ACADEMIA DE STUDII ECONOMICE - Bucureşti
Academy Of Economic Studies - Bucharest

FACULTY OF BUSINESS ADMINISTRATION
(Facultatea de Administrare a Afacerilor cu predare în limbi străine)

Internet Technologies for Business

-

VBScript

By: Professor Vasile AVRAM, PhD

- suport de curs destinat studenților de la sectia engleză -
(course notes for 1st year students of English division)
- anul I - Zi -

Bucureşti 2006

VBScript

CONTENTS

Chapter 6 VBScript

VBScript is a subset of VBA and it allows usage of most familiar functions of these one but there are some important differences:

- provides no support for classes as used in Visual Basic;
- control arrays not allowed;
- do not support data access using JET or RDO. The data access is allowed for back-end purposes by using Internet Service API (ISAPI) provided with Internet Information Server (IIS);
- provides no debugging features (you can use MsgBox function to show important information for you; when finishing the job these can be transformed in comments);
- do not support file operations found in Visual Basic;
- does not allow access to the Windows Help file system;
- does not support VB intrinsic constants (such as vbOKOnly, vbCancel etc);
- there is no concept of Multiple Document Interface (MDI) in VBScript;
- do not support the Screen, Printer, App, Debug and Clipboard objects. It support only the Err object.

Microsoft says that VBScript is integrated with World Wide Web browsers (more exactly with Internet Explorer versions of Microsoft) and designed to work with ActiveX controls and other objects embedded in active HTML documents.
The code modules are supported through the <SCRIPT></SCRIPT> tag. Each script section forms an independent code module that may have its own variables, functions and subroutines (they are similarly to the standard .bas modules found in Visual Basic).
The forms are created using the <FORM></FORM> tag and they are not visible as separate windows in the application. The forms are ways to group controls together for the purpose of addressing their properties and methods in code or to submit data to back-end process.

6.1 Using and placing VBScripts in a HTML page

In the following paragraphs are introduced some examples of using (and placing) VBScripts in a HTML page.

6.1.1 VBScript in the body of the HTML file

VBScript in the body of the HTML page will be executed when the page loads. Generally, the scripts in the body, will generate the content of the page.
Example:

```
<html>
 <head>
   <title>
        Page containing VBScript
   </title>
  </head>
 <body>
   <script type="text/vbscript">
        document.write("This text is displayed by a VBScript!")
   </script>
  </body>
</html>
```

Comments:

The tag <script> have the attribute „type”, that specifies the script type (VBScript in our case). This script composed by a single command that displays inside the page the text: "This text is displayed by a VBScript!".

If the script is included in a comment tag (<!--) then the browsers that do not „know” (interpret) VBScript will not display in the page the script text (script source), as shown in the folowing sample:

```
<!--
document.write("<i>"+"This text is displayed by a VBScript!"+"<\/i>")
//-->
```

The browsers that know VBScript will process the script, even this included in a comment line.

The string „//”, comment in VBScript, tell to the browser do not process the line „-->”. We can not use the sintax „//<!--”, because a browser that do not interpret VBScript will display that string „//”.

6.1.2 VBScript in heading

If we want be shure that the script executes before displaying any element in the page we can include this in the heading part of the HTML page (file). The VBScript in the head section will be executed when called, or when an event is triggered.

Example:

```
<html>
 <head>
   <title>
        Page with VBScript
   </title>
  <script type="text/vbscript">
       document.write("This text is displayed by a VBScript!")
  </script>
  </head>
 <body>
   <P>  This text must appear in the page after the execution of the VBScript.
 </body>
</html>
```

The number of scripts placed in the head section and in the body of a HTML page is unlimited.

6.2 Variables

Variables: are named storage locations that can contain data that can be modified during script is running. The variables are the memory cells used for storing forms input data and/or its computational results. The only datatype accepted is Variant which can contain any kind of data. Table 6.1 shows the data subtypes accepted. The declaration of variables is realized by using the Dim or ReDim statement.

The naming of variables in VBScript uses the rules:

1. An identifier must begin with a letter;
2. Can’t be longer than 255 characters;

3. Can't contain embedded period or embedded type declaration character;
4. Must be unique in same scope (the range from which the variable can be referenced).

The number of variables per procedure is limited to 127 (an array counts as one variable) and each script is limited do not have more than 127 module-level variables. The length of the time a variable exists is its lifetime. A script level variable's lifetime begins when its declaration statement is encountered as procedure begins, and ends when the procedure concludes.

The array elements can have how many dimensions required, each dimension defined by separating from previous with a comma, as in this example Dim MatrixAlpha(10,20,30) that defines a three dimensional array. The addresses of elements (the index) start from 0 for every dimension. In the previous example the number of elements of the array called MatrixAlpha is 11x21x31.

The user can define his constants by following the procedure:

1) define the memory variable for constant;
2) assign the value, as literal value, to the variable;
3) use the name of the variable anywhere (in his scope) is required the literal value assigned to it.

Examples:

```
Dim Pi,CompanyName,EndDate
rem Assigning values
Pi=3.14
CompanyName="Media Advertising, Inc."
EndDate=#12-31-2007#
```

Table 6.1. Possible Data Subtypes for a Variant

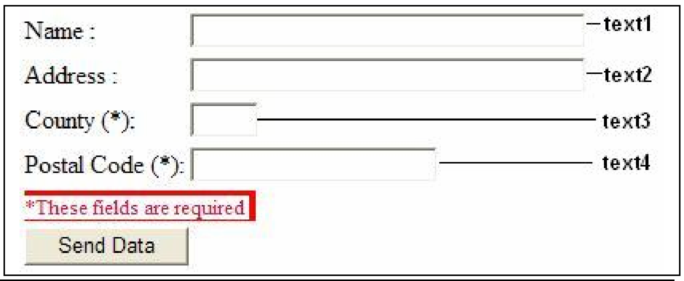
Subtype	**Meaning**
Empty	Variant is un-initialized. Value is either 0 for numeric variables or a zero-length string ("") for string variables.
Null	Variant intentionally contains no valid data.
Boolean	Contains either True or False.
Byte	Contains an integer in the range 0 to 255.
Integer	Contains an integer in the range -32,768 to 32,767.
Single	Contains an integer in the range -2,147,483,648 to 2,147,483,647.
Long	Contains a single-precision, floating-point number in the range -3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values.
Double	Contains a double-precision, floating-point number in the range -1.79769313486232E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values.
String	Contains a variable-length string that can be up to approximately 2 billion characters in length.
Date (Time)	Contains a number that represents a date between January 1, 100 to December 31, 9999.

Object	Contains an object.
Error	Contains an error number.

Example:

```
<SCRIPT TYPE="text/vbscript">
<!--
rem Defines two variant variables
Dim DegreesFahrenheit, DegreesCelsius
Rem Defines an array A of 100 elements and an array B of an unspecified dimension (used for
dynamic resizing)
Dim A(100), B()
Rem Dynamically resizes the array B
ReDim B(50)
-->
</SCRIPT>
```

Example:
This example shows how you can reference and use variables defined in forms (as text box objects) and assign a value. The procedure cleans the text boxes in the form when the web page containing the form is loaded.



```
<HTML>
<BODY bgColor=white>
<TABLE>
<TR>
<TD valign=top>            </TD>
<TD valign=top>
<TABLE> <tr>  <td colspan=2></td> </tr>
<TR>  <TD>Name :</TD><TD><INPUT id=text1 name=text1
            style="WIDTH: 248px; HEIGHT: 22px" size=32></TD>
</TR>
<TR>
  <TD>Address :</TD><TD><INPUT id=text2 name=text2
            style="WIDTH: 248px; HEIGHT: 22px" size=32></TD>
</TR>
<TR>
  <TD>County&nbsp;(*):</TD><TD><INPUT id=text3 name=text3
                  style="WIDTH: 43px; HEIGHT: 22px"></TD>
</TR>
<TR>
  <TD>Postal&nbsp;Code (*):</TD><TD><INPUT id=text4 name=text4 width =
                  "10"></TD>
</TR>
<TR>
  <TD colSpan=2><FONT
            style="BACKGROUND-COLOR: red"><FONT
            style="BACKGROUND-COLOR: #ffffff" color=crimson size=2><FONT
            style="BACKGROUND-COLOR: white">*These fields are
            required</FONT>&nbsp;</FONT> </FONT>        </TD>
</TR>
<TR>
  <TD colSpan=2><INPUT id=button1 name=button1 type=button value=" Send Data "
      LANGUAGE=javascript onclick="return button1_onclick()"></TD>
```

```
</TR>
</TABLE>
</TD>
</TR>
</TABLE>
</BODY>
<SCRIPT LANGUAGE=VBScript>
rem When the page is loaded, clear the text boxes
sub Window_onLoad
        rem Reset edit boxes
        text1.value = ""
        text2.value = ""
        text3.value = ""
        text4.value = ""
end sub

</SCRIPT>
<SCRIPT LANGUAGE=JScript>
</SCRIPT>

</HTML>
```

6.3 Assignments and expressions

Assignments

The general syntax for assignment is:

variable = expression

The interpretation of this is: the *variable* before the assignment operator is assigned a value of the *expression* after it, and in the process, the previous value of *variable* is destroyed.

An assignment statement stores a literal value or a computational result in a variable and is used to perform most arithmetic operation in a program.

Expressions

An *expression* can be an expression on character string, a logical expression or arithmetic expression. It can be a variable, a constant, a literal, or a combination of these connected by appropriate operators (table #.2).

1. An *expression on character string* can be built (in VB but not only) using:
- the concatenation operator: &
- intrinsic functions for extracting substrings from a string variable or string constant such as:
  - **Right**(*string,number_of_characters*) - extracting substring from the end
  - **Left**(*string,number_of_characters*) - extracting substring from the beginning
- functions that manipulate strings:
  - **Cstr**(*expression*) – convert the expression in a character string;
  - **Lcase**(*string_expression*) – convert the string in lower case;
  - **Ltrim**(*string*), Rtrim(*string*), Trim(*string*) – eliminates the spaces (trailing) from left (leading blanks), right and, respectively left-right;
  - **Str**(*number*) – converts number in string;

Ucase(*string*) – converts string to uppercase.

2. A *logical expression* can be:
• **simple**, with the general syntax:
<*variable*>[<*relation_operator*><*variable*>]
or
<*variable*>[<*relation_operator*><*constant*>]
<*relation_operator*>::=<|<=|>|>=|=|<>
• **complex**, with the general syntax:
e_1 **Eqv** e_2 - equivalence;
e_1 **Imp** e_2 - logical implication;
e_1 **Xor** e_2 - exclusive or;
<*logical_expression*$_1$><*logical_operator*><*logical_expression*$_2$>
where the *logical_operator* can be:
And, Or as binary operators (connectives); Not as unary operator.
The precedence of evaluation of logical operators is Not, And, Or, Xor, Eqv, Imp (table #.2).

The logical functions works as explained in chapter two in the book "**Informatics: Computer Hardware and Programming in Visual Basic**" [AvDg03]. Each one has an associated truth table that take carry of the two states True or False and, in some programming environments, a state called Empty (or Null) to distinguish between False an non value.

3. An *arithmetic expression* uses the syntax:
<*operand*$_1$><*arithmetic_operator*><*operand*$_2$>
where:
- <*operand*$_1$>,<*operand*$_2$> can be numeric variables, constants or arithmetic expressions (or calls to functions that returns numeric values)
- *arithmetic_operator* (binary operators) is one of those shown in table 6.2, column Arithmetic.

Table 6.2. VBScript Operators

Arithmetic		**Comparison**		**Logical**	
Description	**Symbol**	**Description**	**Symbol**	**Description**	**Symbol**
Exponentiation	**^**	Equality	**=**	Negation	**Not**
Unary negation	**-**	Inequality	**<>**	Conjunction	**And**
Multiplication	*****	Less than	**<**	Disjunction	**Or**
Division	**/**	Greater than	**>**	Exclusion	**Xor**
Integer division	\	Less than or equal to	**<=**	Equivalence	**Eqv**
Modulo arithmetic	**Mod**	Greater than or equal to	**>=**	Implication	**Imp**
Addition	**+**	Object equivalence	**Is**		
Subtraction	**-**				
String concatenation	**&**				

Constants: can appear as such anywhere as literals, intrinsic constants available in VBscript, or as declarations in the declarative part of the script.

Examples:

Constant	Type
"Welcome to the information century !"	string
$25,000.00	currency
3.14	positive real number
-123	negative integer number
0.123e+3	number written in the scientific notation
“11/12/2002”	date

- can be defined as a declaration by using the syntax:

Const *constantName=expression*[,…]

where:

- *constantName* is an user identifier defined by following the naming rules;
- *expression* an expression evaluated to an agreed data type whose evaluation is considered the default value for the constant.

Examples:

```
Const Pi = 3.14159
Const Vat = 0.19, Star = “★”
Const CutOffDate=#06-30-2007#
```

A declared constant is a named storage location that contains data that can not be modified during the script execution. The most used constants are number constants and string constants. A string constant is sequences from 0 to 1024 characters enclosed in quotes.

6.4 Procedures and functions

The procedures are small logical components in which you can break (split) a program for a specific task. They are very useful for condensing repeated or shared tasks (such as calculations frequently used). The procedures are called to do their job from other procedures. Generally a procedure can take arguments, perform a series of statements, and change the value of its arguments.

The major benefits of programming with procedures are:

- procedures allow you to break your programs into discrete logical units, each of each you can debug more easily than an entire program without procedures;
- procedures used in one program can act as building blocks for other programs, usually with little or no modification.

The general form of a VBScript procedure/function can be described as follows:

```
Procedure_type Procedure_name ([Argument_list])      } the procedure heading
     [declaration_statements]   } procedure body
     [executive_statements]     }
End Procedure_type
```

The *Procedure_type* defines if function or procedure:

Procedure_type::=**Sub** | **Function**

The user identifier *Procedure_name* is declared to be the name of a procedure or function.

The *argument_list* declares the values that are passed in from a calling procedure.

A procedure can have two parts:

– a **declaration** part that contains reservations of memory cells that are needed to hold data and program results and what kind of information will be stored in each memory cell.
– an **executive** part that contains statement (derived from the algorithm you want to communicate to the computer) that are translated into machine language and later on executed.

Any identifier declared in the declaration part is by default usable only during the execution of the procedure and can be referenced only within the procedure.
The procedure *executive_statements* describes the data manipulation performed when the procedure is activated through a procedure call statement. The procedure call statement initiates the execution of a procedure.

After *procedure_name* has finished executing, the program statement that follows the procedure call will be executed. The information passed between a procedure and the program unit calls it are called procedure parameters.
The values passed into a procedure by the calling program are called procedure inputs. The results returned to the calling program are called procedure outputs.

There are two types of procedures used in VBScript:
1) **Sub** procedures do not return a value;
2) **Function** procedures return a value; you return a value by assigning it to the procedure name itself: *Function_name=expression* for the return.

A call to a Sub procedure is a stand_alone statement. A Sub procedure can be invoked by a Call statement:

Call *Procedure_name* (*argument_list*) – if specified the argument list must be enclosed in parenthesis;
or
Procedure_name argument_list

A function procedure can return a value to the caller. A call to a function procedure can be realized using the syntax:
Variable_name=Function_name(*arguments*) or **Call** *Function_name(arguments)* or *Function_name arguments*

There are two differences between Sub and Function procedures:
– generally, you call a function by including the function procedure name and arguments on the right side of a larger statement or expression (*Variablename=Functionname*());
– the result value is returned to the caller by intermediate of an assignment statement to the name of the function.

When we call functions or procedures without arguments we must include an empty set of parenthesis () after the procedure/function name if the function is in the right part of an assignment or in an expression; if the call is a stand-alone sentence the parenthesis not required (and not allowed!).

6.5 Decisional (conditional/alternative) statements

The decision structure is used for choosing an alternative (an operation or block of operations) from two possible alternatives. Algorithm steps that select from a choice of actions are called decision steps.

If … Then … Else. The first syntax for alternative structure can be expressed as:

```
If condition Then
    operation_1
  Else
    operation_2
End If
```

The decision block can be expressed in a natural language as:

- evaluate the expression that defines the logical condition <*condition*>;
- If the result of evaluation is True
  Then execute *operation*$_1$
  Else execute *operation*$_2$;
- continue the execution with the next step in the flow.

If the condition is True Then the group between Then and Else will be executed Else the group of sentences between Else and End If will be executed.

The logical condition <*condition*> is a logical expression that will be evaluated either to True or either to False. The logical conditions can be simple or complex logical conditions.

A simple logical condition has the general syntax:

<*variable*> [<*relation_operator* ><*variable*>]

or

<*variable*> [<*relation_operator* ><*constant*>]

The *relation_operator* can be one of:

Relation Operator	Interpretation
<	**Less than. Example:** delta < 0
<=	**Less than or equal. Example:** delta <= 0
>	**Greater than. Example:** delta > 0
>=	**Greater than or equal. Example:** delta >= 0
=	**Equal to. Example:** a = 0
<>	**Not equal. Example:** a<>0

If <*variable*> is number or Boolean then is possible to directly compare with 0, respectively True and is not necessary to write the equal relation operator.

The simple logical conditions will be connected by the **AND**, **OR, and NOT** logical operators to form complex conditions. The logical operators are evaluated in the order NOT, AND, and OR. The change of the natural order of evaluation can be done by using parenthesis in the same way for arithmetic expressions. The precedence of operator evaluation in Boolean expressions (logical expressions) is:

Not
^,*, /, div, mod, and
+, -, or
<, <=, =, <>, >=, >

If … Then. It is possible do not have a specific operation on the two branches, that means situations as expressed in one of the syntaxes:

a) conditionally executing only one statement:

```
If condition Then statement
```

If the condition is True then the statement is executed

b) conditionally executing a set of sentences:

```
If condition Then
    Sequence of statements₁
End If
```

If the condition is true then the set of sentences placed between If and End If are executed.

VBScript allows nesting *if ... then ... else* sentences (one if statement inside another) to form complex decision structures (decisions with multiple alternatives).

If … Then … Elseif. The nested If can be coded as a multiple-alternative decision. The syntax for that nested If is:

```
If condition₁
    Then
        sequence₁
    ElseIf condition₂ Then

        sequence₂
    ⋮
    Else
    ⋮
End If
```

For the first time *condition*$_1$ is tested. If the result is False *condition*$_2$ and so on until a True evaluated condition reached for each the associated sentence block executed. After executing the reached block the control of processing is passed to the next sentence after End if. If no condition evaluates to True then the sentence block associated to the Else branch executes (if Else defined; if not nothing executes).

Case of. Executes one of several groups of statements depending on the value of an expression (called selector). The case structure (and statement) can is especially used when selection is based on the value of a single variable or a simple expression (called the case selector).

```
Select Case test_ expression
  [Case expression_list₁
    [sentences₁]]
  [Case expression_list ₂
    [sentences ₂]]
    ⋮
  [Case Else
    [sentences ₙ]]
End Select
```

- each *expression_list*$_i$ is represented (or formed) by one or many comma separated values (value list);
- in the block **Select Case** the case **Else** can appear only once and only as a last case;
- if many cases fit to *test_ expression* then the first founded will be executed;
- each sentence block (*sentences*$_i$) can include zero, one or many sentences;

- the evaluation of the test expression is realized only once at the beginning of the Select Case structure.

6.6 Repeating Structure

The repeating structure repeats a block of statements while a condition is True or Until a condition becomes True. The repetition of steps in a program is called a **loop**. The executions of such blocks follow the scenario (while): the condition is evaluated and if the condition evaluates to:

• **True** then executes the block of statements;
• **False** then end the execution of the cycle (Loop) and continue the execution of the program.

If for the first time the condition is False the sentence block is simply skipped.

Conditional Loop With Condition Evaluated First

Syntax:

$$\begin{array}{ll}
\textbf{Do } [\{\textbf{While}|\textbf{Until}\}\textit{condition}] & \rightarrow \text{beginning of cycle} \\
\quad \left.\begin{array}{l}[\textit{statements}] \\ [\textbf{Exit Do}] \\ [\textit{statements}]\end{array}\right\} & \rightarrow \text{body of the cycle} \\
\textbf{Loop} & \rightarrow \text{the end of sentence block}
\end{array}$$

The commands Loop and Exit Do are used to do:

- **Loop** – an unconditional jump (or branch) to the beginning of the associated cycle (the evaluation of the condition);
- **Exit Do** – an unconditional ending of the cycle (a jump to the next sentence defined under the loop that ends the body of the cycle).

Do…Until work as:

1) execute statements;
2) evaluate the condition Loop or Exit

The block of commands between Do… and Loop will be executed while/until the conditional expression “condition” evaluates to True.

Example:
In this example we suppose that the user types numbers containing digits between 0 (zero) and 9(nine) as integer values. Because the function InputBox() reads text values the mistakes (any other characters than digits and/or spaces between digits) will produces a computation error message. To avoid that is necessary to test whether the typed value is number or not.

```
<html>
<head>
<title>A function definition</title>
<script language=vbscript>
<!--
' Read_Number reads a value from keyboard and verifies
' if number or not (a process called validation). If the value is not a number
' signals that to the user who can choose between cancel or resume the operation from typing
Function Read_Number(xNr, denNr)
  Dim Answer
  Do While True = True ' an infinite cycle
    xNr = InputBox("Type the value for " & denNr & ":", "Example")
    If (IsNumeric(Trim(xNr)))= False Then
```

```
        Answer = MsgBox("The Value for " & denNr & " must be Numerical !")
        If Answer = 2 Then ' Cancel Button pressed
            Read_Number = "*Cancel" ' The returned value to the caller is *Cancel
            Exit Do ' Exit from the infinite cycle and return to the caller
        End If
      Else
        Read_Number = Trim(xNr) ' The returned value will be the number
                                ' without extra spaces (to the left or right)
        Exit Do ' Exit from the infinite cycle and return to the caller
     End If
 Loop ' Restart the cycle
End Function
-->
</script>
</head>
<body>
<script language=vbscript>
<!--
dim numarcitit
document.write("Varsta:" & read_number(numarcitit,"Varsta "))
-->
</script>
</body>
</html>
```

Conditional Loop with Condition Evaluated After

In this case the operation is executed first and then the condition is evaluated:

Do
    operations
Loop {While|Until} *condition*

It can be described as:
- the *operations* are executed;
- the *condition* is evaluated;
- **if** the result of the evaluation of the *condition* is False *then loop* to execute again the *operations*;
- **if** the evaluation of the *condition* is True *then* continue the execution of the program (and close the loop).

Counted Loop.

The statement executes a set of statements (operation1) within a loop a specified number of times. A variable is used as counter to specify how many times the statements inside the loop are executed.

Syntax:

For *counter* = i_v **To** f_v [**Step** *s*]
    operations
    [Exit For]
Next [*counter*]

Where:
- i_v – is the initial value (start value – usually 0 or 1);
- f_v – is the end value (the expected value – usually how many times);
- *step* – is the increasing (or decreasing) step for counter; if a value for Step not specified the default is used (+1);

- the value for *s* can be positive or negative:
  - if *s* is positive then must have the inequality $i_v<f_v$ (otherwise the cycle never executes);
  - if *s* is negative then must have the inequality $i_v>=f_v$ (otherwise the cycle never executes);
- the cycle can be stopped unconditionally by intermediate of the sentence **Exit For**

The execution of For (VB) sentence follows the scenario:

1. The value i_v is assigned to the variable *counter*;
2. The value of variable *counter* is compared with the end value f_v (If the value for step is negative is checked if *counter*$<f_v$);
3. The operations are executed;
4. The value of variable *counter* is incremented with the value step (1 if step not specified);
5. Repeat the steps from 2 to 5.

The interpretation of the elements of For…Next sentence is:

what cycle number is ↓ how many times ↓

For *counter*=*start_value* To *end_value* [Step *increment_decrement*]
[*statements*]
[Exit For] ← stop the cycle
[*statements*]
Next [*counter*]

Example:

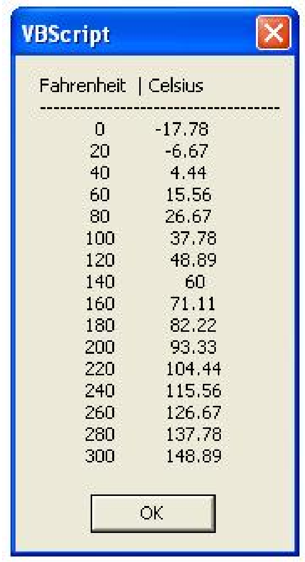
We want to list a Fahrenheit to Celsius correspondence table based on the computation formula:

$$\text{CELSIUS}^o = (5/9)*(\text{FAHRENHEIT}^o - 32)$$

The correspondence table will be displayed (figure #.1) starting with the minimal value (**min**) 0 (zero) and ending with the maximal value (**max**) of 300 degrees and the computation and display will be done from 20 to 20 degrees (**pas**). We use, to solve this problem, assignments instructions, the function MsgBox to display the result and an instruction For that allow us to repeat the execution of a group of sentences until a specified condition satisfied.

The script looks as:

```
<html>
<head>
<title>Fahrenheit-Celsius</title>
<script language=vbscript>
<!--
 Sub Coresp_Temp()
    Dim min, max, pas, fahrenheit, celsius, tabel
    ' Computation of the correspondence Co- Fo
    min = 0 ' Starting Value
    max = 300 ' Ending Value
    pas = 20 ' From 20 to 20 degrees
    fahrenheit=0
    celsius=0
```



```
VBScript

Fahrenheit | Celsius
--------------------------------
    0     -17.78
   20      -6.67
   40       4.44
   60      15.56
   80      26.67
  100      37.78
  120      48.89
  140       60
  160      71.11
  180      82.22
  200      93.33
  220     104.44
  240     115.56
  260     126.67
  280     137.78
  300     148.89

        OK
```

Figure #.1 The output as a message box

```
      tabel=""
      tabel = "Fahrenheit  | Celsius  " & Chr(13) & Chr(10) &_
              string(36, "-") & Chr(13) & Chr(10)
      For fahrenheit = min To max Step pas
         celsius = (5/9)*(fahrenheit-32)
         tabel = tabel & Right(Space(12) & round(fahrenheit,2),12) & "    " _
                       & Right(Space(12) & round(celsius,2),12) & Chr(13) & Chr(10)
      Next
      MsgBox(tabel)
  End Sub
-->
</script>
</head>
<body>
<script language=vbscript>
<!--
 Call Coresp_Temp
-->
</script>
</body>
</html>
```

In the next version the script will display the output to a page that will be displayed by the browser.

```
<html>
<head>
<title>Fahrenheit-Celsius</title>
<script language=vbscript>
<!--
  Sub Coresp_Temp()
      Dim min, max, pas, fahrenheit, celsius
      ' Computation of the correspondence Co- Fo
      min = 0 ' Starting Value
      max = 300 ' Ending Value
      pas = 20 ' From 20 to 20 degrees
      fahrenheit=0
      celsius=0
tabs="&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&n
bsp;&nbsp;"
      document.write("Fahrenheit  | Celsius  " & "<br />")
      document.write(string(36, "-") &  "<br />")
     For fahrenheit = min To max Step pas
         celsius = (5/9)*(fahrenheit-32)
         document.write(Right(Space(12) & round(fahrenheit,2),12) & tabs _
                       & Right(Space(12) & round(celsius,2),12) &  "<br />")
     Next
  End Sub
-->
</script>
</head>
<body>
<script language=vbscript>
<!--
 Call Coresp_Temp
-->
</script>
</body>
</html>
```

For Each ... Next**.** This sentence allows to apply a set of sentences to an object collection or to a multitude (arrays, vectors, multidimensional massive) without specifying the number of cycles (that specification is difficult if the dynamic memory reservation used).

The syntax of that sentence is:

For Each *element* **In** *group*
Sentences
Next *element*

Example:

```
<head>
<title>Using For Each sentence</title>
</head>
<html>
<body>
<script language=vbscript>
<!—
dim divisions(2), i, indent
divisions(0)="English"
divisions(1)="French"
divisions(2)="German"
i=1
indent="&nbsp;&nbsp;&nbsp;&nbsp;&nbsp"
document.write("The faculty sections are:<br />")
For Each x in divisions
  document.write( indent & indent & i & ") <b>" & x & "</b><br />")
  i=i+1
Next
-->
</script>
</body>
</html>
```

the code placed in the body of the HTML page will produces the output:

The faculty sections are:
1) **English**
2) **French**
3) **German**

Annex 1. List of VBScript intrinsic functions

Date/Time Functions

Function	Description
CDate	Converts a valid date and time expression to the variant of subtype Date
Date	Returns the current system date
DateAdd	Returns a date to which a specified time interval has been added
DateDiff	Returns the number of intervals between two dates
DatePart	Returns the specified part of a given date
DateSerial	Returns the date for a specified year, month, and day
DateValue	Returns a date
Day	Returns a number that represents the day of the month (between 1 and 31, inclusive)
FormatDateTime	Returns an expression formatted as a date or time
Hour	Returns a number that represents the hour of the day (between 0 and 23, inclusive)
IsDate	Returns a Boolean value that indicates if the evaluated expression can be converted to a date
Minute	Returns a number that represents the minute of the hour (between 0 and 59, inclusive)
Month	Returns a number that represents the month of the year (between 1 and 12, inclusive)
MonthName	Returns the name of a specified month
Now	Returns the current system date and time
Second	Returns a number that represents the second of the minute (between 0 and 59, inclusive)
Time	Returns the current system time
Timer	Returns the number of seconds since 12:00 AM
TimeSerial	Returns the time for a specific hour, minute, and second
TimeValue	Returns a time
Weekday	Returns a number that represents the day of the week (between 1 and 7, inclusive)
WeekdayName	Returns the weekday name of a specified day of the week
Year	Returns a number that represents the year

Conversion Functions

Function	Description
Asc	Converts the first letter in a string to ANSI code
CBool	Converts an expression to a variant of subtype Boolean
CByte	Converts an expression to a variant of subtype Byte
CCur	Converts an expression to a variant of subtype Currency
CDate	Converts a valid date and time expression to the variant of subtype Date
CDbl	Converts an expression to a variant of subtype Double
Chr	Converts the specified ANSI code to a character
CInt	Converts an expression to a variant of subtype Integer
CLng	Converts an expression to a variant of subtype Long
CSng	Converts an expression to a variant of subtype Single
CStr	Converts an expression to a variant of subtype String
Hex	Returns the hexadecimal value of a specified number
Oct	Returns the octal value of a specified number

Format Functions

Function	Description
FormatCurrency	Returns an expression formatted as a currency value
FormatDateTime	Returns an expression formatted as a date or time
FormatNumber	Returns an expression formatted as a number
FormatPercent	Returns an expression formatted as a percentage

Math Functions

Function	Description
Abs	Returns the absolute value of a specified number
Atn	Returns the arctangent of a specified number
Cos	Returns the cosine of a specified number (angle)
Exp	Returns e raised to a power
Hex	Returns the hexadecimal value of a specified number
Int	Returns the integer part of a specified number
Fix	Returns the integer part of a specified number
Log	Returns the natural logarithm of a specified number
Oct	Returns the octal value of a specified number
Rnd	Returns a random number less than 1 but greater or equal to 0
Sgn	Returns an integer that indicates the sign of a specified number
Sin	Returns the sine of a specified number (angle)
Sqr	Returns the square root of a specified number
Tan	Returns the tangent of a specified number (angle)

Array Functions

Function	Description
Array	Returns a variant containing an array
Filter	Returns a zero-based array that contains a subset of a string array based on a filter criteria
IsArray	Returns a Boolean value that indicates whether a specified variable is an array
Join	Returns a string that consists of a number of substrings in an array
LBound	Returns the smallest subscript for the indicated dimension of an array
Split	Returns a zero-based, one-dimensional array that contains a specified number of substrings
UBound	Returns the largest subscript for the indicated dimension of an array

String Functions

Function	Description
InStr	Returns the position of the first occurrence of one string within another. The search begins at the first character of the string
InStrRev	Returns the position of the first occurrence of one string within another. The search begins at the last character of the string
LCase	Converts a specified string to lowercase
Left	Returns a specified number of characters from the left side of a string
Len	Returns the number of characters in a string
LTrim	Removes spaces on the left side of a string
RTrim	Removes spaces on the right side of a string
Trim	Removes spaces on both the left and the right side of a string
Mid	Returns a specified number of characters from a string
Replace	Replaces a specified part of a string with another string a specified number of times
Right	Returns a specified number of characters from the right side of a string
Space	Returns a string that consists of a specified number of spaces

StrComp	Compares two strings and returns a value that represents the result of the comparison
String	Returns a string that contains a repeating character of a specified length
StrReverse	Reverses a string
UCase	Converts a specified string to uppercase

Other Functions

Function	**Description**
CreateObject	Creates an object of a specified type
Eval	Evaluates an expression and returns the result
GetLocale	Returns the current locale ID
GetObject	Returns a reference to an automation object from a file
GetRef	Allows you to connect a VBScript procedure to a DHTML event on your pages
InputBox	Displays a dialog box, where the user can write some input and/or click on a button, and returns the contents
IsEmpty	Returns a Boolean value that indicates whether a specified variable has been initialized or not
IsNull	Returns a Boolean value that indicates whether a specified expression contains no valid data (Null)
IsNumeric	Returns a Boolean value that indicates whether a specified expression can be evaluated as a number
IsObject	Returns a Boolean value that indicates whether the specified expression is an automation object
LoadPicture	Returns a picture object. Available only on 32-bit platforms
MsgBox	Displays a message box, waits for the user to click a button, and returns a value that indicates which button the user clicked
RGB	Returns a number that represents an RGB color value
Round	Rounds a number
ScriptEngine	Returns the scripting language in use
ScriptEngineBuildVersion	Returns the build version number of the scripting engine in use
ScriptEngineMajorVersion	Returns the major version number of the scripting engine in use
ScriptEngineMinorVersion	Returns the minor version number of the scripting engine in use
SetLocale	Sets the locale ID and returns the previous locale ID
TypeName	Returns the subtype of a specified variable
VarType	Returns a value that indicates the subtype of a specified variable

References

1.	**[AvDg03]**	**Vasile Avram, Gheorghe Dodescu**	**Informatics: Computer Hardware and Programming in Visual Basic, Ed. Economică, Bucureşti, 2003 (Chp. 1.6, 1.7, 1.8, 7.11.3 and 7.11.4)**
2.	**[DgAv05]**	**Gheorghe Dodescu, Vasile Avram**	**Informatics: Operating Systems and Application Software, Ed. Economică, Bucureşti, 2005 (Chp. 10.1, 10.2 and 10.3)**
3.	**[BIS-TDM]**	**Dave Chaffey, Paul Bocij, Andrew Greasley, Simon Hickie**	**Business Information Systems-Technology, Development and Management for the e-business, Prentice Hall, London, second edition, 2003**
4.	**[BF01]**	**Benjamin Faraggi**	**Architectures marcandes et portails B to B, Ed. DUNOD, Paris, 2001**
5.	[RFC 1630]	T. Berners-Lee	RFC 1630 - Universal Resource Identifiers in WWW, Network Working Group, CERN, June 1994
6.	[RFC3986]	T. Berners-Lee W3C/MIT, R. Fielding Day Software, L. Masinter Adobe Systems	Uniform Resource Identifier (URI): Generic Syntax, January 2005
7.	**[KLJL]**	Kenneth C. Laudon, Jane P. Laudon	Essentials of Management Information Systems – Managing the Digital Firm, Prentice Hall, fifth edition, 2003
8.	**[W3C]**	www.w3c.org	World Wide Web Consortium, Web standards collection
9.	**[MNSS]**	Todd Miller, Matthew L. Nelson, Stella Ying Shen and Michael J. Shaw	e-Business Management Models: A Services Perspective and Case Studies, Revere Group
10.	**[MSDN]**	Microsoft Press	Microsoft Developper Network
10.	E-commerce business models http://www.iusmentis.com http://www.iusmentis.com/business/ecommerce/businessmodels/		
11.	http://digitalenterprise.org/models/models.html Professor Michael Rappa, North Carolina State University		