

C Introduction into VBScript

Microsoft offers a VBScript Language Reference and a Tutorial with detailed information about the language. The purpose of this chapter is to give a brief introduction into VBScript. You may read the following pages, if you have never used VBScript or if you like to check special language constructs (like how an if-statement must be written in VBScript).

TIP: I recommend downloading the VBScript language reference as a CHM file from the Microsoft Web site <http://msdn.microsoft.com/scripting>.

Statements, continued lines and comments

VBScript is a subset of Visual Basic for Application (VBA, the language provided from Microsoft Office), and VBA is a subset of Visual Basic (VB). Microsoft has removed several language constructs from VB/VBA to define the VBScript syntax. If you have programmed in VB or VBA, VBScript programming should be no major problem. Below you will find some basic remarks about the structure of VBScript programs.

VBScript statements

VBScript statements must be entered using the syntax rules defined for this language. The following lines contain valid VBScript statements:

```
Value1 = 10
Value1 = Value1 + 10
If Value1 > 100 Then Value1 = 100
Tax = 0.1 : Price = Net * Tax
```

You may enter several statements within one line, if you separate these statements with colons ":". This was used above for instance in the last line. I recommend using several statements within a line sparsely to keep your programs more readable.

NOTE: In contrast to JScript the VBScript statements and keywords are not case-sensitive. Decimal values are to be indicated in the VBScript with a dot (like 14.34).

Continued lines

Using very long statements reduce the readability of your script. The following statement:

```
MsgBox "You have entered a wrong name into the text box shown in the previous form",  
vbOkOnly, "Input Error"
```

shows a message box. Unfortunately the statement is longer as the current line. If a program contains many extra large lines, the readability gets worse (you have always to scroll horizontally to

edit such lines). But you may append a blank followed by the underscore _ character to the end of a line and continue the statement within the next line. This is used in the statement below:

```
MsgBox "You have entered a wrong name into the text box " + _
"shown in the previous form", _
vbOkOnly, _
"Input Error"
```

Each time the language engine detects a line ending with an underscore, a continued line is assumed. Then the next line is threatened as the next part of the current statement. You may use the underscore character to divide a statement up to 10 lines.

NOTE: The last line of a continued statement may not have the appended underscore. Keep also in mind that no comments may be appended to a line containing an underscore character. If you like to use a long string in continued lines, split the string into several sub strings. Each sub string must be terminated with a double-quote. Then you may insert the underscore to divide the statement into several lines. The sub strings may be concatenated using the & or + operator (like "... " + _). I have used this structure in the sample statement shown above.

Comments

Would you like to tell VBScript not to interpret a line or a part of a statement? Just need to write this statement as a comment. In VBScript a quote ' or the *REM* statement are used to mark a comment. If the VBScript interpreter detects this character within a statement, it ignores the rest of the line. Both of the following lines contain comments:

```
' This is a whole line containing a comment
Value1 = Net * Factor      ' comment at line end
```

The second line contains also a comment, which follows the statement. This means, VBScript executes the statement at the beginning of this line and ignores the trailing comment. This may be used to comment statements within a script.

Remarks about the structure of a VBScript program

A VBScript program script may consist of comments and statements. The script files used in this book comes with an extraordinary structure however. This structure is used in the listing shown below:

```
' *****
' File:    WSHTest.vbs
' Author:  (c) G. Born
'
' Demonstrates how the WSH may be used to show
' a simple dialog box.
' *****

Dim WSHShell      ' declare objects
Dim Message
Dim Title
```

```
' init variables
Message = "Hurray, it works"
Title = "WSH sample - by G. Born"

' Now we try to use the MsgBox function
' MsgBox prompt, buttons, title
' prompt:    Text shown in the dialog box
' title:     Title shown in the box
' buttons:   the buttons

MsgBox Message, _
    vbInformation + vbOKOnly, _
    Title

WScript.Quit() ' terminate the script
'* End
```

Listing C.1.*VBScript program*

As I have mentioned in previous chapters, it's always a good programming practice to add comments to your scripts. If you got a foreign script, you will surely welcome, if the author has noted the purpose of the script and other information in the programs header.

NOTE: Also let me mention that you can use VBScript programs within HTML documents and execute these scripts within the Internet Explorer (MSIE). Although the same language engine is used for Internet Explorer and WSH, the objects (or more precisely the object model) exposed from WSH and MSIE differ. For instance, the whole event handling provided from the Internet Explorer is useless within the WSH environment.

Variables and constants

In VBScript you may use variables and constants to store values. Below I will discuss how to defined variables and constants.

Constants

You may use values directly as constants in VBScript statements. The following code defines such a constant:

```
result = Price * amount + 100.0
```

The value 100.0 is used as a constant within the formula. Often however the programmer wish to set a constant or identifier in the VBScript program header. The constant's value is protected from being changes from executing code. A constant must be declared explicitly.

```
Const profit = 100.0
```

The keyword *Const* invokes the constant declaration. This keyword is followed by the constant's name and the constants value. A named constant declared in this way may be used in any further statement within the VBScript program.

```
Price = NetPrice * Amount + profit
```

The advantage of named constants is that you may change the value of a constants within the declaration, instead of amending constant values spread over the whole source code. And you can assign meaningful names to a value (this simplifies code maintenance). It's also possible to define several constants within a single line:

```
Const VAT = 0.16, Profit = 10
```

VBScript supports (in contrast to VB or VBA) only the *Variant* data type (see below). Named constants are declared as public by default, this means the constant is available to the whole script. If you declare a constant within a procedure, the scope is valid only within the procedure. You may overwrite this default scope using the *Public* and *Private* keywords (see also the following pages).

TIP: Numerical constants are written usually in the decimal system (10.14 for instance). You can use however also constants in the hexadecimal system in the format *&Hxxx*, where *xxx* stands for a hexadecimal number. The value *&H0C* corresponds for instance with the decimal number 12 (see below »Comparison Operators«).

Intrinsic constants

VBScript contains several predefined constants (named Built-in or intrinsic constants) like *vbOkOnly*, which may be used within your scripts. The statement:

```
MsgBox "Hello", 0 + 64, "Test"
```

is much more cryptic compared to a statement containing named constants like it is shown below:

```
MsgBox "Hello", vbOkOnly + vbInformation, "Test"
```

The only difficulty during writing the code is that you must know the exact name of the required constant.

TIP: Download the VBScript language reference from the Microsoft Web site <http://msdn.microsoft.com/scripting>. This language reference contains also a list of intrinsic constants.

Variables

Variables are placeholders that refer to a memory location where a program can store values. Variables may be (in contrast to constants) change its value during program operation. A variable may be used directly within a program:

```
Price = 45      ' set the price
Discount = 17
```

As soon as the name of a variable occurs the first time within the program, VBScript creates the variable in the computer memory and assigns an initial value. VBScript supports only the *Variant* data type for a variable. This means you can store different values like numbers, texts and so on into a variable.

Some remarks about VBScript data types

Compared to Visual Basic or VBA, VBScript supports only one *Variant* data type. A Variant is a special kind of data type that may contain different kind of information. The format of a value stored in a Variant depends on the value. If you assign a number to a variable, it will be stored in a numeric format. A date value will be stored in the date format, texts are stored in a string format and so on.

NOTE: For C/C++ programmers: The Variant data type class is implemented as a union of many data types

VBScript tries to accomplish the most logical operation on variables. For instance: If you add the content of two variables with numerical values the result becomes also numerical. The statement:

```
Sum1 = Price + 15.0
```

stores a numerical result within the variable *Sum1*. The statement:

```
Result = "Value " + Sum1
```

causes a problem: The first part of the expression on the right side of the assignment is a string, whilst the variable *Sum1* contains a numerical value. The VBScript interpreter can't assign a result, so an error message is shown (**Figure C.1**).

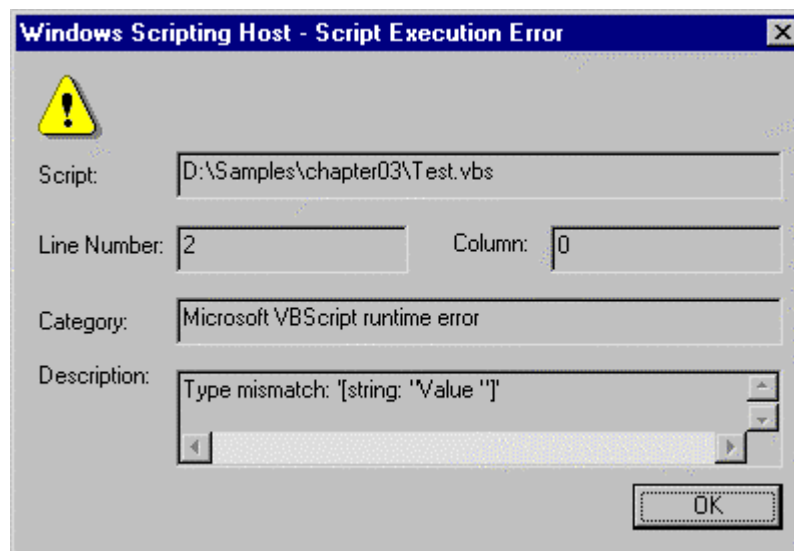


Figure C.1.
Error message as a result of type mismatch

In such a case you may convert the subtype using the following statement.

```
Result = "Value " + CStr(sum1)
```

The VBScript function *CStr()* converts the numerical value contained in the variable *sum1* into a *Variant* data type with the string format. This result may be concatenated with the second string "Value " and the result may be assigned to the variable *Result*.

NOTE: Concatenating strings using the "+" operator isn't a good programming practice (although I use it sometimes). The "+" operator »add« some operands, which is a concatenation operation for strings. A better approach uses the "&" operator for string

concatenation (like "Hello " & "World"). In this case VBScript will do the type conversion automatically (without using CStr()).

You can treat also numbers as strings, if these are defined as constants. To do this you must enclose the constants in quotation marks. The statement:

```
Text = "15" & "30"
```

doesn't produce the value 45, instead *Text* contains the value "1530"! You can use conversion functions to convert data from one subtype to another *Variant* subtype.

Variant subtypes

VBScript supports subtypes for the *Variant* data type. I have mentioned above that a variable may contain different values like numbers, strings, date values and so on. Within a date value also a numerical value is used. But this value uses a different format compared with a numerical value. Other data types represents logical values (*true* or *false*), or integer numbers and floating point numbers.

NOTE: The difference is made within the subtypes of a *Variant*. The VBScript Programmers Reference contain a table with all subtypes supported within a *Variant* variable.

In contrast to VB or VBA you can't predefine a subtype for a variable. VBScript assigns the subtypes automatically. But you can use conversion functions like *Asc*, *CBool*, *CByte*, *CCur*, *CDate*, *CDBl*, *Chr*, *CLng*, *CInt*, *CSng*, *CStr*, *Hex* and *Oct* to convert the data types. Additional you may use the *VarType* function to query the subtype.

The *Option Explicit* keyword

Using an implicit variable declaration causes the risk to create incorrect typed variable names. If you mistype a variable name, the VBScript interpreter creates a new variable. Let's assume you wrote *Pris* instead of *Price*. The following code snippet contain this typo:

```
VAT = 16.0
Price = 0
Net = 115.00
Pris = Net * (1.0 + VAT/100)
MsgBox Price, vbOkOnly, "Price "
```

The program sequence shown above won't work. The price calculated should be assigned to the variable *Price*, and it is intended to show this value in a message box. The typo in the assignment statement causes that the price calculated is stored in the (new) variable *Pris*. Therefore the message box shows always the result 0.

To detect mis-typed variable names you can use the keyword *Option Explicit* within the first line of your program. If the interpreter recognizes the *Explicit* keyword, all variables must be declared explicitly using *Dim*, *Private*, *Public* or *Redim* statements. If you forget this declaration, the WSH reports each implicit declared variable in an error dialog. This uncovers also mistyped variable names.

```
Option Explicit
' *****
' File:      ErrorTest2.vbs (VBScript WSH)
```

```
' Author: (c) G. Born
'
' Use an undeclared (mis-typed) variable to demonstrate
' the Option Explicit statement.
'*****
Dim Message ' declare variables
Dim Title

' init variables
Message = "Hello world"
Titel = "WSH sample " ' this name isn't declared!!

' Show a message:

MsgBox Message, vbInformation + vbOKOnly, Title

WScript.Quit() ' terminate script
' End
```

Listing C.2.

Demonstrating the Option Explicit error (ErrorTest2.vbs)

The script contains a mistyped variable *Titel*. If the interpreter reach the line with the string assignment to the variable *Titel*, an error message shown (**Figure C.2**).

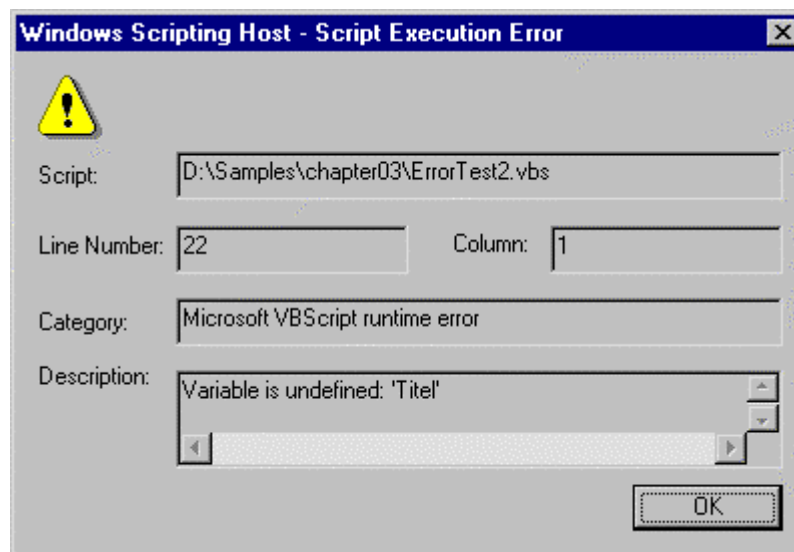


Figure C.2.

Error message indicating a undefined variable

Using the *Dim* statement

In VBScript scripts you will find sometimes the keyword *Dim*. I mentioned this keyword above. The keyword *Dim* may be used to:

- ♦ explicitly define a variable at procedure, function or script level.
- ♦ or to define an array variable.

If you use the *Option Explicit* statement, you must declare all variables using the *Dim* keyword. This may be done within the script header or inside in a procedure/function:

```
Dim text
Dim x
Dim Price, Vat
```

NOTE: During declaration of a variable its value gets automatically initialized. Numerical variables receive the value 0 whilst strings is set to an empty string ("").

Public and Private within a variable declaration

It was mentioned already in the previous section: A variable declared with *Dim* on script level is also valid within the procedures/functions. A variable declared with *Dim* within a procedure or function is valid only inside within the procedure/function. So the variable scope should be clear. The VBScript language reference knows however still the keywords *Public* and *Private*, to limit the validity of variables.

```
Public Test
Public Price, Vat
```

The keyword *Private* restricts the scope of a declared variable to the script. If you use the *Public* keyword, the variable will be known in all scripts and procedures.

NOTE: The statements *Public xvalue* or *Private xvalue* are illegal within a procedure/function and causes a syntax error. If you use the *Option Explicit*-statement, you are forced to declare variable using the *Dim* statement.

Perhaps you wonder now, what kind of sense *Private* and *Public* keywords make? The *Dim* statement suffices to determine the variable scope between procedures/functions and script level. But VBScript was developed however originally for HTML documents. And such a document can contain several scripts arranged within <SCRIPT>-Tags in the HTML code. With the keyword *Public* you can declare a variable that is valid also outside the local script in other scripts (of the same HTML document).

Array declaration with Dim

VBScript supports arrays, which must be declared in the following order:

```
Dim value (10)
value (0) = 11
```

The first line defines an array with 11 items (the index starts always with 0). The first array element may be accessed using *value(0)*. The second statement shown above assigns the value 11 to

the first array element. The *Dim* statement allows you to declare multidimensional arrays. The statement:

```
Dim value (10,10)
```

declares a two dimensional array. *Value (0,0)* is the first element in row 0 and column 0. VBScript allows up to 60 dimensions for arrays. The lower bound for arrays is always set to 0. Therefore you can set only the upper bound of an array within the *Dim* declaration.

NOTE: Using the *Dim* statement with an empty bracket (for instance *Dim value ()*) define dynamic arrays. The dimension of a dynamic array may be redefined using the *ReDim* statement.

Variable names

A variable name may contain up to 255 characters. You can choose the variable name freely, as long as the name suffices the following criteria:

- ♦ The name must begin with a letter (*Test* is a valid name whilst *123* is illegal).
- ♦ Within the variable name you doesn't may use blanks, dots, commas and some special characters (for instance *!*, *-*, *+*).

And you may not use keywords like *Sub*, *If*, *End*, *Dim* and so one for variable names.

Operators

VBScript supports several types of operators (arithmetic operators, logical operators, comparison operators, operators for concatenation). Below you will find a short introduction into these operators.

Arithmetic operators

The following table contains the arithmetic operators available in VBScript.

Operator	Remark
\wedge	Exponentiation ($x = y^{\text{Exponent}}$)
$+$	Addition ($x = a + b$)
$-$	Subtraction or negative sign ($x = -10$ or $x = a - 100$)
$*$	Multiplication ($x = b * 30$)
$/$	Division ($x = a / b$)
\backslash	Integer-Division ($x = a \backslash b$)
Mod	Modulo ($x = a \text{ Mod } b$)

Table C.1.*Arithmetic operators*

NOTE: It is possible to use the + operator also for string concatenation (for instance *Name* = "Mill" + "er"). But this can cause problems. Therefore you should use the & operator for string concatenation. If an operand contains the value *Null* or *Empty*, the result is also *Null* or *Empty*. Further information may be found in the VBScript language reference.

Assigning object references with the Set-Operator

The *Set* statement has a special meaning in VBScript. To access an object you need a reference to the object. This references must be assigned using *Set*. The following statement shows how to use *Set*:

```
Set objAdr = WScript.Arguments
```

This assigns an object reference to the variable *objAdr* which points to the *Arguments* property of the *WScript* object. The variable uses the data type *Variant*, but the subtype is set to *Object*. After assigning an object variable, you may use this reference in a statement:

```
MsgBox objAdr.Item(0)
```

The statement above shows the first argument passed to the script in a message box.

NOTE: At this point I like to give a short explanation about the dot, which appears in many names. The dot separates the names of objects, methods or properties within the statement. In the line shown above *objAdr* is the object reference, while *Item* presents a property.

Logical Operators

VBScript supports a few logical operators to evaluate expressions. The following table presents these operators.

Operator	Remarks
Not	NEGATION (x = Not y)
And	AND (x = a And b)
Or	OR (x = a Or b)
Xor	EXCLUSIVE OR (x = a Xor b)
Eqv	EQUIVALENCE (x = a Eqv b)
Imp	IMPLICATION (x = a Imp b)

Table C.2.*Logical operators*

Logical operators are used often in branches. The following statement tests two conditions:

```
If a > 100 And a < 1000 Then
```

Both conditions (a > 100 and a < 1000) deliver a logical value *true* or *false* which are compared with *And*. Only if both conditions are *true* the *IF* branch is executed.

You may use the operators *Not*, *And*, *Or* and *Xor* to calc bit operations on byte and integer values. The *Not* operator uses the truth table shown below:

Not	Bit
0	1
1	0

Table C.3.
Not operator

The *Not* function reads the bit and inverts its value. A 0 is converted into a 1 and vice versa. Such operations may be best demonstrated within the binary or hexadecimal system. A decimal number 3 may be written in a binary system as 0011 (if we use 4 digits for the representation). Using the *Not* operator creates the following result:

1100

which is equivalent to the hexadecimal value 0CH or decimal 12. The *And* operator compares two bits. If both bits are 1 the result is also 1. This is given in the next table. The input values are given in column *Bit* whilst the result is shown in column *And*.

And	Bit	Bit
0	0	0
0	0	1
0	1	0
1	1	1

Table C.4.
And Operator

The following statement uses the *And* operator:

```
MsgBox (3 And 7)
```

and the result 3 is shown in the message box. The statement within the bracket does a bit wise operation with *And*. For novices: A decimal 3 may be represented as a binary value of 0011 and decimal 7 is equivalent to the binary value 0111. So the *And* operator delivers according the table above the result 0011, which is equivalent to 3 decimal.

The *Or* operator compares two bits with OR. If one bit is 1, the result is also set to 1. The following table shows the possible results of the *Or* operator.

Or	Bit	Bit
0	0	0
1	0	1
1	1	0
1	1	1

Table C.5.
Or operator

The *Or* operation:

```
MsgBox (3 Or 7)
```

delivers the result 7 in a message box. The decimal value 3 is equivalent to 0011 decimal and 7 decimal is 0111 binary. The *Or* operator delivers 0111 binary which is 7 decimal. The last binary operator that is used in common is *Xor*. This operator causes the result 1, if both input values are different.

Xor	Bit	Bit
0	0	0
1	0	1
1	1	0
0	1	1

Table C.6.
Xor operator

Comparison operators

VBScript knows a few comparison operators. I have shown the first compare operator already within the previous section. These operators allow the comparison of expressions (which may contain numbers, strings and so on). The next table contains the compare operators available in VBScript.

Operator	Remark
<	Less then (a < b)
>	Greater then (a > b)
=	Equal (a = b)
<=	Less then or equal (a <= b)
>=	Greater then or equal (a >= b)
<>	Not equal (a <> b)

Table C.7.
Comparison operators

Comparison operators are used in branches and in loops:

```
While a < 10
..
Wend
If a > 100 Then
...
```

```
End If
```

NOTE: Keep in mind during using of compare operators that VBScript knows only *Variant* variables. If the two subtypes are of different data types, VBScript converts the data types automatically. As a result of this conversion it can be happen, that values, which are not equal, are interpreted as equal.

Operator priorities

You can use parentheses to set the priorities for the evaluation of operators. Without parentheses the parser uses implicit operator priority (Exponentiation, Negation, Multiplication/Division, Integer Division, Modulo, Addition/Subtraction, Concatenation). The previous sequence shows the order of the priorities, while Exponentiation has the highest priority. For comparing operators we will have the following priority order (=, <>, <, >, <=, >=), where the Equal sign = has the highest priority. For logical operations the *Not* operator has the highest priority, followed by *And*, *Or*, *Xor*, *Eqv*, *Imp*, &.

Control structures

VBScript supports several control structures to control loops and branches. Below I will introduce these structures.

If ... Then

The *If* statement may be used to implement a branch, depending on a compare operation. The statement:

```
If a > 100 Then a = 100
```

resets the variable *a* to 100, if its value is greater than 100. The next sequence:

```
If a > 100 Then  
    a = 100  
    b = 20  
End If
```

compares the variable *a* with 100. If the value is higher than 100, then the statements between *If .. Then* and *End If* are executed. Otherwise the program continues with the statement following the *End If* line.

If ... Then ... Else

This variant of the *If* statement may be used to create two branches. The sequence:

```
If a > 100 Then  
    a = 100  
    b = 20  
Else  
    a = a + 10
```

```
b = a \ 10  
End If
```

tests the variable *a*. If the value is greater 100, the statements between *If .. Then* and *Else* are executed. Otherwise the statements between *Else* and *End If* are used.

If ... Then ... Elself

This variant of an *If* statement allows nesting of several *If* blocks. The sequence shown below uses this construction:

```
If a = 1 Then  
    b = 100  
    c = 20  
ElseIf a = 2 Then  
    b = 200  
    c = 40  
ElseIf a = 3 Then  
    b = 300  
    c = 60  
End If
```

The variable *a* is tested for different values. If a compare operation delivers the result *true*, the statements between *ElseIf* and the next *ElseIf* or *End If* keyword are executed.

Select Case

This keyword may be used to test a variable for several conditions. Depending on the variable's value several blocks of code may be defined. The sequence below shows how to use a *Select Case* construction:

```
Select Case a  
    Case 1  
        b = 100  
        c = 20  
    Case 2  
        b = 200  
        c = 40  
    Case 3  
        b = 300  
        c = 60  
    Case Else  
        b = 0  
        c = 0  
        a = 1  
End Select
```

This code sequence tests variable *a*. The *Case* statements contain the value to be tested against the variable. If a condition is *true*, the statement within the *Case* branch are executed. If none of the *Case* conditions is *true*, the (optional) *Case Else* branch is used.

Loops

Loops are used to repeat some statements within a block. The following sections contain a short overview about the loop constructs available within VBScript.

Do While ... Loop

A *Do While* sequence creates a loop. The header of this loop contains a condition, which must be *true* to execute the statements within the loop. If the condition is *false*, the interpreter continues program execution with the statements following the *Loop* keyword. The code sequence shown below uses a *Do While* loop.

```
a = 1
Do While a < 10
    a = a + 1
Loop
```

The condition *a < 10* is tested before the loop is entered. If the condition is *true*, the interpreter executes the statements till the *Loop* keyword is reached. The condition in the loop's header is tested again. This is repeated until the condition becomes *false*. In this case the interpreter continues with the statement following the *Loop* keyword.

Therefore the condition within the loop's header must deliver the values *false* or *true* to execute and terminate the loop.

Do Until ... Loop

The *Do Until* statement creates a loop, which is tested at the entrance of the loop. If the condition is *false*, the loop is executed. The loop terminates as soon as the condition becomes *true*. The following sequence shows a sample using this statement:

```
a = 1
Do Until a > 10
    a = a + 1
Loop
```

Within this loop the condition *a > 10* is tested. If the condition isn't *true*, the statements within the loop are executed. After the condition within the loop's header becomes *true*, the program is continued with the statement after the *Loop* keyword.

Do ... Loop While

The statement *Do ... Loop While* may be used to create a loop containing the condition test at the end of the loop. If the condition at the end of the loop is *true*, the loop will be executed repeatedly. The loop terminates, if the condition becomes false. The code shown below demonstrates the use of this keywords:

```
a = 1
Do
  a = a +1
Loop While a < 10
```

This loop tests the condition $a < 10$ at the end of the block. If the condition is *true*, the statements between *Do* and *Loop* are processed again. If the condition is getting *false*, the next statement following the loop is executed.

Do ... Loop Until

With *Do ... Loop Until* you may create a loop which is tested at the end of the block. If the condition is *false*, the loop statements are processed again. If the condition becomes *true*, the loop terminates, and the statement following the loop gets executed:

```
a = 1
Do
  a = a +1
Loop Until a > 10
```

The code shown above tests the condition $a > 10$ at the loop's end. The loop is processed till the condition becomes *true*.

Exit Do

The *Exit Do* keyword may be used within all *Do* loops to terminate the loop. If the interpreter recognizes this statement, the loop is terminated and the statement following the loop is executed.

For ... Next

A *For* loop may be used to process a predefined number of steps. All statements within the *For ... Next* block are processed during each step. The sequence shown below demonstrates how a *For* statement may be used within VBScript:

```
For i = 1 To 10
  a = a +1
Next
```

The loop is repeated 10 times. The value *i* contains the loop index. The step width of a *For* loop is set to 1 by default. But you may use the *For i = start To end Step x* construction to set implicit the step width to the value *x*.

For Each ... Next

Some other important construction is the *For Each* sequence. These sequence is used to create a loop to process all elements within a collection or within an array. The loop will be repeated for each item within the collection/array. The next lines shows how to use such a loop:


```
For Each x In Worksheets
...
Next
```

Exit For

The statement *Exit For* may be used to terminate a *For* loop. If the interpreter detects this statement, the next line following the loop is executed.

While ... Wend

You can use a *While ... Wend* loop to process a code sequence several times. The loop terminates, if the condition in the line containing the *While* statement becomes *false*. The code below shows how to use this kind of loop:

```
Dim value
value = 1
While value < 10
    value = value + 1
...
Wend
```

The loop shown above is repeated until *value* becomes 10. You may also use the *Do ... Loop* statements to get the same result.

Procedures and functions

In VBScript you may use built-in procedures and functions, and you may define your own functions and procedures. Below I will discuss the basics to create user defined procedures and functions VBScript.

Functions

Functions are to be used, if only one result (which can be a variable or an array) is returned to the calling program. A function is declared with the following statements:

```
Function Name (Arguments)
...
    Name = result
End Function
```

The return value must be assigned within the function body to the function name. This is done in the sequence above using the *Name = result* statement. VBScript uses always a *Variant* data type for the return value. The first line defining a function must contain the keyword *Function* followed by the function name (*GetValue* for instance), followed by a bracket (). Within the bracket you may declare parameters (also called arguments), which are required for the function call. The listing shown below demonstrates how to use a function in VBScript.

```
'*****
' File:      Function.vbs (WSH sample in VBScript)
' Author:    (c) Günter Born
'
' Demonstrates how to use a function.
'*****
DIM i, j

j = 0

For i = 1 to 10    ' loop 10 times
    j = addx (i, j) ' Add values using function addx
Next

WScript.Echo "Result: ", j

WScript.Quit

Function addx (val1, val2)
    addx = val1 + val2
End Function
' End
```

Listing C.3.

Sample using a function

This sample uses the function *addx* with two parameters *val1* and *val2* to add these two values. The result is returned as a function value within the following statement:

```
addx = val1 + val2
```

addx is the function name already built in the function header. The result *val1 + val2* is assigned to this function name. The function block ends with the statement *End Function* that terminates the function and returns control back to the caller.

NOTE: VBScript supports also the optional *Exit Function* statement. If this statement is found within a function, the function terminates and the control is returned to the calling module.

A user-defined function may be used in the same way as a built-in VBScript function: Insert the function name and the requested parameters (in brackets) on the right side of an assignment statement. Within the listing shown above the function is called using the following statement:

```
j = addx (i, j)
```

User defined functions are the right tool to extend VBScript. Let's assume you need to calculate the sales price of several products that includes a value added tax (VAT). You may define a function *GetPrice* using the code shown below:

```
Function GetPrice (Net, VAT) ' Get the price including tax
    GetPrice = Net * (1.0 + VAT/100.0)
End Function
```

Listing C.4.

Function to add VAT to a net price

The VAT value may be passed (as a percentage value) to the function. Using this function within a script requires only the function name and the arguments.

```
Net = 10.0
Vat = 16.0
Price = GetPrice (Net, Vat)
Price1 = GetPrice (100.0, 16.0)
```

The statements above demonstrates that you may use variables and/or constants as function parameters.

NOTE: You may not define a function within another function or procedure. The code of a function must be declared on the script level. You should also note that local variable declared within the function body are only valid during the function call.

Passing arguments ByRef/ByVal

Passing arguments to a function may be done with »Call by Value« or with »Call by Reference«. Both variants are forced using the keywords *ByVal* or *ByRef*:

```
Public Function GetPrice (ByRef Net, ByVal Tax) ' calculate price
    GetPrice = Net * (1.0 + Vat/100.0)
End Function
```

If the interpreter detects the *ByVal* keyword, only the value of a parameter is passed to the function. If you uses the keyword *ByRef*, the address of the argument's value (the address of a constant or a variable) is passed to the function. Passing a parameter by reference allows the called function/procedure to change the value of a parameter (see also the following sections).

Built-in functions

VBScript provides a collection of built-in (intrinsic) functions for different purposes. You may use these functions in the same way as user defined functions. The statement:

```
i = Asc ("A")
```

assigns the ANSI-Code of the character *A* to the variable *i*. The conversion is done using the intrinsic function *Asc*, which need the character as a parameter.

Further information about functions and built-in functions may be obtained from the VBScript language reference (<http://msdn.microsoft.com/scripting>). Note

Procedures

Beside functions procedures are also in common use within VBScript applications. Procedures must be declared using the syntax shown below:

```
Sub Name (Parameter)
...
End Sub
```

A procedure delivers (in contrast to a function) no return value. To pass variable to a procedure, you may use different approaches:

- ◆ Declare a global variable at script level. The scope of a global variable is also valid within a procedure.
- ◆ You may pass variables as parameters to a procedure. This allows you to change these values within the procedure to return the results to the calling module.

The code sequence shown below demonstrates how to use a procedure to calculate a price including value added tax.

```
' *****  
' File:    Procedure.vbs (WSH sample in VBScript)  
' Author:  (c) Günter Born  
'  
' Demonstrates how to use a procedure  
' *****  
DIM Price, Net, Tax  
  
Tax = 16.0  
Net = 100.0  
  
GetPrice Net, Tax    ' calculate value  
WScript.Echo "Result: ", Price  
  
WScript.Quit  
  
Sub GetPrice (net1, tax1)  
    Price = net1 * (1.0 + (tax1/100.0))  
End Sub  
'* End
```

Listing C.5.

Using a procedure

The sample uses several techniques to exchange information with a procedure. Within the script level I have defined several global variables. The variable *Price* is global and may be accessed within the procedure. So we use this global variable to return the result to the calling program. The procedure itself requests two parameters during call. Note the syntax of the procedure call. You must specify the procedure name followed by the parameters. The parameters are separated with commas. An assignment as it is used in a function call is illegal.

NOTE: The *Call* keyword is not required during a procedure call. If you use the *Call* keyword however, you need to insert the parameters into a bracket. The statement *Call GetPrice (100.0, 0.16)* is equivalent to *GetPrice 100.0, 0.16*.

You may use also the keywords *Public* and *Private* within a procedure declaration (same as with functions) to set the scope of a procedure explicitly.

Remarks about parameter passing (ByRef, ByVal)

Within a function or procedure call parameters must be separated with commas. A parameter is a kind of local variable within the procedure. The interpreter may pass the value of a variable or the

address of a variable to the called procedure. If a variable's value, which is passed by address, is changed within a procedure, the value changes also on the level of the caller.

Within the sample shown above I have declared a variable *Price* on script level, to allow a procedure to return the result. This isn't a good programming style however, because such procedures can't be used without restrictions. The programmer must know the name of the global variable, which is changed within the procedure. If somebody mistype the variable name, the script won't work. Would it not be handier, if you may pass the values as parameters to the procedure and get also the results back? The modified code shown below demonstrates that the use of global variables to return a result from a procedure isn't required.

```
' *****
' File:    Procedure1.vbs (WSH sample in VBScript)
' Author:  (c) Günter Born
'
' Use a procedure.
' *****
DIM Price, Net, Tax

Tax = 16.0
Net = 100.0

Call GetPrice (Price, Net, Tax) ' calculate value
WScript.Echo "Result: ", Price

WScript.Quit

Sub GetPrice (pris, net, tax)
    pris = net * (1.0 + (tax/100.0))
End Sub
'* End
```

ListingC.6.

Modified procedure sample

The procedure call requires a third parameter *Price*. This parameter is declared a *pris* within the *GetPrice* procedure definition. If the value of *pris* is changed within the *GetPrice* procedure, this is reflected also to the variable *Price*. So your program can use the result calculated within the procedure without using a global variable to exchange values. Just for demonstration purposes I have also used the *Call* statement within this sample. But this hasn't any effect how the procedure works.

This kind of parameter passing is known as *Call by Reference* and will be used by default in VBScript. The interpreter passes the address of a variable to the procedure instead of passing just the value. So the procedure can use the address to change the value of a variable within the caller. The parameter passing by reference may be forced also using the *ByRef* keyword within the procedure declaration. You must set the keyword in front of the parameter.

In several cases it is not allowed that a procedure may change the values of the calling program. For instance: A program defined a variable *tax*, which is set once to 16,0. This value may be used in several procedure calls. It may be deadly, if a procedure changes this value accidentally, because the calling program still needs the original value for further processing. The parameters *net* and *tax* used in the sample above must not be returned from the procedure. To prevent that a modi-

fication of parameters within the procedure will influence the caller, you may declare in VBScript that a parameter must be passed as a value. This is shown in the listing below.

```
'*****
' File:    Procedure2.vbs (WSH sample in VBScript)
' Author:  (c) Günter Born
'
' Demonstrate using the Call-by-Value feature.
'*****
DIM Price, Net, VAT

VAT = 16.0
Net = 100.0

Call GetPrice (Price, Net, VAT) ' calculate value
WScript.Echo "Result: ", Price, " VAT: ", VAT
WScript.Quit

Sub GetPrice (ByRef pris, ByVal net, ByVal tax)
    pris = net * (1.0 + (tax/100.0))
    tax = 17.0
End Sub
'*End
```

Listing C.7.

Parameter passing with ByVal

The procedure declaration indicates for each parameter whether it shall be passed *ByVal* or *ByRef*. *Pris* is passed by reference (because we must return the result to the calling module), whilst *net* and *tax* are passed as values. If the parameters *net* or *tax* are changed within the procedure, it doesn't take effect on the values *Net* and *VAT* used in the script.

NOTE: As a last remark I should note that VBScript (up to version 5.0) doesn't supports the *On Error Goto* statement known from VB and VBA. You may use only the *On Error Resume Next* statement. A detailed VBScript language reference and tutorial may be downloaded from Microsoft's website msdn.microsoft.com/scripting.