EMBEDDED SERVER PAGES REFERENCE

Last Updated: Wednesday, July 23, 2008

CONTENTS

How Embedded Server Pages Works

Embedded Server Pages are HTML pages with embedded JavaScript logic. When a client requests an ESP page, the web page is compiled into pure JavaScript that is then interpreted at run-time. Although slower than pure C code, the ESP JavaScript calls C routines for many functions so the overall result is fast page generation and response to client requests.

Example ESP Web Page

So what does an ESP HTML web page look like? ESP pages are standard HTML pages with embedded Javascript code containing scripting logic. You can use all your normal HTML tags. ESP tags delimit the JavaScript using the delimiters <% and %> to bracket the scripting code.

```
<body>

<h1>Normal HTML header</h1>

<% Script Code Here %>

</body>
```

This script code is executed and replaced by the ESP interpreter with the output from the script code. This all occurs at the server before any of the HTML is sent to the client. The client never sees the script logic and this greatly enhances the security of your application or device. Don't confuse ESP with client side JavaScript. A single page may have both server side and client side JavaScript. ESP JavaScript is placed between <% and %> tags and is replaced by the web server before it is sent to the client. Once the page reaches the clients browser, it will execute any client side JavaScript which is typically between <SCRIPT> and </SCRIPT> tags.

The following example demonstrates just a few of the constructs available using ESP.

```
<HTML>
<HEAD><TITLE>Simple ESP Test Page</TITLE></HEAD>

<BODY>

<P> The HTML query string for this page is @@QUERY_STRING</P>

<!-- This quickly generates a very large table -->

<TABLE>

<% for (i = 9; i < 3; i++) { %>

    <TR><TD>Line</TD><TD>@@i</TD>

<% } %>

</TABLE>
```

```
<!-- Sample ESP Procedure Calls -->

<% displayCurrentSecurityStats("myDb", 1); %>

<%
    // Inside JavaScript blocks we can put any server-side
    // JavaScript we like

    i = 2;

    write("i is " + i);
%>

</BODY>

</HTML>
```

As you can see, the page is standard HTML with JavaScript logic inside ESP tags. What makes this especially interesting, is that you can easily create new Embedded JavaScript procedures in your application to suite your needs. These can then be called from the ESP web pages. This close nexus between the HTML page and your application logic is what makes AppWeb such an easy platform to use to create dynamic web pages.

ESP pages typically have a ".esp" extension although they can in some web servers such as AppWeb be configured to match by URL prefix. They are processed by the ESP web server handler.

ACCESSING VARIABLES

There are three methods to access JavaScript variables within ESP scripts. You can use the ESP write procedure to output the value of a variable back to the client. For example, assuming you have the current temperature in a JavaScript variable called temp.

```
<P>Today's temperature is <% write(temp); %></P>
```

As this kind of variable access is a very common occurrence, a shorter form may be more convenient. Because the JavaScript assignment operators sets the result, it can be used to return a value to the client. For example:

```
<P>Today's temperature is <% =temp; %></P>
```

Even easier is to use the @@ directive which does not require any <% %> enclosing tags. You use this by prepending the required variable with @@. For example:

```
<P>Today's temperature is @@temp</P>
```

SCRIPTED HTML

Because the ESP web page is compiled into JavaScript, you can script normal HTML tags. If a section of HTML is enclosed by a JavaScript for loop, you can output the HTML block each time round the loop. This is an easy way to generate tables. In the example below, three rows of a table are output.

```
<%  for (i = 0; i < 3; i++) { %>
    <TR><TD>Line</TD><TD>@@i</TD></TR>
<% } %>
```

SERVER SIDE INCLUDES (SSI)

The AppWeb ESP implementation also supports server side includes. To include another document use the directive:

```
<% include fileName.esp %>
```

This will replace the ESP script with the content of the nominated filename The included file is assumed to be an ESP page that will be parsed at the point of the include directive. It may contain further ESP tags and include directives.

You can also include pure Embedded JavaScript code using the JavaScript include procedure:

```
<% include("myLib.js"); %>
```

In this case, the included file must contain pure JavaScript and must not contain ESP directives.

EMBEDDED JAVASCRIPT

JavaScript is an interpreted, object oriented language suitable for embedding in applications and devices. Best known as the scripting language for browsers, JavaScript is general purpose language that can also be used for web server scripting and many other embedded scripting needs. Embedded JavaScript is used by Embedded Server Pages to allow access to HTTP variables and to retrieve dynamic data for presentation to the client's browser.

JavaScript was originally developed by Netscape but has subsequently been standardized as ECMAScript. The ECMA-262 standard describes JavaScript 1.5 which is the basis for Embedded JavaScript.

JavaScript is syntactically similar to C or C++. JavaScript expresssions and statements such as if/else, for, +, -, &&, || are almost identical to C. However, JavaScript is an untyped language that will automatically convert data from one type to another as expressions and statements require. This makes it much more like Perl than a strongly typed language. JavaScript also has objects and methods and resembles Java and C++ in this regard and it supports associative arrays like Pearl or TCL. Thus, JavaScript is a nice blend of some of the best features of C, C++, Java and Perl.

JavaScript is best when it is used for scripting. It was not designed for large-scale application development. For that purpose, strongly typed languages are a better choice.

This JavaScript documentation includes sections on:

- Language
- Objects and Arrays
- Expressions and Operators
- JavaScript Statements
- Standard Globals
- Extending JavaScript

For more information about JavaScript, we recommend the excellent book on JavaScript by David Flanagan, JavaScript the Definitive Guide by O'Reilly, ISBN: 0-596-00048-0, 2002.

JAVASCRIPT QUICK EXAMPLE

So what does JavaScript actually look like? The following code fragment demonstrates the syntax and some of the features of JavaScript:

```
/*
 *  Can use either C or C++ style comments
 */
function justForDemo(start, end)    // Arguments do not have types
{
    var i, j;                       // Declare variables with var
    var objects = new Array(10);    // Create an array with ten elements

    for (i = start; i < 10; i++) {  // Classic for loop
        objects[i] = new Object();  // Store a new object
    }

    for (o in objects) {            // Iterate over all the objects
        // Can do something here to each object
    }
    return "Hello World";
}

justForDemo(0, 10);                 // Invoke the function
```

EMBEDDED JAVASCRIPT OVERVIEW

Embedded JavaScript (EJS) was created as a subset of ECMAScript as specified in the ECMA-262 specification. The ECMA specification defines a complete and extensive language of which Embedded Javascript implements a well defined subset focussed on the needs of the embedded market.

The complete ECMA-262 specification has many features that are not essential for embedded use and the resultant memory footprint for the complete spec (typically > 500K) is too large for many embedded applications. Embedded JavaScript in contrast, requires approximately ~50K of code and 20K of RAM to execute depending on the size of the program.

Design Goals

The design goals for EJS were, to develop a useful and powerful subset of the JavaScript language, while minimizing the system resources required in which to execute. Redundant language features were eliminated and unnecessary features were ommitted. An architecture and implementation was then chosen for EJS to prioritize small memory size above fast execution speed. For example: a recursive descent parser was selected over a table driven parser as it requires considerably less memory.

Recursive Descent Parser

Embedded JavaScript uses a compact, recursive descent parser. This is a one-pass, parse and execute interpreter. Compared to two-pass interpreters that create in-memory representations of the script to execute, the one-pass, recursive descent parser has greatly reduced memory requirements. It is somewhat slower than two-pass interpreters that use Just-in-time compilation technologies.

Multiple Instances

EJS supports multiple independent instances at run-time. Embedded Server Pages uses this feature to allow multiple HTTP requests to execute concurrently.

Garbage Collection

Unlike C and C++ where objects are explicitly destroyed, JavaScript uses an automatic mechanism called Garbage Collectionto reclaim unused memory when objects are no longer used.

Because Embedded JavaScript may be used in embedded real-time systems, determinism in the garbage collection was an important design goal. However, the traditional JavaScript mark and sweep algorithms exhibit decidedly non-deterministic behavior. Consequently, a modified reference counting approach was chosen. This approach is very efficient and uses little CPU or memory to implement, but it can be temporarily miss reclaiming objects that have cyclical references. This is normally not a problem when implemented in systems like Embedded Server Pages that use smaller scripts that run for short durations.

Data Types

Embedded JavaScript borrows from some of the proposed JavaScript 2.0 features to provide a rich suite of types. 32-bit and 64-bit integers are provided along with floating point, boolean and string primitive types. EJS can also be configured when building from source code to select the default numeric type.

Quick Language Overview

Embedded JavaScript implements the following standard JavaScript elements:

- Case sensitivity, white space, optional semicolons, C and C++ style comments
- Identifiers
- Data types including numbers, booleans, strings with quoted backslash characters
- Expressions / operators (< <= == != > >= + - / % << >> ++ -- && || !)
- If/else statement
- for and for/in
- Functions and return statement
- Objects and new statement
- Arrays and associative arrays

The following language elements are not implemented:

- Exceptions including try, catch and throw
- Labeled statements including: break, continue
- Control flow statements including: switch, while, do, export, import, with
- Regular expressions
- The operators === !== <<< >>>
- The prefix ++ and -- operators
- Function, array and object literals
- Standard object methods: toLocaleString, hasOwnProperty, propertyIsEnumerable, isPrototypeOf
- Standard array methods: length, join, sort, reverse, slice, splice, push, pop, unshift, shift, toLocaleString
- Number and Date classes
- Object class prototypes and class methods

Other differences in Embedded JavaScript from the ECMAScript standard are listed below. All these differences were deliberate design decisions made in support of the design goals.

- Strings are stored in ASCII not UNICODE.
- Number types are integers by default instead of floating point.

LANGUAGE SYNTAX

Embedded JavaScript is case sensitive and uses ASCII as its character set. White space between tokens is ignored with the exception that statements that omit trailing semicolons must be on a line by themselves.

Comments can use either the C commenting style with text between /* */ delimiters, or the C++ style may be used where text from // to the end of the line is regarded as a comment.

Identifiers are names used for variables and functions. Identifiers must begin with either a letter, and underscore '_' or a dollar sign. Identifiers must not use any reserved word from the list below:

RESERVED WORDS

Embedded JavaScript regards all these words as reserved.

- delete
- else
- false
- for
- function
- if
- in
- new
- null
- return
- this
- true

- var

Embedded JavaScript also regards these words as reserved. They are not yet implemented, but users should avoid them for future compatibility.

- break
- case
- catch
- continue
- default
- do
- finally
- instanceof
- switch
- throw
- try
- typeof
- void
- while
- with

It is also recommended that you do not use any of the following words. ECMA may implement them in future standards.

- abstract
- boolean
- byte
- char
- class
- const
- debugger
- double
- enum
- export
- extends
- final
- float
- goto
- implements
- import
- int
- interface
- long
- native
- package
- private
- protected
- public
- short
- static
- super
- synchronized
- throws

- transient
- volatile

PRIMITIVE DATA TYPES

Variables are declared either by the var keyword or by simply assigning a value to a new variable name. In JavaScript is is not essential to declare variables before using them. However, it is considered good practice to do so.

JavaScript does not have explicit type declarations. They are not given an explicit type when declared. The underling type of a variable is determined by the type of the data stored in the variable. For example:

```
var x;

x = "Hello World";
```

This will cause the variable x to store the string "Hello World". Embedded JavaScript will store this as the primitive String data type. If this variable is assigned an integer, the underlying type will be changed. For example:

```
x = 55;
```

The x variable will not hold an integer primitive type with a value of 55. If x is used in expressions, its value will be copied and converted to other types as required.

Embedded JavaScript supports the following primitive data types. These types can be combined in objects to create arbitrary structured data types.

- Boolean
- Floating Point
- Integer
- Strings

JavaScript will automatically convert variables from one type to another as required by the current expression.

NUMERIC TYPE

The ECMA standard specifies that JavaScript numerics are stored and manipulated as floating point entities. For embedded uses, this is often less than ideal. Consequently, EJS selects the default numeric type to be 32-bit integer. Note that floating point can still be used and expressions can still be explicitly converted to floating point by multiplying by 1.0.

BOOLEAN TYPE

A boolean value can be either true or false. Assigning either true or false to a variable or the result of a conditional expression will set the type of a variable to boolean. The following example will set x to true.

```
var y = 1;
```

```
var x = (y == 1);
```

EJS defines the global variables true and false with the appropriate boolean values.

INTEGER TYPE

Integer literals are 32-bit integers. Integer literals may be specified as octal values by prefixing with a '0' or hexadecimal by prefixing with 0x or 0X.

FLOATING POINT TYPE

Floating point numbers are stored as C doubles. Floating point literals may be specified simply by using a decimal point in the literal. For example:

```
var pi = 3.14;
```

Floating point literals can also use an exponential notation of the form: floating number followed by 'e' or 'E', followed by an optional sign '+' or '-', followed by an integer exponent. In other words:

```
[digits] [.digits] [E|e] [(+|-)digits]
```

For example:

```
var pi = 3.14e1;

var pi = 314.0e-2;
```

Embedded JavaScript also defines two useful floating point constants. The global variable NaN is set to the "not a number" value. This value is the result when a math operation gives an undefined value or error. For example, divide by zero will result in the NaN value.

The other constant is Infinity. If a floating point operation overflows the largest possible floating number, the result is equal to the Infinity value.

STRING TYPE

JavaScript strings are immutable sequences of ASCII letters or escaped literals. They are primitive types and are not treated as objects. This means that they are always passed by value and not by reference. String literals may be enclosed in either single or double quotes.

JavaScript strings are easy to work with due to the inbuilt ability to append strings via the "+" operator. For example:

```
var str = "Happy Birthday" + name;
```

EJS supports the following escaped letters inside string literals:

Escape Sequence	Description
\b	Backspace (0x8)
\f	Form feed (0xc)
\n	New line (0xa)
\r	Carriage return (0xd)
\t	Horizontal tab (0x9)
\v	Vertical tab (0xb)
\uNNNN	Unicode character (0xNNNN)
\xNN	Hexadecimal character (0xNN)
\NNN	Octal character (ONNN)
\’	Literal single quote
\”	Literal double quote
\\	Literal backslash

FUNCTIONS

A JavaScript function is a JavaScript that can be executed by name with supplied arguments. Functions are declared via the function keyword followed by optional arguments and a function body contained within braces. For example:

```
function myPoint(x, y)
{
    return x + y;
}
```

Functions can use a return statement to return an arbitrary value value. This may be a primitive type or it may be a reference to an object.

To invoke a function, the function name is used with the () operator following. The actual arguments are supplied within the "()" . The arguments to the function are evaluated left to right. For example:

```
var x = myPoint(1, y + 7);
```

Functions may also be assigned to properties of objects to create methods.

SCOPE AND THE FUNCTION CALL OBJECT

When functions are invoked, EJS creates a new set of local variables for use by the function. EJS also creates several objects to describe the function call.

The function call arguments are passed in as named parameters and they are also stored in the arguments[] array. The arguments array allows functions to be written to take a variable number of arguments. For example:

```
function max()
{
    biggest = 0;
```

```
    for (i = 0; i < arguments.length; i++) {
        if (arguments[i] > biggest) {
            biggest = arguments[i];
        }
    }
}
```

EJS also defines a callee object property in the arguments object. The callee property has a length property member that stores the expected number of arguments by the function. For example:

```
function myFunc(x, y)
{
    if (arguments.callee.length != 2) {
        // Error
    }
}
```

NOTE: JavaScript and EJS do not validate the number of arguments supplied to a function at run-time.

Embedded JavaScript does not implement a few capabilities from the ECMA specification relating to functions. These include nested functions, the prototype property and the apply and call methods.

VARIABLES

VARIABLE SCOPE

JavaScript defines two scopes for variables: global and local. Global variables are shared among functions that execute in a single JavaScript program. Local variables are created for each function and are destroyed on function exit. Objects are created on an object heap and references to the objects may be stored in either the global or local variable stores.

As described earlier, JavaScript variables may be optionally declared via the var statement. When used inside a function, the variable is created in the local variable store. If used outside any function, the variable is created in the global store.

UNDEFINED AND NULL

If a variable is declared but not assigned, its value is said to be undefined. For example:

```
var x;
```

This declares a variable x, but does not assign it a value. All such unassigned variables are set to the undefined value. Variables can be tested for equality with undefined. For example:

```
if (x == undefined)
{
```

```
    // x does not yet have a value
}
```

Do not confuse undefined with null. The null value implies that a variables does not point to a valid object. You can assign null to a variable to disconnect the variable from pointing to an object.

Type Conversions

JavaScript will usually handle type conversions for you automatically. Adding a numeric to a string will cause the number to be converted to a string and appended to the left hand side string. Adding an integer to a floating point will cause the integer to be converted to a floating point prior to the arithmetic.

If you want to explicitly force a type conversion, use the following:

Initial Type	Target Type	Action	Example
Integer	String	Add a string on the right	num + “”;
String	Integer	Add a numeric on the right	1 + string;
Integer	Floating Point	Multiply by 1.0	num * 1.0;
Object	String	Invoke the toString method	o.toString()

Type Conversion Rules

Embedded JavaScript observes the following type conversion rules:

If the left operand is a string, the right hand operand will be converted to a string. If both operands are numeric, a numeric operation is done.

If either operand is an object, it is converted to a string.

If both operands are boolean and the operation is ==, !=, then proceed without a type conversion.

If one operand is a string and the other is a numeric, try to convert the string to a number. Otherwise, try to convert the numeric to a string.

If both are numeric, but not of equal types. Give priority to floating point, then integer, then boolean.

Objects and Arrays

JavaScript has powerful, object oriented capabilities that allow it to closely model many data types and structures. It provides strong support for objects, arrays and associative arrays. Objects may be dynamically created and automatically garbage collected and unlike strongly typed languages such as C++ and Java, object definitions may also be easliy created at run-time.

Embedded JavaScript does not support object literals, object prototypes, class methods or class properties. This was done to minimize the memory footprint. The supported language constructs provide easy alternatives to these more complex features.

OBJECTS OVERVIEW

JavaScript like other object oriented languages, supports the notion of objects that are collections of named variables. These variables are usually called object properties and are referenced via a dot notation. For example:

```
var o = new Object();

o.color = "red";
```

Properties may be any JavaScript primitive type or they may be objects themselves. They may also be functions in which case they are referred to as object methods. Properties may also be referenced using an array index notation. For example:

```
o["color"] = "blue";
```

This feature allows objects to function as associative arrays.

CREATING OBJECTS

Objects are created via the new operator which returns a reference to the object. Objects are not created in the local or global variable store. Rather, they are created in the JavaScript memory heap. When the result of the new operator is assigned to a variable, what is assigned is a reference to the object. There may exist multiple references to an object at any time. When there are no remaining references, the objects is not accessible and is deleted by the EJS garbage collector. For example:

```
var o = new Object();

var o2 = o;
```

After the assignment to o2, there are two references to the object. You an explicitly remove a reference to an object by assigning another value or null to a variable. Continuing the example above, if we assign null to the variable "o", then the object created will have only one reference. Assigning null to o2 will remove the final reference and the object will be destroyed.

When objects are passed to functions, they are passed by reference. The object is not copied.

OBJECT CONSTRUCTORS

When the new operator is used, what is supplied to the right of the operator is an object constructor. The Object () function is actually the constructor for the object class. EJS includes builtin constructors for the Object and Array classes, and you may create your own constructors for your custom objects.

An object constructor is a global function with the name of the desired object class that returns the required object instance. By convention, object constructors start with an upper case letter. For example:

```
function MyObj()
{
    var o = new Object();      // Create a new bare object
    o.height = 0;           // Create and initialize a height property
    o.width = 0;             // Create and initialize a width property
    return o;                 // Return the object
}

var firstObj = new MyObj();     // Create a first instance
var secondObj = new MyObj(); // Create a second instance
```

This creates two instances firstObj and secondObj using the MyObj constructor. Constructors create the required properties by simply initializing the properties.

OBJECT METHODS

Object methods are created similarly to object properties by assigning a function to a named object property. For example:

```
//
//  Method for MyObj
//
function print() {
    println("height = " + this.height);
    println("width = " + this.width);
}

//
//  Constructor
//
function MyObj () {
    var o = new Object();        // Create a new bare object
    o.height = 0;           // Create and initialize a height property
    o.width = 0;             // Create and initialize a width property
    o.print = print;
    return o;                    // Return the object
}

var o = MyObj();                // Create the object
o.print();                 // Invoke the print method
```

The modified MyObj constructor now assigns the print method and creates a print property in the newly minted object.

SUBCLASSING

To subclass an object class, you create a new object constructor that invokes the subclass constructor. You then add your properties to the object returned by the base constructor. For example:

```
function YourObj()
{
    var o = new MyObj();
    o.color = "black";
    return o;
}
```

JavaScript does not express classes as C++ or Java does. Rather objects are created by constructors that determine the properties of the object. ECMAScript does have a way to replicate objects more quickly by creating an object prototype. This consists of creating a template object prototype that is used when the constructor is invoked. This paradigm is often unfamiliar to developers and so EJS has chosen not to implement it. The proposed JavaScript 2.0 release may include a class construct which is under consideration for inclusion in a future version of EJS.

STANDARD METHODS

JavaScript objects define the toString method which converts the object representation into a string. The default toString method returns "[object this]". You may override the toString method by assigning your own function to the toString property in your object constructor.

ARRAYS

JavaScript arrays build upon the object class and provide a powerful array base type that can be indexed numerically or via strings. In reality, JavaScript Arrays are a subclass of the base Object class with a little extra functionality.

When using string indicies, the array implements an associative array. When using numeric indicies, it behaves more like a classical array. In both cases, arrays will grow on demand as new elements are stored in the array and will be automatically garbage collected when the array is no longer referenced or the program / script exits.

Array elements are read and written using the [] operator. For example:

```
customers["bill"] = new Customer();          // Using a string index

visited[43] = 1;                             // Using a numeric index
```

ASSOCIATIVE ARRAYS

Associative arrays are very useful construct in a scripting language, especially when combined with the for / in statement. Associative arrays allow you to store arbitrary data indexed by a string key. The data stored can be a primitive type or it may be an object of your choosing. The EJS associative array index mechanism is an efficient hash lookup.

For example:

```
var store = new Array(100);

store["product 1"] = new Product
...
for (product in store) {
    print("Product is " + store[product].id);
}
```

The for / in statement iterates over all the elements of the array, setting the index variable "product" to each element in the store array.

NUMERICAL ARRAYS

Numerical arrays store arbitrary data indexed by numerical values with an origin at zero. Numerical arrays are created via the new operator and the Array() constructor. The Array constructor supports invocation with no parameters, a single size parameter and a list of elements to insert into the array. For example:

```
var empty = new Array();              // Create an empty array

var ten = new Array(10);              // Create an empty array with 10 slots

var products = new Array("milk", "eggs", "bread");
```

When an array is created the length property is set to the current length of the array. When an array is created with a single size parameter, it's length will be set to the requested size and all the elements in the array will be set to undefined.

EJS does not support array literals.

EXPRESSIONS AND OPERATORS

Expressions and operators in JavaScript follow closely their cousins in C/C++ or Java. Users familiar with these languages should feel right at home.

Expressions are combinations of variables and operators to produce a resulting value. Expressions can be used as the right hand side of an assignment, as function arguments, and in return statements. Expressions are the heart of JavaScript.

Embedded JavaScript implements a subset of the ECMAScript specification for expressions and operators. The following operators are not supported: ===, !==, >>>, prefix ++ and --, &, |, ^, ?:. The operator / assignment combinations such as *= are also not supported.

OPERATOR SUMMARY

The following table summarises the operators available in Embedded JavaScript. For a complete specification, please consult the ECMA-262 specification.

Operator	Operands	Description
/ * %	Numbers	Divide, multiply, modulo
+	Numbers, Strings	Add numbers, concatenate strings
!	Boolean	Unary not
++	Numbers	Post-increment
--	Numbers	Post-decrement
<<	Integers	Left shift
>>	Integers	Right shift
== !=	All	Compare if equal, not equal
< <=	Numbers, Strings	Compare if less than, less than or equal
> >=	Numbers, Strings	Compare if greater than, greater than or equal
&&	Booleans	Logical AND
||	Booleans	Logical OR
.	Object property	Property access
[]	Array[integer] Array[string]	Array element access
()	function(any, …)	Function call
new	constructor function	Create a new object using the given constructor
delete	delete	Undefine a property

Operator Precedence

When expressions are evaluated, the operands bind to the operators depending on the precedence of the operators. For example:

```
a = x + y * z;
```

This is equivalent to:

```
a = x + (y * z);
```

The following table lists the operators in order of highest to lowest precedence.

Operator
. [] () new
++ -- - + delete
* / %
+ -
<< >>
< <= > >=
== !=
&&
||
=

Operator Notes

Addition

The + operator will add numeric operands (integers and floats). If used with string operands it will catenate the strings. If the left hand operand is a string, the right hand operand will be converted to a string. If the right hand operand is an object, it will have its toString method invoked to convert its value to a string.

Subtraction, Multiplication, Division, Modulo

If used with non-numeric operands, they will be converted to numbers first. If one operand is floating point, the other will be converted to be floating also.

Equality, Inequality

The ECMA JavaScript standard treats the == and === operators differently. Embedded JavaScript only implements the == operator. The == operator will compare its operands for equality after performing any required type conversions according to the following procedure:

- If they are primitive types and are both of the same type, their values are compared.
- If they are not both of the same type, EJS will use type conversion first before comparing.
- If one is undefined and the other is null, they are regarded as being NOT equal.

Less Than, Greater Than

String comparisions are performed according to the ASCII collating sequence.

Delete

The delete operator removes the specified propery from the referenced object. For example:

```
delete customer.currentOrder;
```

This will delete the currentOrder property from the customer object.

JavaScript Statements

JavaScript statements are similar to their C/C++ counterparts but with some nice object oriented additions. Statements respresent the body of a JavaScript program by combining logic with expressions.

Embedded JavaScript implements a subset of the statements defined by the ECMAScript specification. The following statements are not supported: switch, while, do, break, continue, try/catch/finally/throw and with.

Statement Overview

Statement	Syntax	Brief Description

;	;	Empty statement, do nothing
for	for(init; condition; increment) statement	Standard loop
for (.. in ..)	for (variable in object)	Iterate over all properties in an object
function	function name([arg1 […, arg2]) { statements }	Define a function
if / else	if (expression) statement	Conditionally execute a statement
return	return [expression];	Return a value from a function
var	var identifier [= value] […, identifier [= value]];	Declare and initialize variables

FOR

The for statement provides the basic looping construct for EJS. Similar to the C for loop it implements an initialization, conditional and increment phases of the statement. The for statement also provides a for / in construct that is described below.

```
for (initialization; conditional; increment)
    statement;
```

For example:

```
for (var i = 0; i < 10; i++) {
    print("i is " + i);
}
```

You can put any expression or statement in the initialization or increment sections. In this example, we simply define a variable I and initialize it to zero in the initialization section and increment it each time after the statement body is executed. The conditional expression must evaluate to a boolean value which if true allows the statement body to be executed. The increment expression is evaluated after the statement body is executed and before the conditional is retested.

FOR .. IN

The for statement has a powerful variant that allows you to iterate over all the properties in an object. It uses the following syntax:

```
for (variable in object)
    statement;
```

This statement will execute the statement body for each property in the object. Each time, the variable will be set to the name of the next property in the object. The order of the walk through all the properties is undefined. NOTE: it will not be set to the property value, but will be set to the property name. For example:

```
for (var v in customer) {
    println("customer." + v + " = " + customer[v]);
}
```

This will print "customer.propertyName = value" for each property defined in the customer object.

FUNCTION

The function statement defines a new global function according to the syntax:

```
function name([arg1 [... , arg2]) {
    statements
}
```

The function name must be an identifier and not a reserved word and is followed by an optional list of arguments. Between braces, a set of statements define the function body. For example:

```
function min(arg1, arg2) {
    if (arg1 < arg2) {
        return arg1;
    } else {
        return arg2;
    }
}
```

Function declarations can also be nested, i.e. a function may be defined in the statement body within an outer function. In this manner, the inner function will only be visible within the scope of the outer function.

When the function is invoked a new local variable store is created so that any variables declared and used in the function will be private to the function. Functions invoke other functions and each function will have its own local variables. If a variable is assigned to without using the var statement, the variable will be created in the global variable store.

IF / ELSE

The if statement is the primary conditional execution ability with JavaScript. It takes the form:

```
if (expression)
    statement
[ else statement ]
```

The expression is evaulated and if true, the first statement is executed. If an else phrase is added and the expression evaluates to false, then the else statement will be executed.

Statements may be grouped using braces. For example:

```
if (i < j) {
    write("i is " + i);
    write("j is " + j);
} else {
    // Do something
}
```

The conditional expression may be a compound conditional expression. For example:

```
i = 0;
j = 1;
if (i < j || j != 0 || getToday() == "sunday") {
    // Do something
}
```

EJS uses lazy evalation where if the first expression (i < j) is true, then the following expressions will not be evaulated. In the example above, getToday() will not be called as i is less than j.

SWITCH

You can chain if / else statements together to implement a switch capability. For example:

```
if (today == "monday") {
    // Do something
} else if (today == "tuesday") {
   ...
} else {
    // Default
}
```

RETURN

The return statement is used to supply a return value inside a function. The return statement may appear without an expression to cause the function's execution to terminate and return.

A return expression may be a simple value or it may be an object reference. For example:

```
function myFunc(x)
{
    if (x == 0) {
        return null;
    }
    return new MyObj(x);
}
```

VAR

The var statement declares variables and initializes their values. Although not strictly required by the language, as variables will be declared automatically by simply assigning to them, it is good practice to always use var declarations for all variables. The var statement takes the form:

```
var identifier [ = value ] [... , identifier [ = value]];
```

For example:

```
var x;
var y = 4;
var a, b = 2, c = "sunny day";
```

If an initializer value is not defined, the identifier will be set to the undefined value.

If the var statement is used within a function, the variable is defined in the local variable store. If it is used outside a function, the variable is defined on the global store. If a variable is assigned to without a var statement, then the variable is created on the global store. For example:

```
x = 2;

function myFunc()
{
    x = 7;
}

write("x is " + x);
```

This code snippit will print "x is 7".