

# D Introduction into JScript

Within this appendix you get a short introduction into the JScript language. If you have used JavaScript or Java already, programming in JScript will be no major problem. Have a look how a JScript program is structured and how you may use constants and variables. Read which program constructs are offered in JScript. Check out how functions are handled in JScript.

## Basics

Below I like to discuss a few basics, which are relevant for JScript programs.

### What's JScript?

JScript is Microsoft's implementation of ECMA 262 Script. ECMA is the abbreviation for European Computer Manufacturer. This group creates a vendor independent standard for a scripting languages, which has its roots in JavaScript developed from Netscape in the Netscape Navigator. Therefore JavaScript is very close to the ECMA 262 specification. This mean, you may use your JavaScript knowledge to write JScript programs. You must omit only a few Netscape specific statements and some objects and methods, which are not supported within the WSH. JScript represents a complete implementation of the ECMA 262 standard, which is extended with a few features to support the Microsoft Internet Explorer. But you must not take care about these extensions within WSH scripts.

**NOTE:** JScript is also (like VBScript) an interpreted language. Therefore you need only the source code stored in a *.js* file to execute the script within the WSH. JScript is (similar to C++ and Java) object orientated.

### The structure of a JScript-program

If you have used JScript or JavaScript within HTML documents, I like to mention the first difference: JScript program for the WSH doesn't contain any HTML-tags. The whole script is stored within a *.js* file. The listing shown below is a typical JScript program:

```
//*****
// File:   ErrorTest1.js   (JScript)
// Author: (c) G. Born
// Windows Scripting Host Sample Script
// Contains a statement to activate the debugger
//*****

var mbOKCancel = 1;           // declare variable
var mbInformation = 64;
var mbCancel = 2;
```

```
var Text    = "Test sample";
var Title   = "Born's Windows Scripting Host sample";

var WSHShell = WScript.CreateObject("WScript.Shell");
var intDoIt = WSHShell.Popup(Text,
                             0,
                             Title,
                             mbOKCancel + mbInformation);

if (intDoIt == vbCancel)
{
    WScript.Quit();
}

WScript.Echo("Sample executed");
WScript.Quit();           // terminate script
// End
```

**Listing D.1.**  
*A JScript program*

Compared to the VBScript program discussed in the previous chapter, JScript comes with a few differences, which I like to explain below. Compared to JScript programs included in a HTML document, a WSH script needs no event handling.

## Comments

Would you like, that the interpreter doesn't interpret a line or part of the line within the code? Then you must mark this statement as a comment. Comments begin in JScript with two slashes.

```
// This is a comment
WScript.Quit(); // we are ready...
```

The comment may use the whole line, or it may be appended to an executable statement (as you can see in the 2nd line above). If a comment is found, the rest of the line is ignored.

**NOTE:** JScript supports also comments in the format `/* .... */` to spread multiple lines. But I haven't used this format within this book.

## Remarks about JScript statements

JScript statements must be entered according to the JScript syntax rules. Compared to VBScript (and other languages) there are a few remarkable differences.

**Important:** Because JScript has a similar syntax as C or Java, you must take care how a statement is written. Keywords, function names or variable names must be spelled case sensitive. Therefore `res = Born()` is different from `res = born()`. Especially beginners fail with this requirement. This results in many syntax errors during script development.

And all statements must be closed with a semicolon ; (exception: statements, which are in front of the ending } bracket of a code block). To simplify the code, you may terminate any statement with a semicolon. The lines shown below are valid JScript statements:

```
value = 10;  
value = value + 10;  
Tax = 0.1;
```

## Continued lines

JScript doesn't know a special character (as VBScript) to mark statements which are continued over several lines., because the semicolon identify the end of a valid statement clearly. Therefore you may break a statement into several lines. Below I have used this to create a valid statement:

```
WScript.Echo ("Hello",  
    "I was here");
```

**NOTE:** I shall mention that a continued line may not contain a comment! And, if a statement contains a string, the line break must not be within this string. In this case you may break the string into several sub strings and concatenate these sub strings using the + sign.

## Several statements per line

You may use the colon to separate several statements in VBScript (see previous appendix). But this doesn't work for JScript. If you like to put several statements into a single line, you must separate the statements with commas or semicolons. The following code sequence shows how to do this:

```
var x = 15, y = 20;  
WScript.Echo (x + y); WScript.Echo ("done");
```

The sequence produces a message box showing the value 35. To keep your programs transparent and more readable, I recommend (beside other techniques like "talking" variable names) however dispense this construction. Or do you understand immediately the following line?

```
For (var i = 0; i <= 10; i++, j++)
```

Here I have used the comma within the loop to increment the variable *j* for each pass. It should be no problem to move the statement to increment the variable *j* into the body of the loop.

**NOTE:** In my opinion this is one reason why C programs using constructions as shown above are difficult to read.

## Constants

Constants are numbers or strings within a JScript statement. JScript knows several possibilities to define constants.

```
Result = 15 + 10;  
Name = "Born";  
Pi = 3.14;
```

The first line shown above contains the two constants 15 and 10, which are added, and the result is assigned to the variable *result*. In line two we assign a string constant "Born" to a variable. The last line contains also a constant with the value 3.14.

**NOTE:** JScript doesn't know predefined constants like VBScript. If you like to use symbolic constants like *vbOkonly* within your script, you must declare such constants as variables. I will use this technique within the samples to improve the readability of the source code.

## Variables

Variables allow you to store values in memory and identify it with a name in JScript. Variables must be declared in JScript before the first use. The declaration may be done either implicitly using an assignment statement or using the *var* keyword. The following lines use this technique:

```
Price = 17;  
Tax = 16;
```

A good programming practice however declares a variable explicitly using the *var* keyword. This is used within the following line:

```
var text;      // declare a variable without assigning a value  
var x = 19;    // declare a value and set its value  
var Price = 19;  
var y = Math.sin(x);  
var x += y;    // the shorthand version  
var x = x + y; // and the more readable version  
var text = "Value ";
```

The first statement defines just a variable (its value is set implicit to zero). The other lines contain variable declaration with assignment statements defining also the variable's value and type.

## Remarks about the variable scope

Perhaps you wonder, why we distinct between implicit and explicit variable declaration? Why should I define a variable using the *var* keyword, if it is sufficient to use the variable name in an assignment statement? Here's an argument, which puts out the small, but fine difference: The kind of the declaration influences the scope of a variable. The scope determines where you can access a variable:

- ◆ A variable declared within a function with *var* (for instance *var sumx = 0;*) is only valid within this function.
- ◆ If you use an implicit variable declaration (for instance *Vat = 16;*) within a function or on script level, the variable name has a global scope. This means you may use this variable within the whole script.

I recommend using the *var* keyword to declare a variable within a function. This restricts the scope of a variable to the function level. If you need a global variable, declare its name using a *var* statement in the script's header. This improves the readability and the maintenance of your script programs. In this case you can for instance define some pseudo »constants« in the program's header, whose value may be amended easily.

## Variable names

JScript is a case-sensitive language, therefore you must take care how to spell a variable name with lower case and upper case letters. The variable *Born* is not equivalent to the name *born*. JScript requires the following rules for variable names:

- ◆ The first character of a variable name must be a letter, an underscore (\_) or a Dollar sign (\$). The name *Born12* is valid whilst *123* isn't a variable name (because this name doesn't begin with a letter and it's nothing else than a number).
- ◆ The other characters in a variable name may be letters, numerals, the underline character or the Dollar sign. However blanks and other special characters like umlauts, +, -, \* and so on are not allowed within a variable name. So *My name* is not a valid variable name, because it contains a blank. Instead you must write the name as *My\_name*.

In JavaScript a variable name may be no longer as 32 characters. In JScript a variable name may contain an unlimited number of characters. For obvious reasons you should prefer however names between 8 and 15 characters in length (this saves a lot of typing and reduce the probability for misspellings).

And you should name your variables in a sense full manner. Do you remember half a year later what a variable name *x1* mean? By the way, here is another restriction about variable naming conventions: A name must not correspond to reserved JScript keywords. The table below contains a list of reserved keywords. Compared with the JScript language reference, this table contains a few additional keywords from JavaScript. It will be wise to recognize these keywords also as forbidden for your variable names.

abstract	Enum	int	super
boolean	Export	interface	switch
break	Extends	long	synchronize
byte	False	native	this
case	final	new	throw
catch	Finally	null	throws
char	Float	package	transient
class	For	private	true
const	Function	protected	typeof
continue	Goto	public	try
debugger	If	return	var
default	Implements	short	void
delete	Import	static	while
do	In		with
double	Instanceof		

else

**Table D.1.**

*Reserved keywords in JScript*

The variable names *\_pagecount* or *Part9* are valid, whilst *19year* is illegal because the first character is a numeral. If you declare a variable without assigning a value, the interpreter creates the variable in the memory, but the value will be set to *undefined*. Using such an uninitialized variable on the right side of an assignment causes trouble:

```
var factor;           // value still undefined
var Price = 100 * factor; // Price will be set to "NaN"
```

The statements shown above demonstrates this. The value of the first variable *factor* is still undefined. Therefore the interpreter assigns the value *NaN* to the variable *Price*. *NaN* is the abbreviation for »Not a Number«. If you like to set a value during variable creation, assign a value *null* or any other value during variable declaration:

```
var fact1 = null;      // assign a special value null
var note = 3 * fact1;   // value is set to 0
```

Also: Using an implicit variable declaration keeps the risk to use an illegal value. Let's have a look into the following code sequence:

```
Name = "";           // implicit variable declaration
var aMess = Name + first_name;
```

The second line causes a run-time error, because the 2nd variable *first\_name* is declared implicitly but the interpreter doesn't assign a value.

## Remarks about values and data types

JScript variables doesn't own a fixed data type, the language uses a *Variant* data type. Therefore you can't define an explicit data type during declaring a variable. The *Variant* data type keeps variable values in the required format (numbers, strings, dates and so on). The JScript interpreter uses an implicit type conversion, if necessary. In some cases you may force this automatic type conversion. Numbers may be embedded without problems into a string (like "Text" + 99). If you like to assign a string like "99" to a numeric value, you must use the functions for type conversion (*pursuant()* and *parseFloat()*).

The code sequence shown below uses a type conversion to assign a numeric variable to a string:

```
var from = 1;
var till = 10;
var action = "Count from ";
action += from + " till " + till + ".";
```

Executing this code sets the variable *action* to the string "Count from 1 till 10.". The numeric values are converted into strings. The code sequence shown below sets the variable *x*:

```
var x = 0;
x += 1 + "10";
```

Executing these statements assigns the value "0110" to *x*. I found this statement within the JScript language reference. The statement is really tricky (and should not be used, to keep your programs readable). The term *1 + "10"* on the right side of the assignment statement concatenate a numeric

value with a string. The JScript interpreter converts the numeric value to "1" and returns the string "110". Now this string must be assigned to the variable *x*. The assignment operator is preceded by a *+* character, which forces an addition of the new value to the value already contained in the variable *x*. *x* contains already the numeric value 0 set in the first line. To add the string evaluated on the right side of the assignment, the current value of *x* must be converted to a string "0". Then the new value "0" is concatenated with "110" using the *+=* operator. So the result is "0110".

**NOTE:** The discussion above shows what you can do with JScript. C coders would like this language. But I won't recommend to use such a programming style. Instead try to use a clear programming style. The half-second you will save entering the statement is nothing to the overhead you need to debug your programs.

## Remarks about data subtypes

JScript uses only a few subtypes for variables and constants:

- ◆ *Numeric:* You may insert constants like 423 or 3.14159 directly into your source code. JScript supports both integer and floating-point numbers. Constants may be written using different radix. If a number begins with the characters 0x (or 0X – a zero followed by the letter x), it indicates a hexadecimal number (the number may contain the characters 0 till 9, A, B, C, D, E and F). A number beginning with a 0 (zero without the following x character), defines an octal value (this number may contain only numerals between 0 and 7). Decimal numbers are represented with numerals between 0 and 9. Floating-point numbers contain either a decimal point, an optional "e" or "E", used to represent the exponent (for instance 12.30, 10.0E20 or 20E-10). And you may use the signs »+« or »-«.
- ◆ *Boolean:* Variables of this type contain the constants *true* or *false*. The result of a compare operator may deliver also a boolean value.
- ◆ *Strings:* These variables are defined with an assignment of string constants like *"This is a text"* or *'1234'*. Strings are enclosed in JScript with quotes ' or double-quotes ".
- ◆ *Null:* This is a special value belonging to an uninitialized variable.

These data types are sufficient to write your JScript programs. A detailed discussion of data types may be found in the JScript programmer's reference.

## Special escape characters within strings

If you use strings, you should know also a few characters which have a special meaning in JScript. For instance " or ' indicates the beginning or the ending of a string.

\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Tab
\'	quote
\"	double-quote
\\	Backslash

The backslash character is used as an escape character, and the following character is inserted into the string. To insert a backslash into a string, you must use the `\\` combination. The double backslash character is important for instance, if a string contains a path definition. The string `"C:\\name"` causes an error in JScript, because the characters `\\n` are interpreted as »new line«. So you must use `"C:\\\\name"` instead. The characters `\\n\\r` may be used within a string to format a message box. If you need to insert a quote or double-quote into a string, you must write `\"` or `\\`. The statement:

```
Text = "He says: "WSH is cool!""
```

causes a run-time error, because the interpreter recognizes two strings and a constant, which could not be resolved. To insert the `"` into the string, you must write the statement as:

```
Text = "He says: \"WSH is cool!\""
```

The interpreter detects the `\"` sequence and insert the double-quote.

## Expressions and operators

JScript allows expressions and operators within a statement. The following section describes these possibilities in JScript.

### Assignment operator

We used the assignment operator (`=`) already during a variable definition. The statement:

```
var tax = 17;
```

defines a variable and assigns the value *17*. Below I will discuss how the assignment operator may be combined with additional operators (like `+=`).

### Comparison operators

*If* statements using comparison operators for instance to check two values, and the operators return a boolean value (*true*, *false*). JScript supports the following comparison operators:

<code>==</code>	equal
<code>!=</code>	not equal
<code>&gt;=</code>	greater than and equal to
<code>&lt;=</code>	less than and equal to
<code>&lt;</code>	less than
<code>&gt;</code>	greater than

The next statement shows the use of such an operators:

```
if (tax == 17) flag = 1;
```

If the variable *tax* is equal to *17*, the variable *flag* is set to *1*.

**NOTE:** Using the operators `==` and `!=` causes sometimes an automatic type conversion. If you like to suppress such an automatic type conversion, you must write the operators like `===` and `!==`.



## Calculation operators

Within a calculation you need calculation operators. The most simple calculation operator is the + operator (which we have used above). The statements shown below contain a few calculation operators:

```
var price = 10 + 1;  
end_price = net * (1.0 + tax);  
net = price - discount;  
var res = 100 / 25;
```

These statements use the calculation operators +, -, \* and /. JScript uses the common rules to evaluate an expression with several operators (\*, / have a higher priority as +, -). But you may use parenthesis ( ) to group sub expressions, which are evaluated first. JScript supports the following calculation operators:

+	Addition (a = a + b)
-	Subtraction (a = a - b)
*	Multiplication (a = a * b)
/	Division (a = a / b)
%	Modulo-Division (a = a % b)

These operators may be combined with the assignment operator = (same as in the C language). So it is valid to write for instance +=. Samples will be given on the following pages.

**NOTE:** A string concatenation need to be done using the + operator (like `var name = "Günter" + " Born"`). This is different from VBScript, where it is recommended to use & for concatenation (although the + operator works in VBScript). The VBScript ^ exponent operators is provide in Jscript by `Math.pow(base, exponent)`.

## Increment and decrement operators

To add or subtract 1 from a variable, you may use the increment or decrement operator.

++i	increments i
--i	decrements i
*=	multiplikation-incrementation
/=	division-incrementation
%=	modulo division-incrementation

These expressions are a bit unaccustomed for many Pascal or Basic programmers. Since it saves time during writing the code, these operators loved particularly by C programmers. I recommend however to use the familiar operators to assign and add a value (`i = i + 1`). The following lines contain a few expression using both types of operators:

a += b	is equivalent to a = a + b
a -= b	is equivalent to a = a - b
a *= b	is equivalent to a = a * b
a /= b	is equivalent to a = a / b
a %= b	is equivalent to a = a % b (Modulo-Division)

`a = ++i` increments `i` with 1 and assign it to `a`

`a = i++` assigns the value of `i` to `a` and increments `i` with 1 afterward

`a = --i` decrements `i` with 1 and assign the result to `a`

`a = i--` assigns the value `i` to `a` and decrements `i` with 1 afterward

The increment-/decrement operators shown above comes in handy also within loops.

**NOTE:** The position of the + or - sign influences the way in which a value is threaten. Within `a = ++b;` the ++ operator is in front of a variable. The interpreter scans the expression from the left to the right. The ++ operator comes first, so it is used to increment the variable `b`. Afterwards the value will be assigned to the variable `a`. The statement `a = b++;` causes that the value of `b` is assigned to `a`. Then the value of `b` is incremented.

## Logical operators

Sometimes logical operators are required (for instance in bit operations). JScript supports the following operators:

<code>&amp;&amp;</code>	<i>And</i> operator
<code>  </code>	<i>Or</i> operator
<code>&gt;&gt;</code>	Bits shift right
<code>&lt;&lt;</code>	Bits shift left
<code>&gt;&gt;&gt;</code>	unsigned bit shift right
<code>!</code>	Logical <i>Not</i>
<code>~</code>	Bitwise <i>Not</i>
<code>&amp;</code>	Bitmask with <i>And</i>
<code> </code>	Bitmask with <i>Or</i>
<code>^</code>	Bitmask with exclusive <i>Or (Xor)</i>

**NOTE:** A sample how bit operators *And*, *Or* or *Xor* works is discussed in the previous appendix. JScript uses a predefined priority list for the operators. These priorities are used, if a statement contains several operators, which are not set into parentheses. **Table D.2** lists these priorities. The first entry owns the lowest priority.

Operators
Comma ,
Assignments: = += -= *= /= %= <<= >>= >>>= &= ^=  =
Condition ? :
Logical Or

logical And &&
bit wise Or
Bit wise Xor ^
bitwise And &
equal, not equal == !=
Relational < <= > >=
Bitwise shift << >> >>>
Addition/Subtraction + -
Multiplication/Division * / %
Negation/Increment ! ~ - ++ --
Call, Member () [] .

**Table D.2.**  
*operator precedence*

## Control structures

Only in a few rare cases your scripts will be linear. Most of your scripts contains branches and also control structures to decide which branch must be executed. Below I will discuss JScript control structures.

### if statement

The *if* statement may be used in different versions. The code:

```
if (condition)
{
    statements, if condition is true
}
```

tests the condition. If the condition, which must be set into parenthesis () and may contain the comparison operators mentioned above, is *true*, the statements within the *if* block are executed. If the block contains only one statement, this statement may follow in the next line. Blocks containing several statements must be enclosed with a curly bracket { }. This is shown in the following sequence:

```
if (value <= 16.0)
{
    WScript.Echo ("Sorry, you loose the game");
    value = 0
}
```

Within this code snippet, the variable *value* will be set to 0, if the current value is less than or equal 16. Here we use the brackets to set a block of statements. By the way, the semicolon within the last statement within a block is optional, because the ending `}` defines the statement's end.

If you need a condition, which executes one of two branches, you must use the *if .. else* structure. The statement uses the syntax shown below:

```
if (condition)
{
    statements, if condition is true
}
else
{
    statements, if condition is false
}
```

The *if* statement tests the condition. If the result is *true*, the statements within the block following the *if* statement are executed. If the condition is *false* the statements in the *else* block are executed:

```
if (value <= 16.0)
{
    WScript.Echo ("Sorry, you loose the game");
    value = 0
}
else
{
    WScript.Echo ("Congratulation, you won");
}
```

**TIP:** One reason for program malfunction is a wrong nesting of statements within the *if* block. You may omit the bracket `{ }`, in case that *if* or *else* is followed by one statement.

## Conditional operator

JScript supports a conditional operator, which assigns a value depending from a condition. The operator uses the following syntax:

```
(condition) ? value1 : value2
```

The parentheses contains a condition. The parentheses is followed by a question mark. If the condition is *true*, *value1* in front of the colon is used. Otherwise the second value is used. This is shown in the next line:

```
status = (age >=18) ? "Adult" : "Child";
```

If *age* is above or equal 18, *status* will be set to *Adult*. Otherwise *status* will be set to *Child*.

## for loop

*for* loops may be used to repeat a block of statements in a defined manner. The number of repeating are defined with a counter. The *for* loop uses the following syntax:

```
for(var count = 1; count <= 100; count++)
```

```
{
  statements
}
```

The variable *count* in the header of a loop is set to the start value during the program flow enters the loop. The keyword *var* defines a local variable for the counter used within the loop body. During each pass the value of *count* will be incremented (this is caused by *count++*) or decremented (this is caused using *count--* in the third parameter). Because ++ is following the variable name, the variable value is used first, and then it is incremented. The end condition for the loop is defined in the second parameter (*count <= 100*). The next listing shows how a loop may be used within a WSH script.

```
//*****
// File:      WSHDemo.js (WSH sample in JScript)
// Author:    (c) Günter Born
//
// Trace the program with message boxes
//*****
// The next statements enables/disables trace messages

// variables declared here are global
// var DebugFlag = false;    // disable trace
var DebugFlag = true;      // enable trace (

j = 0;
debug ("Start", 0, 0);

for (var i = 1; i <= 10; i++)    // try 10 loops
{
  debug ("Step: ", i, j);
  j = j + i;                    // Add values
}

debug ("End", i, j);

WScript.Echo ("Result: ", j);

WScript.Quit ();

function debug (text, count, val)
{
  if (DebugFlag)                // Debug mode enabled?
    WScript.Echo (text, i , "Interims result: ", j)
}
// End
```

**Listing D.2.**  
*Using a for loop*

## for ... in loop

A *for in* loop may be used in JScript to access elements (objects in a collection or items in arrays). This loop uses the following syntax:

```
for (variable in [object | array])
{
    statements
}
```

The keyword *in* is followed either by the name of a JScript object or of an array. The *for* loop causes that (a reference to) each item within the collection/array is assigned to the variable *variable*. The loop terminates, after all elements are processed.

## while loop

*while* loops are executed till a loop condition becomes *false*. This loop uses the following syntax:

```
while (condition)
{
    statements
}
```

The condition is tested during each pass. If the condition is *true*, the statements within the { } block are executed. Equivalent to the *for* loop you may omit the bracket { }, if the loop contains only one statement. The sample below shows how to use this kind of loop:

```
var i = 0;

while (i <= 10)    // try 10 passes
{
    WScript.Echo ("Step: ", i);
    i++;           // Add values
}
```

The program loops till the index is set to 11. Each pass shows the current index within a message box.

## do ... while loop

*do ... while* loops work similar like *while loops*. Such a loop may be used to process a block of statements several times, till the condition becomes *false*. The following syntax is used:

```
do
{
    statements
}
while (condition);
```

Comparing to a *while* loop, the condition in a *do .. while* loop is tested on the loop's end. This means, the loop will be executed at least only once (even the condition is false). If the condition is *true*, the statements within the { } block are executed again. If the loop contains only one state-

ment, you may omit the { } brackets. The next code snippet is derived from the JScript tutorial and it shows how to use this loop to process all drives of a *Drives* collection:

```
function GetDriveList()
{
    var fso, s, sharename, objDrives, drive;    // declare local variables
                                                // use FileSystemObject
    fso = new ActiveXObject("Scripting.FileSystemObject");
    objDrives = new Enumerator(fso.Drives);    // get drives collection
    s = "";                                     // init result variable
    do                                         // the loop
    {
        drive = objDrives.item();             // get item
        s = s + drive.DriveLetter;             // store drive letter in result
        s += " - ";
        if (drive.DriveType == 3)              // shared drive?
            sharename = drive.ShareName;        // use share name
        else if (drive.IsReady)                // local fixed drive ?
            sharename = drive.VolumeName;        // yes, use volume name
        else
            sharename = "[Drive not ready]";    // removable drive
        s += sharename + "\n";                 // add new line
        objDrives.moveNext();                  // skip to next item
    }
    while (!objDrives.atEnd());                // test end condition
    return(s);                                // terminate and return result
}
```

## switch statement

The *switch* statement offers the possibility to execute several blocks of statements depending on the value of an expression. *Switch* uses the following syntax:

```
switch (expression)
{
    case label :
        statements
    case label :
        statements
    ...
    default :
        statements
}
```

At the beginning of the construction the condition is given. Depending on the value of this condition one of the *case* branches is selected and the statements within this block are executed. The identifier *label* is a placeholder for the value of the expression. If the expression is equal to the given value in *label*, the block is executed. If no block fits the condition, the statements within the *default* branch are used. The next sequence shows how to use the *switch* statement:

```
function Test(x)
```

```
{
  switch (x){
    case 1:
      ...
    case 2:
      ...
    case 3:
      ...
    default:
      ...
  }
}
```

Here we use a variable *x* for the condition. If *x = 1*, the first branch in *case 1:* is executed. On *x = 2* the next branch is processed and so on. A *break* statement (like in C) isn't needed to prevent fall conditions from falling through.

## ***break* and *continue***

In JScript you may use the (optional) *break* keyword to terminate a loop unconditionally (*break* is similar to the *Exit* statement in VBScript). If the interpreter detects this keyword, the current loop terminates and the code following the loop is executed.

The *continue* keyword causes the opposite. If the interpreter recognized this keyword, the program control is transferred immediately to the beginning of the loop. The loop index will be incremented or decremented, and the loop is executed again.

# Built-in-objects and -functions

JScript supports a few built-in objects (*String*-object, *Math*-object, *Date*-object) and functions. Below you will find a few information about these topics.

## Functions

Functions combines several operations under one name and may be called from a program using the function name. If the function terminates, a result is returned. JScript supports user-defined and built-in functions. User-defined functions are defined according to the following syntax:

```
function test (arguments)
{
  body with statements
  return;
}
```

A function is declared with the keyword *function* followed by a function name and an argument list set in brackets. The statements within the function body must be enclosed in curly brackets { }. The *return* statement terminates a function.



The function may be called within a script. You must insert the function name and the parameters set in brackets. Because a function returns a value, the function call must always be on the right side of an assignment statement. The next line calls the function *test* for instance:

```
result = test ();
```

Parameters must be set in parenthesis. If the function doesn't require parameters, an empty bracket must be used. Commas separate several parameters.

## Built-in-functions

The JScript language definition contains a few Built-in-functions, to handle expressions, special characters or convert strings and numeric values.

The *eval* function evaluates a string, which is passed as an argument (parameter), and returns the evaluated value (for instance *value = eval("14+15");*). *parseFloat* gets the argument and tries to return a floating-point value. If the string argument contains an illegal character (not +, - or 0 to 9, . or e), the string will be converted only from the begin to the illegal character. If the argument doesn't contain a number, the *NaN* (Not a Number) value is returned. The function *parseInt* requires a string as the first argument. The second argument must contain the code for the base (10=decimal, 8 = octal, 16 = hexadecimal etc.). An illegal string causes a result *NaN*. *parseInt* returns always an integer value.

**NOTE:** A detailed description of all built-in functions may be found in the JScript language reference (<http://msdn.microsoft.com/scripting>).

## Objects

Beside functions JScript also supports a few built-in objects to process strings, execute mathematical operations or to manipulate date and time values:

- ◆ The *String* object is used, if a string is assigned to a variable or property (like *name = "Born";*). The object supports several methods to manipulate strings.
- ◆ The *Math* object offers methods and properties for mathematical operators (for instance *value = Math.PI;* assigns the property *Pi* to a variable).
- ◆ The *Date* object is used to handle date and time values (for instance *var Name = new Date(parameters);* creates a new date object, *today = new Date();* returns the date).

In JScript you may handle objects and arrays in the same way. You may access also objects and collections in a similar way. To access a method or property of an object, you must insert the object name, followed by a dot and the name of the property/method into the code. The statement:

```
WScript.Echo ("Hello");
```

uses the *Echo* method of the *WScript* object to display a text within a message box.

**NOTE:** Here I should point out that JScript requires parenthesis to submit parameters to a procedure (method), whilst VBScript doesn't require this parenthesis.

Accessing an object within a collection, you can use an index value. The following statements are equivalent:

```
Res = Object.width;  
Res = Object[3];    // [3] should be equivalent to index "width"  
Res = Object["width"];
```

While the bracket are valid through accessing the numerical index, the dot must be omitted, if an index value is used. Therefore the next statement causes a syntax error:

```
Res = Object.3;
```

If an object contains another object as a property, the naming scheme must be extended:

```
var x4 = toDoToday.shoppingList[3].substring(0,1);
```

The object property is followed by a dot, followed by a sub-object. Arrays may be handled very easy in JScript. The next statement defines an array:

```
var name = new Array(17);
```

The array contains the items with the indexes from 0 to16. You may use the following code to assign a value to the first item:

```
name[0] = "Born";
```

The number of array items may be estimated using the following code:

```
number = name.length;
```

The next statement creates a multi-dimensional array:

```
var namex = name [3][7];
```

**NOTE:** Here I like to terminate the introduction into JScript. You may get the whole JScript language reference including a tutorial from Microsoft's web site <http://msdn.microsoft.com/scripting>. But take care, there are different JScript language engines available. Windows 98 and Internet Explorer 4.0 installs the version 3.x of the JScript language engine. Internet Explorer 5.0 comes with the JScript language engine version 5.0. And you can download the latest language engine from the web. Version 5.0 supports a few new features like error handling. Further information may be found in the language reference.