface. If we choose to look through the XPIDL files (or this book's index), then it's easy to spot the interfaces in the `nsIStringBundle.idl` file. There are two interfaces, `nsIStringBundleService` and `nsIStringBundle`. Because "Service" is in the name of the first interface, it must be an XPCOM Service; that's a starting point for us. Recall that services produce an object using `getService()`; nonservices produce an object using `createObject()`.

We also note that the `createBundle()` method of that interface looks like this:

```
nsIStringBundle createBundle(in string aURLSpec);
```

OK, so this method creates an object with `nsIStringBundle` interface from a URL. That's easy enough to understand. The `nsIStringBundle` interface has a `getStringFromName()` method, which will extract a string from the string bundle (properties) file. We're not too concerned that the XPIDL files use their own `wstring` and `string` types; we know that XPConnect will convert those types to something that JavaScript can discern—a string primitive value that will appear as a `String` object.

This interface file also states at the top the associated XPCOM Contract ID, so we have found our XPCOM pair:

```
@mozilla.org/intl/stringbundle;1 nsIStringBundleService
```

We don't need a Contract ID for the `nsIStringBundle` interface because the service will create objects with that interface for us when we call `createBundle()`. We do need to get the service object in the first place. It is easy to ask for it using our discovered XPCOM pair:

```
var Cc = Components.classes;
var Ci = Components.interfaces;
var cls = Cc["@mozilla.org/intl/stringbundle;1"];
var svc = cls.getService(Ci.nsIStringBundleService);
```

If no errors appear in the JavaScript Console (and none should), then `svc` now holds the service object we've grabbed. We're ready to code. Listing 5.11 shows the results.

Listing 5.11 NoteTaker code to read in strings from a string bundle.

```
var Cc = Components.classes;
var Ci = Components.interfaces;
var cls = Cc["@mozilla.org/intl/stringbundle;1"];
var svc = cls.getService(Ci.nsIStringBundleService);
var URL = "chrome://notetaker/locale/dialog.properties";
var sbundle = svc.createBundle(URL);
```

```
function load_data()
{
  var names = ["summary", "details", "chop-query", "home-page", "width",
          "height", "top", "left"];
  var box, desc, value;

  for (var i = names.length; i>0; i--)
  {
    value = sbundle.getStringFromName("dialog."+ names[i]);
    desc  = document.createElement("description");
    desc.setAttribute("value",value);
    box   = document.getElementById("dialog." + names[i]);
    box.appendChild(desc);
  }
}
```

The `sbundle` variable contains a URL-specific XPCOM object. The URL we've chosen must follow the rules for property files. The function `load_data()` reads properties from that file and manipulates the DOM so that a `<description value="string">` tag is added to each placeholder `<box>` tag as content. Note how the object for `<description>` is built up and then added to the box tag at the end. That is more efficient than adding things piece by piece to the existing DOM.

This code also relies on the XUL document having some ids in place. We must add these ids by hand to the XUL. To do that, change every example of

```
<box class="temporary"/>
```

to something like

```
<box class="temporary" id="dialog.summary"/>
```

Finally, we'll run this `load_data()` function from another event target stolen from Chapter 6, Events: the `<window>` tag's `onload` attribute. Doing it this way ensures that the whole document exists before we start operating on it:

```
<window xmlns= "http://www.mozilla.org/keymaster/gatekeeper/
     there.is.only.xul" onload="load_data()>
```

Because the properties file is stored in the locale part of the chrome, the locale needs to be set up as described in the "Hands On" section of Chapter 3, Static Content. In other words, a `contents.rdf` file must be in place, and the `installed-chrome.txt` file must be up to date. The properties file itself might look like Listing 5.12.

Listing 5.12 dialog.properties property file for NoteTaker.

```
dialog.summary=My Summary
dialog.details=My Details
dialog.chop-query=true
dialog.home-page=false
dialog.width=100
```

```
dialog.height=90
dialog.top=80
dialog.left=70
```

The path of this file relative to the top of the chrome should be

```
notetaker/locale/en-US/dialog.properties
```

After all that work, the NoteTaker dialog box now looks like Figure 5.4.

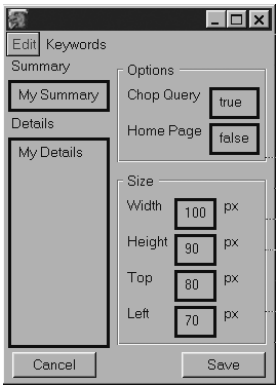


Fig. 5.4 NoteTaker dialog box with scripted property strings.

We've now successfully used XPCOM to interact with resources outside the loaded document. That's a big step, even if all we did was read a file. We'll leave writing this information out to a file to another day. Writing files (and reading them) is a long story. That story is told in Chapter 16, XPCOM Objects.

To summarize this activity, we've worked with DOM, AOM, XBL, and XPCOM objects. JavaScript has full access to the loaded content and, when installed in the chrome, can do anything it wants to do to that content. Most of the functionality we've added in this chapter has just been fancy little tricks. In future chapters, we'll replace those procedures with more professional work.

5.7 DEBUG CORNER: SCRIPT DIAGNOSIS

JavaScript is not a fully compiled language; consequently, it leaves the programmer with quite a bit of work in the area of detecting bugs. Because variables, properties, and methods are all resolved at run time, many bugs can lie hidden until very late in the development cycle. It's essential that the test-and-debug process have some structure and not be reduced to guesswork. This section describes the tools that can be applied to that process.

Good coding practices are probably the best defense against bugs. Keep your JavaScript code separate from XML. Always use terminating semicolons, meaningful variable names, and indentations. Always check arguments that are supplied to your functions, and return meaningful values at all times. Always use try blocks if the Mozilla interfaces you are using are capable of exceptions. Heed the advice on preferences in Chapter 1, Fundamental Concepts.

The `dump()` method of the window object is a very useful tool. When it is enabled, you can start Mozilla with the `-console` option and have diagnostic text spew out to a window without affecting the platform's own windows. That output can also be captured to a log file. On UNIX, `-console` can be used only from the command line. If your scripts include timed events, either generated by you or received by you, logged output is sometimes the only way to know what the order of events was in the processing.

Mozilla also supports a `javascript:` URL. This URL is useful for ad hoc tests of a document's state. If the document is loaded into a Navigator window (XUL documents can be loaded into Navigator as for any Web document), then `javascript:` URLs can be used to test the document's content. This is most useful when the document receives extensive input from the user. That input is usually stored as state information, which can be probed with statements like

```
javascript:var x=window.state.formProgress; alert(x);
```

`alert()`, of course, is a small dialog box that displays one or more lines of text. It can be placed anywhere in a script that is attached to a Mozilla window and can provide simple feedback on the state of the scripting environment and its content. `alert()` also pauses the JavaScript interpreter, which is a good thing for basic processing but a bad thing where information is supposed to be consumed in a time-critical way by the script (like streaming media). `alert()` is a trivial debugger.

Watchpoints are another trivial but extremely useful debugging tool. Every JavaScript object created has a `watch()` method. This method is used to attach a hidden function to a property of the object. This hidden function acts a little like Mozilla's "`set function ()`" syntax. Listing 5.13 illustrates.

Listing 5.13 Logging of object property changes using watchpoints.

```
function report(prop,oldval,newval)
{
   dump(prop + "old: " + oldval + "; new: " + newval);
   return newval;  // ensures watched code still works
};

var obj = { test:"before" };

obj.watch("test",report);
obj.test = "after";         // report() called
obj.unwatch("test");
```

The function `report()` has arguments imposed on it by `watch`. When the test property is watched, every change results in the `report()` function

being called as a side effect. When the property is unwatched, the side effect ceases. This is also a tactic for logging changes as they occur.

Finally, the `debugger` keyword can be inserted in scripts anywhere. It starts the JavaScript Debugger, whose commands can be learned about by typing **/help** in the command line at the bottom of its window. This debugger is based on the XPCOM pair:

```
@mozilla.org/js/jsd/debugger-service;1 jsdIDebuggerService
```

If you don't like the JavaScript Debugger, you can implement something on top of this component and its `jsdIDebuggerService` interface yourself.

When obscure problems occur, shut down Mozilla completely, restart it, and retest. Always show the JavaScript Console (under Tools | Web Development), and always clear the console log before loading test documents so that new errors are obvious. On Microsoft Windows, occasionally check if zombie Mozilla processes are hanging around without any windows to identify them.

5.8 SUMMARY

The JavaScript language is small and easy to master. It is much like many other languages in the C family of languages, except for its unique treatment of objects. The scope and prototype chain features of the language are curious and subtle, and many hours can be spent crafting clever uses. JavaScript 2.0 will de-emphasize prototype chains in favor of classes.

Compared with the core language, the interfaces that JavaScript uses are very extensive. They range from simple and convenient to remarkably obscure and encompass Java, plugins, and legacy browser support.

Well-trained Web developers will find Mozilla's support for DOM interfaces easy to use, familiar, and powerful. It is immediately applicable to XUL-based applications. The extensive standards support is something of a relief after years of cross-browser compatibility checks. Beginner XML programmers are advised to study the XML half of DOM 1 closely until the concepts of Node, Element, Document, and factory method sink in.

XPConnect and XPCOM infrastructure is the brave new world of Mozilla. With over a thousand components (and a thousand interfaces), there's only one way to eat this elephant: one bite at a time. You will probably never use some interfaces; your use of other interfaces will depend on how deeply you attempt to customize the platform and how aggressive your application is.

XPCOM provides a world of objects with the potential to make JavaScript programming as complex as the programming worlds of Java, C, or C++. XPCOM interfaces each have an XPIDL definition that is easy to read after it is located. Browsing through these interfaces can give you an early hint of Mozilla's full capabilities.

Rather than delve into XPCOM and never return, we next go back to the world of user interfaces and consider user input.