

**Prepared by** : Hitendra Panchal  
**Subject** : VBScript  
**Dated** : November 08, 2008

---

## Adding VBScript Code to an HTML Page

```
<html>
<head>

</head>

<body>

<script type="text/vbscript">

document.write("Hello from VBScript!")

</script>

</body>

</html>
```

You can use the SCRIPT element to add VBScript code to an HTML page.

The <SCRIPT> Tag

VBScript code is written within paired <SCRIPT> tags. For example, a procedure to test a delivery date might appear as follows:

```
<SCRIPT LANGUAGE="VBScript">
<!--
    Function CanDeliver(Dt)
        CanDeliver = (CDate(Dt) - Now()) > 2
    End Function
-->
</SCRIPT>
```

Beginning and ending <SCRIPT> tags surround the code. The LANGUAGE attribute indicates the scripting language. You must specify the language because browsers can use other scripting languages. Notice that the `CanDeliver` function is embedded in comment tags (<!-- and -->). This prevents browsers that don't understand the <SCRIPT> tag from displaying the code.

Since the example is a general function — it is not tied to any particular form control — you can include it in the HEAD section of the page:

```

<HTML>
<HEAD>
<TITLE>Place Your Order</TITLE>
<SCRIPT LANGUAGE="VBScript">
<!--
    Function CanDeliver(Dt)
        CanDeliver = (CDate(Dt) - Now()) > 2
    End Function
-->
</SCRIPT>
</HEAD>
<BODY>
...

```

You can use SCRIPT blocks anywhere in an HTML page. You can put them in both the BODY and HEAD sections. However, you will probably want to put all general-purpose scripting code in the HEAD section in order to keep all the code together. Keeping your code in the HEAD section ensures that all code is read and decoded before it is called from within the BODY section.

One notable exception to this rule is that you may want to provide inline scripting code within forms to respond to the events of objects in your form. For example, you can embed scripting code to respond to a button click in a form:

```

<HTML>
<HEAD>
<TITLE>Test Button Events</TITLE>
</HEAD>
<BODY>
<FORM NAME="Form1">
    <INPUT TYPE="Button" NAME="Button1" VALUE="Click">
    <SCRIPT FOR="Button1" EVENT="onClick" LANGUAGE="VBScript">
        MsgBox "Button Pressed!"
    </SCRIPT>
</FORM>
</BODY>
</HTML>

```

Most of your code will appear in either Sub or Function procedures and will be called only when specified by your code. However, you can write VBScript code outside procedures, but still within a SCRIPT block. This code is executed only once, when the HTML page loads. This allows you to initialize data or dynamically change the look of your Web page when it loads.

# VBScript Conditional Statements

## Conditional Statements

Very often when you write code, you want to perform different actions for different decisions. You can use conditional statements in your code to do this.

In VBScript we have four conditional statements:

- **if statement** - use this statement if you want to execute a set of code when a condition is true
  - **if...then...else statement** - use this statement if you want to select one of two sets of lines to execute
  - **if...then...elseif statement** - use this statement if you want to select one of many sets of lines to execute
  - **select case statement** - use this statement if you want to select one of many sets of lines to execute
- 

## If...Then.....Else

You should use the If...Then...Else statement if you want to

- execute some code if a condition is true
- select one of two blocks of code to execute

If you want to execute only **one** statement when a condition is true, you can write the code on one line:

```
if i=10 Then msgbox "Hello"
```

There is no `..else..` in this syntax. You just tell the code to perform **one action** if the condition is true (in this case if `i=10`).

If you want to execute **more than one** statement when a condition is true, you must put each statement on separate lines and end the statement with the keyword "End If":

```
if i=10 Then  
  
    msgbox "Hello"  
  
    i = i+1  
  
end If
```

There is no `..else..` in this syntax either. You just tell the code to perform **multiple actions** if the condition is true.

If you want to execute a statement if a condition is true and execute another statement if the condition is not true, you must add the "Else" keyword:

```
if i=10 then  
  
    msgbox "Hello"  
  
else  
  
    msgbox "Goodbye"
```

```
end If
```

The first block of code will be executed if the condition is true, and the other block will be executed otherwise (if i is not equal to 10).

---

### ***If...Then....Elseif***

You can use the if...then...elseif statement if you want to select one of many blocks of code to execute:

```
if payment="Cash" then
    msgbox "You are going to pay cash!"
elseif payment="Visa" then
    msgbox "You are going to pay with visa."
elseif payment="AmEx" then
    msgbox "You are going to pay with American Express."
else
    msgbox "Unknown method of payment."
end If
```

---

### ***Select Case***

You can also use the SELECT statement if you want to select one of many blocks of code to execute:

```
select case payment
case "Cash"
    msgbox "You are going to pay cash"
case "Visa"
    msgbox "You are going to pay with visa"
case "AmEx"
    msgbox "You are going to pay with American Express"
case Else
    msgbox "Unknown method of payment"
end select
```

You can control the flow of your script with conditional statements and looping statements.

#### Controlling Program Execution

Using conditional statements, you can write VBScript code that makes decisions and repeats actions. The following conditional statements are available in VBScript:

- [If...Then...Else](#) statement
- [Select Case](#) statement

#### Making Decisions Using If...Then...Else

The If...Then...Else statement is used to evaluate whether a condition is **True** or **False** and, depending on the result, to specify one or more statements to run. Usually the condition is an expression that uses a comparison operator to compare one value or variable with another. For information about comparison operators, see [Comparison Operators](#). If...Then...Else statements can be nested to as many levels as you need.

#### Running Statements if a Condition is True

To run only one statement when a condition is **True**, use the single-line syntax for the If...Then...Else statement. The following example shows the single-line syntax. Notice that this example omits the **Else** keyword.

```
Sub FixDate()  
  
    Dim myDate  
  
    myDate = #2/13/95#  
  
    If myDate < Now Then myDate = Now  
  
End Sub
```

To run more than one line of code, you must use the multiple-line (or block) syntax. This syntax includes the **End If** statement, as shown in the following example:

```
Sub AlertUser(value)  
  
    If value = 0 Then  
  
        AlertLabel.ForeColor = vbRed  
  
        AlertLabel.Font.Bold = True  
  
        AlertLabel.Font.Italic = True  
  
    End If  
  
End Sub
```

#### Running Certain Statements if a Condition is True and Running Others if a Condition is False

You can use an If...Then...Else statement to define two blocks of executable statements: one block to run if the condition is **True**, the other block to run if the condition is **False**.

```
Sub AlertUser(value)  
  
    If value = 0 Then
```

```

    AlertLabel.ForeColor = vbRed

    AlertLabel.Font.Bold = True

    AlertLabel.Font.Italic = True

Else

    AlertLabel.ForeColor = vbBlack

    AlertLabel.Font.Bold = False

    AlertLabel.Font.Italic = False

End If

End Sub

```

### Deciding Between Several Alternatives

A variation on the **If...Then...Else** statement allows you to choose from several alternatives. Adding **ElseIf** clauses expands the functionality of the **If...Then...Else** statement so you can control program flow based on different possibilities. For example:

```

Sub ReportValue(value)

    If value = 0 Then

        MsgBox value

    ElseIf value = 1 Then

        MsgBox value

    ElseIf value = 2 then

        MsgBox value

    Else

        MsgBox "Value out of range!"

    End If

```

You can add as many **ElseIf** clauses as you need to provide alternative choices. Extensive use of the **ElseIf** clauses often becomes cumbersome. A better way to choose between several alternatives is the **Select Case** statement.

### Making Decisions with Select Case

The **Select Case** structure provides an alternative to **If...Then...ElseIf** for selectively executing one block of statements from among multiple blocks of statements. A **Select Case** statement provides capability similar to the **If...Then...Else statement**, but it makes code more efficient and readable.

A **Select Case** structure works with a single test expression that is evaluated once, at the top of the structure. The result of the expression is then compared with the values for each **Case** in the structure. If there is a match, the block of statements associated with that **Case** is executed, as in the following example.

```
Select Case Document.Form1.CardType.Options(SelectedIndex).Text

    Case "MasterCard"

        DisplayMCLogo

        ValidateMCAccount

    Case "Visa"

        DisplayVisaLogo

        ValidateVisaAccount

    Case "American Express"

        DisplayAMEXCOLogo

        ValidateAMEXCOAccount

    Case Else

        DisplayUnknownImage

        PromptAgain

End Select
```

Notice that the **Select Case** structure evaluates an expression once at the top of the structure. In contrast, the **If...Then...ElseIf** structure can evaluate a different expression for each **ElseIf** statement. You can replace an **If...Then...ElseIf** structure with a **Select Case** structure only if each **ElseIf** statement evaluates the same expression.

## VBScript Looping Statements

---

### *Looping Statements*

Very often when you write code, you want to allow the same block of code to run a number of times. You can use looping statements in your code to do this.

In VBScript we have four looping statements:

- **For...Next statement** - runs statements a specified number of times.
- **For Each...Next statement** - runs statements for each item in a collection or each element of an array
- **Do...Loop statement** - loops while or until a condition is true

- **While...Wend statement** - Do not use it - use the Do...Loop statement instead
- 

### ***For...Next Loop***

You can use a **For...Next** statement to run a block of code, when you know how many repetitions you want.

You can use a counter variable that increases or decreases with each repetition of the loop, like this:

```
For i=1 to 10
    some code
Next
```

The **For** statement specifies the counter variable (**i**) and its start and end values. The **Next** statement increases the counter variable (**i**) by one.

### **Step Keyword**

Using the **Step** keyword, you can increase or decrease the counter variable by the value you specify.

In the example below, the counter variable (**i**) is increased by two each time the loop repeats.

```
For i=2 To 10 Step 2
    some code
Next
```

To decrease the counter variable, you must use a negative **Step** value. You must specify an end value that is less than the start value.

In the example below, the counter variable (**i**) is decreased by two each time the loop repeats.

```
For i=10 To 2 Step -2
    some code
Next
```

### **Exit a For...Next**

You can exit a For...Next statement with the Exit For keyword.

---

### ***For Each...Next Loop***

A **For Each...Next** loop repeats a block of code for each item in a collection, or for each element of an array.

```
dim cars(2)
cars(0)="Volvo"
```

```
cars (1)="Saab"  
  
cars (2)="BMW"  
  
For Each x in cars  
    document.write(x & "<br />")  
Next
```

### **Do...Loop**

You can use Do...Loop statements to run a block of code when you do not know how many repetitions you want. The block of code is repeated while a condition is true or until a condition becomes true.

#### **Repeating Code While a Condition is True**

You use the While keyword to check a condition in a Do...Loop statement.

```
Do While i>10  
    some code  
Loop
```

If **i** equals 9, the code inside the loop above will never be executed.

```
Do  
    some code  
Loop While i>10
```

The code inside this loop will be executed at least one time, even if **i** is less than 10.

#### **Repeating Code Until a Condition Becomes True**

You use the Until keyword to check a condition in a Do...Loop statement.

```
Do Until i=10  
    some code  
Loop
```

If **i** equals 10, the code inside the loop will never be executed.

```
Do
```

```
some code  
Loop Until i=10
```

The code inside this loop will be executed at least one time, even if **i** is equal to 10.

### Exit a Do...Loop

You can exit a Do...Loop statement with the Exit Do keyword.

```
Do Until i=10  
  
    i=i-1  
  
    If i<10 Then Exit Do  
  
Loop
```

The code inside this loop will be executed as long as **i** is different from 10, and as long as **i** is greater than 10.

Looping allows you to run a group of statements repeatedly. Some loops repeat statements until a condition is **False**; others repeat statements until a condition is **True**. There are also loops that repeat statements a specific number of times.

The following looping statements are available in VBScript:

- [Do...Loop](#): Loops while or until a condition is **True**.
- [While...Wend](#): Loops while a condition is **True**.
- [For...Next](#): Uses a counter to run statements a specified number of times.
- [For Each...Next](#): Repeats a group of statements for each item in a collection or each element of an array.

#### Using Do Loops

You can use Do...Loop statements to run a block of statements an indefinite number of times. The statements are repeated either while a condition is **True** or until a condition becomes **True**.

#### Repeating Statements While a Condition is True

Use the **While** keyword to check a condition in a Do...Loop statement. You can check the condition before you enter the loop (as shown in the following ChkFirstWhile example), or you can check it after the loop has run at least once (as shown in the ChkLastWhile example). In the ChkFirstWhile procedure, if **myNum** is set to 9 instead of 20, the statements inside the loop will never run. In the ChkLastWhile procedure, the statements inside the loop run only once because the condition is already **False**.

```
Sub ChkFirstWhile()  
  
    Dim counter, myNum  
  
    counter = 0  
  
    myNum = 20
```

```

Do While myNum > 10
    myNum = myNum - 1
    counter = counter + 1
Loop
MsgBox "The loop made " & counter & " repetitions."
End Sub

```

```

Sub ChkLastWhile()
    Dim counter, myNum
    counter = 0
    myNum = 9
    Do
        myNum = myNum - 1
        counter = counter + 1
    Loop While myNum > 10
    MsgBox "The loop made " & counter & " repetitions."
End Sub

```

### **Repeating a Statement Until a Condition Becomes True**

There are two ways to use the **Until** keyword to check a condition in a **Do...Loop** statement. You can check the condition before you enter the loop (as shown in the following **ChkFirstUntil** example), or you can check it after the loop has run at least once (as shown in the **ChkLastUntil** example). As long as the condition is **False**, the looping occurs.

```

Sub ChkFirstUntil()
    Dim counter, myNum
    counter = 0
    myNum = 20
    Do Until myNum = 10
        myNum = myNum - 1
        counter = counter + 1
    Loop

```

```
Loop
    MsgBox "The loop made " & counter & " repetitions."
End Sub
```

```
Sub ChkLastUntil()
    Dim counter, myNum
    counter = 0
    myNum = 1
    Do
        myNum = myNum + 1
        counter = counter + 1
    Loop Until myNum = 10
    MsgBox "The loop made " & counter & " repetitions."
End Sub
```

### **Exiting a Do...Loop Statement from Inside the Loop**

You can exit a **Do...Loop** by using the **Exit Do** statement. Because you usually want to exit only in certain situations, such as to avoid an endless loop, you should use the **Exit Do** statement in the **True** statement block of an **If...Then...Else** statement. If the condition is **False**, the loop runs as usual.

In the following example, `myNum` is assigned a value that creates an endless loop. The **If...Then...Else** statement checks for this condition, preventing the endless repetition.

```
Sub ExitExample()
    Dim counter, myNum
    counter = 0
    myNum = 9
    Do Until myNum = 10
        myNum = myNum - 1
        counter = counter + 1
        If myNum < 10 Then Exit Do
    Loop
```

```
MsgBox "The loop made " & counter & " repetitions."
```

```
End Sub
```

#### Using While...Wend

The **While...Wend** statement is provided in VBScript for those who are familiar with its usage. However, because of the lack of flexibility in **While...Wend**, it is recommended that you use **Do...Loop** instead.

#### Using For...Next

You can use **For...Next** statements to run a block of statements a specific number of times. For loops, use a counter variable whose value increases or decreases with each repetition of the loop.

The following example causes a procedure called `MyProc` to execute 50 times. The **For** statement specifies the counter variable `x` and its start and end values. The **Next** statement increments the counter variable by 1.

```
Sub DoMyProc50Times()
```

```
Dim x
```

```
For x = 1 To 50
```

```
MyProc
```

```
Next
```

```
End Sub
```

Using the **Step** keyword, you can increase or decrease the counter variable by the value you specify. In the following example, the counter variable `j` is incremented by 2 each time the loop repeats. When the loop is finished, the total is the sum of 2, 4, 6, 8, and 10.

```
Sub TwosTotal()
```

```
Dim j, total
```

```
For j = 2 To 10 Step 2
```

```
total = total + j
```

```
Next
```

```
MsgBox "The total is " & total
```

```
End Sub
```

To decrease the counter variable, use a negative **Step** value. You must specify an end value that is less than the start value. In the following example, the counter variable `myNum` is decreased by 2 each time the loop repeats. When the loop is finished, total is the sum of 16, 14, 12, 10, 8, 6, 4, and 2.

```
Sub NewTotal()
```

```
Dim myNum, total
```

```

For myNum = 16 To 2 Step -2
    total = total + myNum
Next

MsgBox "The total is " & total

End Sub

```

You can exit any **For...Next** statement before the counter reaches its end value by using the **Exit For** statement. Because you usually want to exit only in certain situations, such as when an error occurs, you should use the **Exit For** statement in the **True** statement block of an **If...Then...Else** statement. If the condition is **False**, the loop runs as usual.

Using For Each...Next

A **For Each...Next** loop is similar to a **For...Next** loop. Instead of repeating the statements a specified number of times, a **For Each...Next** loop repeats a group of statements for each item in a collection of objects or for each element of an array. This is especially helpful if you don't know how many elements are in a collection. In the following HTML code example, the contents of a **Dictionary** object is used to place text in several text boxes.

```

<HTML>

<HEAD><TITLE>Forms and Elements</TITLE></HEAD>

<SCRIPT LANGUAGE="VBScript">

<!--

Sub cmdChange_OnClick

    Dim d    'Create a variable

    Set d = CreateObject("Scripting.Dictionary")

    d.Add "0", "Athens"    'Add some keys and items

    d.Add "1", "Belgrade"

    d.Add "2", "Cairo"

    For Each I in d

        Document.frmForm.Elements(I).Value = D.Item(I)

    Next

End Sub

-->

```

```
</SCRIPT>

<BODY>

<CENTER>

<FORM NAME="frmForm"

<Input Type = "Text"><p>
<Input Type = "Text"><p>
<Input Type = "Text"><p>
<Input Type = "Text"><p>
<Input Type = "Button" NAME="cmdChange" VALUE="Click Here"><p>

</FORM>

</CENTER>

</BODY>

</HTML>
```

## VBScript Procedures

In VBScript, there are two kinds of procedures; the [Sub](#) procedure and the [Function](#) procedure.

### Sub Procedures

A Sub procedure is a series of VBScript statements (enclosed by Sub and End Sub statements) that perform actions but don't return a value. A Sub procedure can take arguments (constants, variables, or expressions that are passed by a calling procedure). If a Sub procedure has no arguments, its Sub statement must include an empty set of parentheses ().

The following Sub procedure uses two intrinsic, or built-in, VBScript functions, [MsgBox](#) and [InputBox](#), to prompt a user for information. It then displays the results of a calculation based on that information. The calculation is performed in a Function procedure created using VBScript. The Function procedure is shown after the following discussion.

```
Sub ConvertTemp()
    temp = InputBox("Please enter the temperature in degrees F.", 1)
    MsgBox "The temperature is " & Celsius(temp) & " degrees C."
End Sub
```

### Function Procedures

A Function procedure is a series of VBScript statements enclosed by the Function and End Function statements. A Function procedure is similar to a Sub procedure, but can also return a value. A Function procedure can take arguments (constants, variables, or expressions that are passed to it by a calling procedure). If a Function procedure has no arguments, its Function statement must include an empty set of parentheses. A Function returns a value by assigning a value to its name in one or more statements of the procedure. The return type of a Function is always a Variant.

In the following example, the Celsius function calculates degrees Celsius from degrees Fahrenheit. When the function is called from the ConvertTemp Sub procedure, a variable containing the argument value is passed to the function. The result of the calculation is returned to the calling procedure and displayed in a message box.

```
Sub ConvertTemp()  
    temp = InputBox("Please enter the temperature in degrees F.", 1)  
    MsgBox "The temperature is " & Celsius(temp) & " degrees C."  
End Sub
```

```
Function Celsius(fDegrees)  
    Celsius = (fDegrees - 32) * 5 / 9  
End Function
```

#### Getting Data into and out of Procedures

Each piece of data is passed into your procedures using an argument . Arguments serve as placeholders for the data you want to pass into your procedure. You can name your arguments any valid variable name. When you create a procedure using either the Sub statement or the Function statement, parentheses must be included after the name of the procedure. Any arguments are placed inside these parentheses, separated by commas. For example, in the following example, `fDegrees` is a placeholder for the value being passed into the Celsius function for conversion.

```
Function Celsius(fDegrees)  
    Celsius = (fDegrees - 32) * 5 / 9  
End Function
```

To get data out of a procedure, you must use a Function. Remember, a Function procedure can return a value; a Sub procedure can't.

## VBScript Functions

This page contains all the built-in VBScript functions. The page is divided into following sections:

### ***Date/Time Functions***

Function	Description
<a href="#">CDate</a>	Converts a valid date and time expression to the variant of subtype Date
<a href="#">Date</a>	Returns the current system date

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

## **Conversion Functions**

[Top](#)

<b>Function</b>	<b>Description</b>
<a href="#">Asc</a>	Converts the first letter in a string to ANSI code
<a href="#">CBool</a>	Converts an expression to a variant of subtype Boolean
<a href="#">CByte</a>	Converts an expression to a variant of subtype Byte
<a href="#">CCur</a>	Converts an expression to a variant of subtype Currency
<a href="#">CDate</a>	Converts a valid date and time expression to the variant of subtype Date
<a href="#">Cdbl</a>	Converts an expression to a variant of subtype Double
<a href="#">Chr</a>	Converts the specified ANSI code to a character
<a href="#">CInt</a>	Converts an expression to a variant of subtype Integer

<a href="#">CLng</a>	Converts an expression to a variant of subtype Long
<a href="#">CSng</a>	Converts an expression to a variant of subtype Single
<a href="#">CStr</a>	Converts an expression to a variant of subtype String
<a href="#">Hex</a>	Returns the hexadecimal value of a specified number
<a href="#">Oct</a>	Returns the octal value of a specified number

## **Format Functions**

[Top](#)

<b>Function</b>	<b>Description</b>
<a href="#">FormatCurrency</a>	Returns an expression formatted as a currency value
<a href="#">FormatDateTime</a>	Returns an expression formatted as a date or time
<a href="#">FormatNumber</a>	Returns an expression formatted as a number
<a href="#">FormatPercent</a>	Returns an expression formatted as a percentage

## **Math Functions**

[Top](#)

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

## **Array Functions**

[Top](#)

<b>Function</b>	<b>Description</b>
<a href="#">Array</a>	Returns a variant containing an array

<a href="#">Filter</a>	Returns a zero-based array that contains a subset of a string array based on a filter criteria
<a href="#">IsArray</a>	Returns a Boolean value that indicates whether a specified variable is an array
<a href="#">Join</a>	Returns a string that consists of a number of substrings in an array
<a href="#">LBound</a>	Returns the smallest subscript for the indicated dimension of an array
<a href="#">Split</a>	Returns a zero-based, one-dimensional array that contains a specified number of substrings
<a href="#">UBound</a>	Returns the largest subscript for the indicated dimension of an array

## String Functions

[Top](#)

Function	Description
<a href="#">InStr</a>	Returns the position of the first occurrence of one string within another. The search begins at the first character of the string
<a href="#">InStrRev</a>	Returns the position of the first occurrence of one string within another. The search begins at the last character of the string
<a href="#">LCase</a>	Converts a specified string to lowercase
<a href="#">Left</a>	Returns a specified number of characters from the left side of a string
<a href="#">Len</a>	Returns the number of characters in a string
<a href="#">LTrim</a>	Removes spaces on the left side of a string
<a href="#">RTrim</a>	Removes spaces on the right side of a string
<a href="#">Trim</a>	Removes spaces on both the left and the right side of a string
<a href="#">Mid</a>	Returns a specified number of characters from a string
<a href="#">Replace</a>	Replaces a specified part of a string with another string a specified number of times
<a href="#">Right</a>	Returns a specified number of characters from the right side of a string
<a href="#">Space</a>	Returns a string that consists of a specified number of spaces
<a href="#">StrComp</a>	Compares two strings and returns a value that represents the result of the comparison
<a href="#">String</a>	Returns a string that contains a repeating character of a specified length
<a href="#">StrReverse</a>	Reverses a string
<a href="#">UCase</a>	Converts a specified string to uppercase

## Other Functions

[Top](#)

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

## VBScript Data Types

VBScript has only one data type called a Variant. A Variant is a special kind of data type that can contain different kinds of information, depending on how it is used. Because Variant is the only data type in VBScript, it is also the data type returned by all functions in VBScript.

At its simplest, a Variant can contain either numeric or string information. A Variant behaves as a number when you use it in a numeric context and as a string when you use it in a string context. That is, if you are working with data that looks like numbers, VBScript assumes that it is numbers and does what is most appropriate for numbers. Similarly, if you're working with data that can only be string data, VBScript treats it as string data. You can always make numbers behave as strings by enclosing them in quotation marks (" ").

#### Variant Subtypes

Beyond the simple numeric or string classifications, a Variant can make further distinctions about the specific nature of numeric information. For example, you can have numeric information that represents a date or a time. When used with other date or time data, the result is always expressed as a date or a time. You can also have a rich variety of numeric information ranging in size from Boolean values to huge floating-point numbers. These different categories of information that can be contained in a Variant are called subtypes. Most of the time, you can just put the kind of data you want in a Variant, and the Variant behaves in a way that is most appropriate for the data it contains.

The following table shows subtypes of data that a Variant can contain.

Subtype	Description
Empty	Variant is uninitialized. Value is 0 for numeric variables or a zero-length string ("") for string variables.
Null	Variant intentionally contains no valid data.
Boolean	Contains either <a href="#">True</a> or <a href="#">False</a> .
Byte	Contains integer in the range 0 to 255.
Integer	Contains integer in the range -32,768 to 32,767.
Currency	-922,337,203,685,477.5808 to 922,337,203,685,477.5807.
Long	Contains integer in the range -2,147,483,648 to 2,147,483,647.
Single	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.
Date (Time)	Contains a number that represents a date between January 1, 100 to December 31, 9999.
String	Contains a variable-length string that can be up to approximately 2 billion characters in length.
Object	Contains an object.

Error	Contains an error number.
-------	---------------------------

## VBScript Features

The following table is a list of VBScript features.

Features by Category

Category	Keywords
Array handling	<a href="#">ArrayDim</a> , <a href="#">Private</a> , <a href="#">Public</a> , <a href="#">ReDimIsArray</a> <a href="#">Erase</a> <a href="#">LBound</a> , <a href="#">UBound</a>
Assignments	<a href="#">Set</a>
Comments	<a href="#">Comments using ' or Rem</a>
Constants/ Literals	<a href="#">EmptyNothing</a> <a href="#">Null</a> <a href="#">True</a> , <a href="#">False</a>
Control flow	<a href="#">Do...LoopFor...Next</a> <a href="#">For Each...Next</a> <a href="#">If...Then...Else</a> <a href="#">Select Case</a> <a href="#">While...Wend</a> <a href="#">With</a>
Conversions	<a href="#">AbsAsc</a> , <a href="#">AscB</a> , <a href="#">AscW</a> <a href="#">Chr</a> , <a href="#">ChrB</a> , <a href="#">ChrW</a> <a href="#">CBool</a> , <a href="#">CByteCCur</a> , <a href="#">CDateCdbl</a> , <a href="#">CIntCLng</a> , <a href="#">CSng</a> , <a href="#">CStrDateSerial</a> , <a href="#">DateValueHex</a> , <a href="#">OctFix</a> , <a href="#">IntSgn</a> <a href="#">TimeSerial</a> , <a href="#">TimeValue</a>
Dates/Times	<a href="#">Date</a> , <a href="#">TimeDateAdd</a> , <a href="#">DateDiff</a> , <a href="#">DatePartDateSerial</a> , <a href="#">DateValueDay</a> , <a href="#">Month</a> , <a href="#">MonthNameWeekday</a> , <a href="#">WeekdayName</a> , <a href="#">YearHour</a> , <a href="#">Minute</a> , <a href="#">SecondNow</a> <a href="#">TimeSerial</a> , <a href="#">TimeValue</a>
Declarations	<a href="#">ClassConst</a> <a href="#">Dim</a> , <a href="#">Private</a> , <a href="#">Public</a> , <a href="#">ReDimFunction</a> , <a href="#">SubProperty</a> <a href="#">Get</a> , <a href="#">Property</a> <a href="#">Let</a> , <a href="#">Property</a> <a href="#">Set</a>
Error Handling	<a href="#">On ErrorErr</a>
Expressions	<a href="#">EvalExecute</a> <a href="#">RegExp</a> <a href="#">Replace</a> <a href="#">Test</a>
Formatting Strings	<a href="#">FormatCurrencyFormatDateTime</a> <a href="#">FormatNumber</a> <a href="#">FormatPercent</a>
Input/Output	<a href="#">InputBoxLoadPicture</a> <a href="#">MsgBox</a>
Literals	<a href="#">EmptyFalse</a> <a href="#">Nothing</a> <a href="#">Null</a> <a href="#">True</a>
Math	<a href="#">Atn</a> , <a href="#">Cos</a> , <a href="#">Sin</a> , <a href="#">TanExp</a> , <a href="#">Log</a> , <a href="#">SqrRandomize</a> , <a href="#">Rnd</a>
Miscellaneous	<a href="#">Eval</a> <a href="#">FunctionExecute</a> <a href="#">Statement</a> <a href="#">RGB</a> <a href="#">Function</a>
Objects	<a href="#">CreateObjectErr</a> <a href="#">Object</a> <a href="#">GetObject</a> <a href="#">RegExp</a>
Operators	<a href="#">Addition (+)</a> , <a href="#">Subtraction (-)</a> <a href="#">Exponentiation (^)</a> <a href="#">Modulus arithmetic (Mod)</a> <a href="#">Multiplication (*)</a> , <a href="#">Division (/)</a> <a href="#">Integer Division (\)</a> <a href="#">Negation (-)</a> <a href="#">String concatenation (&amp;)</a> <a href="#">Equality (=)</a> ,

	<a href="#">Inequality (&lt;&gt;)</a> <a href="#">Less Than (&lt;)</a> , <a href="#">Less Than or Equal To (&lt;=)</a> <a href="#">Greater Than (&gt;)</a> <a href="#">Greater Than or Equal To (&gt;=)</a> <a href="#">Is And</a> , <a href="#">Or</a> , <a href="#">XorEqv</a> , <a href="#">Imp</a>
Options	<a href="#">Option Explicit</a>
Procedures	<a href="#">CallFunction</a> , <a href="#">SubProperty Get</a> , <a href="#">Property Let</a> , <a href="#">Property Set</a>
Rounding	<a href="#">AbsInt</a> , <a href="#">Fix</a> , <a href="#">RoundSgn</a>
Script Engine ID	<a href="#">ScriptEngineScriptEngineBuildVersion</a> <a href="#">ScriptEngineMajorVersion</a> <a href="#">ScriptEngineMinorVersion</a>
Strings	<a href="#">Asc</a> , <a href="#">AscB</a> , <a href="#">AscWChr</a> , <a href="#">ChrB</a> , <a href="#">ChrWFilter</a> , <a href="#">InStr</a> , <a href="#">InStrBInStrRev</a> <a href="#">Join</a> <a href="#">Len</a> , <a href="#">LenBLCase</a> , <a href="#">UCaseLeft</a> , <a href="#">LeftBMid</a> , <a href="#">MidBRight</a> , <a href="#">RightBReplace</a> <a href="#">Space</a> <a href="#">Split</a> <a href="#">StrComp</a> <a href="#">String</a> <a href="#">StrReverse</a> <a href="#">LTrim</a> , <a href="#">RTrim</a> , <a href="#">Trim</a>
Variants	<a href="#">IsArray</a> <a href="#">IsDate</a> <a href="#">IsEmpty</a> <a href="#">IsNull</a> <a href="#">IsNumeric</a> <a href="#">IsObject</a> <a href="#">TypeName</a> <a href="#">VarType</a>

## VBScript Variables

A variable is a convenient placeholder that refers to a computer memory location where you can store program information that may change during the time your script is running. For example, you might create a variable called `ClickCount` to store the number of times a user clicks an object on a particular Web page. Where the variable is stored in computer memory is unimportant. What is important is that you only have to refer to a variable by name to see or change its value. In VBScript, variables are always of one fundamental data type, [Variant](#).

### Declaring Variables

You declare variables explicitly in your script using the [Dim](#) statement, the [Public](#) statement, and the [Private](#) statement. For example:

```
Dim DegreesFahrenheit
```

You declare multiple variables by separating each variable name with a comma. For example:

```
Dim Top, Bottom, Left, Right
```

You can also declare a variable implicitly by simply using its name in your script. That is not generally a good practice because you could misspell the variable name in one or more places, causing unexpected results when your script is run. For that reason, the [Option Explicit](#) statement is available to require explicit declaration of all variables. The **Option Explicit** statement should be the first statement in your script.

### Naming Restrictions

Variable names follow the standard rules for naming anything in VBScript. A variable name:

- Must begin with an alphabetic character.
- Cannot contain an embedded period.
- Must not exceed 255 characters.
- Must be unique in the scope in which it is declared.

### Scope and Lifetime of Variables

A variable's scope is determined by where you declare it. When you declare a variable within a procedure, only code within that procedure can access or change the value of that variable. It has local scope and is a procedure-level variable. If you declare a variable outside a procedure, you make it recognizable to all the procedures in your script. This is a script-level variable, and it has script-level scope.

The lifetime of a variable depends on how long it exists. The lifetime of a script-level variable extends from the time it is declared until the time the script is finished running. At procedure level, a variable exists only as long as you are in the procedure. When the procedure exits, the variable is destroyed. Local variables are ideal as temporary storage space when a procedure is executing. You can have local variables of the same name in several different procedures because each is recognized only by the procedure in which it is declared.

### Assigning Values to Variables

Values are assigned to variables creating an expression as follows: the variable is on the left side of the expression and the value you want to assign to the variable is on the right

## VBScript Constants

A constant is a meaningful name that takes the place of a number or string and never changes. VBScript defines a number of intrinsic constants . You can get information about these intrinsic constants from the

### Creating Constants

You create user-defined constants in VBScript using the [Const](#) statement. Using the Const statement, you can create string or numeric constants with meaningful names and assign them literal values. For example:

```
Const MyString = "This is my string."  
Const MyAge = 49
```

Note that the string literal is enclosed in quotation marks (" "). Quotation marks are the most obvious way to differentiate string values from numeric values. You represent Date literals and time literals by enclosing them in number signs (#). For example:

```
Const CutoffDate = #6-1-97#
```

You may want to adopt a naming scheme to differentiate constants from variables. This will prevent you from trying to reassign constant values while your script is running. For example, you might want to use a "vb" or "con" prefix on your constant names, or you might name your constants in all capital letters. Differentiating constants from variables eliminates confusion as you develop more complex scripts.

## VBScript Operators

VBScript has a full range of operators, including [arithmetic operators](#), [comparison operators](#), [concatenation operators](#), and [logical operators](#).

### Operator Precedence

When several operations occur in an expression, each part is evaluated and resolved in a predetermined order called operator precedence. You can use parentheses to override the order of precedence and force some parts of an expression to be evaluated before others. Operations within parentheses are always performed before those outside. Within parentheses, however, standard operator precedence is maintained.

When expressions contain operators from more than one category, arithmetic operators are evaluated first, comparison operators are evaluated next, and logical operators are evaluated last. Comparison operators all have equal precedence; that is, they are evaluated in the left-to-right order in which they appear. Arithmetic and logical operators are evaluated in the following order of precedence.

#### Arithmetic

Description	Symbol
<a href="#">Exponentiation</a>	^
<a href="#">Unary negation</a>	-
<a href="#">Multiplication</a>	*
<a href="#">Division</a>	/
<a href="#">Integer division</a>	\
<a href="#">Modulus arithmetic</a>	Mod
<a href="#">Addition</a>	+
<a href="#">Subtraction</a>	-
<a href="#">String concatenation</a>	&

#### Comparison

Description	Symbol
<a href="#">Equality</a>	=
<a href="#">Inequality</a>	<>
<a href="#">Less than</a>	<
<a href="#">Greater than</a>	>
<a href="#">Less than or equal to</a>	<=
<a href="#">Greater than or equal to</a>	>=
<a href="#">Object equivalence</a>	Is

#### Logical

Description	Symbol
<a href="#">Logical negation</a>	Not
<a href="#">Logical conjunction</a>	And
<a href="#">Logical disjunction</a>	Or

<a href="#">Logical exclusion</a>	Xor
<a href="#">Logical equivalence</a>	Eqv
<a href="#">Logical implication</a>	Imp

When multiplication and division occur together in an expression, each operation is evaluated as it occurs from left to right. Likewise, when addition and subtraction occur together in an expression, each operation is evaluated in order of appearance from left to right.

The string concatenation (&) operator is not an arithmetic operator, but in precedence it falls after all arithmetic operators and before all comparison operators. The **Is** operator is an object reference comparison operator. It does not compare objects or their values; it checks only to determine if two object references refer to the same object.

## VBScript Interfaces in SQL Server 2000 Let You Transform Data and Provide Reports to Your Users

**SUMMARY** Application service providers often must send information to clients automatically rather than on-demand. For example, a manufacturer may want to know each day how many of their products were sold by a retail chain. While SQL Server is ideal for maintaining this type of database, you have to write scripts to get the data out in a client-friendly format. Here you will see how you can use Data Transformation Services (DTS), a powerful tool in SQL Server, to automate the retrieval and formatting of data from SQL Server 2000 and make the process of pushing data to your users a lot easier. Push, also known as Web-casting, automates the search and retrieval of data. It is initiated by Web or database servers rather than users. Based on some defined criteria, a push application will automatically search a database and deliver the desired information when and where the application directs (frequently straight to a user's desktop via e-mail). This is not only a convenient means of receiving important information that a user may not otherwise get easily, it's substantially changing the way in which day-to-day business is conducted.

Push applications generally come in two forms. One type, which is often offered by large news and database groups, includes e-mail, listserves, and direct delivery services. Push applications can also enable developers and users to create profiles and record their preferences so they can receive relevant information from multiple sources. The sample application we'll describe here is based upon the first type of push application. However, our application is generic enough to support a custom profiling functionality as well.

Organizations that need to centralize and consolidate data can use Data Transformation Services (DTS) tools in SQL Server™ 2000 to retrieve and transform data from different sources into a channeled destination. Using these tools, you can perform a simple one-time data transfer or develop complex workflow-driven packages. The DTS tools also provide a graphical user interface and an object model that can be programmed relatively easily.

A DTS package is a combination of connections, tasks, transformations, and workflow constraints. Each package can contain one or more steps or tasks that can be executed

sequentially or in parallel when the package is run. When executed, the package connects to the correct data sources, copies data and database objects, transforms data, and notifies other users or processes of events. Packages can be edited, password protected, scheduled for execution, and retrieved by version. You can easily create a package using the DTS Designer tool that we will describe later in this article.

With VBScript or JScript® you can create a task that performs functions that are not available in the other tasks in DTS. For example, you can:

- Create and use ADO connections, commands, recordsets, and other objects to access and manipulate data
- Format and transform the data using functions, subroutines, and COM objects
- Create, use, and modify the values stored in DTS global variables and ActiveX® script constants
- Integrate other DTS tasks and workflows

## Sample Application

Our sample app will show you how to use a VBScript task to perform functions that are not available in the other DTS tasks and how VBScript scripts can function as scheduled tasks. We will not discuss DTS global variables, workflows, or integration with other tasks because our purpose is to show the power of VBScript within SQL Server 2000.

**Figure 1** outlines the steps we followed to develop our sample DTS application. The VBScript task in the DTS package e-mails periodic information to the authors regarding sales of their books. This data is formatted into user-friendly reports. We are using the Pubs database from SQL Server 2000 as our data source, but the technique is generic and can be applied to any data model.

### Figure 1 Workflow

Our first step was to populate the e-mail field with the author's e-mail address. To do this, we selected the Pubs database and extended the authors table by adding a varChar field of size 20, naming it au\_email, as you can see in **Figure 2**. We assume that a separate GUI or process would be responsible for populating this field in a production application, but for demonstration purposes you can enter these addresses manually.

### Figure 2 Adding an E-mail Field

Next we created a new local DTS package in SQL Server 2000. To open the DTS Designer to create the package, you need to start Enterprise Manager in SQL Server, right-click on Data Transformation Services in the console tree, and choose New Package, as shown in **Figure 3**. With the DTS Designer GUI you can build and configure packages by dragging and dropping methods and by completing property sheets on the various DTS objects composing the package.

### Figure 3 Create New Package

Our third step was to add a VBScript task by dragging and dropping the ActiveX Script Task from the Task toolbar to the design surface, as shown in **Figure 4**. You can change the name of a task from "ActiveX Script Task: undefined" to whatever is appropriate for your application.

### Figure 4 ActiveX Task Properties

Our next step was to develop functions within the VBScript task to retrieve, transform, and e-mail the data to the users. To make it easy to explore our sample application, just copy the functions from our sample code (see the link at the top of this article). Within the ActiveX Script Task properties window, make sure that the language is set to VBScript Language. You can set the language by clicking on the Language Button (see **Figure 4**). Then copy our source code into the designer, deleting the default main function code you see in the designer window.

You will need to change constants in SQL Server to make the sample application work for you since these constants will vary (explained later).

Make sure you save the task when you're finished, as shown in **Figure 5**. We named our package BookSales.

### **Figure 5 Saving the Package**

You can run the sample application in several ways. First, click on the Parse button to make sure that you do not have any syntax errors. This button is available within the ActiveX Script Task properties window. Then click on the Go button in the toolbar at the top of the DTS Designer. Alternatively, you can execute the task by right-clicking on whichever task you choose and selecting Execute step from the context menu. Or you can execute the BookSales package as shown in **Figure 6** within the Enterprise Manager. Upon successful completion of the task, the e-mail (see **Figure 7**) will be sent to the authors.

### **Figure 6 Run BookSales**

Our BookSales package can automatically be scheduled to run by selecting the Schedule Package menu item shown in **Figure 6**. Enter the necessary frequency and duration parameters.

### **Figure 7 E-mail Scheduling**

To report order number, quantity, pay terms, and title, the BookSales package will run daily at 11:00 pm starting from 1/1/2002. Note that SQL Server Agent must be loaded and configured to schedule DTS packages. You should see the BookSales package as a job within the Enterprise Manager under Management | SQL Server Agent | Jobs.

## **The VBScript Code**

All our sample code is in the sourcecode.txt file available for download with this article. After we cover details on the constants that need to be set and quickly look at the main function, we'll explain how the code manages the sales data, accesses the database, formats the recordset, and sends reports.

The following three constants must be adjusted according to your system and network settings:

```
Const SMTP_SERVER = "exchange.afs-link.com"

Const SENDER_E-MAIL = """Book Sales Reporting Service"" & _
"<amehta@afs-link.com>"

Const DB_CONNECT_STRING = "Provider=SQLOLEDB.1;Data " & _
```

```
"Source=(local);Initial Catalog=Pubs;user id = 'sa';password=''"
```

The first constant, Const SMTP\_SERVER = "exchange.afs-link.com," is the SMTP server, the DNS name of your mail server. The e-mail sender function needs this information to send out the e-mail. Your network administrator will know this setting.

Const SENDER\_E-MAIL = ""Book Sales Reporting Service" <amehta@afs-link.com>" is the e-mail address of the sender. It usually is something similar to system@domainname.com. The first part of this constant (Book Sales Reporting) allows you to type in any text; typically this will be the name of the department sending the daily sales report. The actual e-mail address must be entered within the <>. Again, your network administrator should be able to tell you what to enter here.

The last constant is the database connection string that provides all the necessary information for connecting to the database. The ADO connection string looks like this:

```
Const DB_CONNECT_STRING = "Provider=SQLOLEDB.1;Data " & _  
"Source=(local); Initial Catalog=Pubs;user id = " & _  
"'sa';password=''"
```

We assume that readers have a basic knowledge of ADO, so we have included only a brief summary of what each part of the connection string means.

The database driver section of the string is the part that reads "Provider=SQLOLEDB.1;". This defines the type of ODBC driver needed to connect to the database. We are using SQL Server; if that is not the case for your application, you can find the necessary ODBC driver information at [Microsoft Open Database Connectivity](#).

The Data Source specifies the location of the server you are searching for. We have specified "local" because SQL Server is on our local machine. You may need to specify an IP address or the name of the machine on which the database resides.

Initial Catalog contains the name of the database; in our code, this is the Pubs database that comes with SQL Server. And finally, user id specifies the name of the user and password for the user.

By default, the entry function to the ActiveX task is named Main, but you can change it. The Main function consists of only two lines. First it calls the Process\_Daily\_Sales subroutine, and then it returns an ActiveX script constant (DTSTaskExecResult\_Success). In the Package Object Browser shown in **Figure 8**, you can view all of the project constants and global variables.

## Figure 8 Project Constants

These constants and global variables can be used to control execution of steps within a DTS package. For the sake of simplicity, we will not describe these constants and global variables in great detail. It is important that the Main function returns the ActiveX script constant as DTSTaskExecResult\_Success because our example is just a one-step VBScript task.

## Managing the Sales Data

The Process\_Daily\_Sales function is the guts of the application. It retrieves in recordset format the list of authors whose books have been sold today. Then it formats the recordset into an HTML table, and finally it takes this HTML table and e-mails it to each of the appropriate authors.

This function has three local variables: two ADO recordsets and one date:

```
Dim rstAuthors
```

```
Dim rstSales
```

```
Dim Todays_Date
```

Since not all Pubs databases are populated with sales data for today, the following two lines of code generate the report based upon 9/14/1994, a date we chose that will work with almost all Pubs databases. We simply uncomment the line of code to be processed and comment the other line:

```
'Todays_Date = "'" & Date() & "'"
```

```
Todays_Date = "'9/14/1994'"
```

Next, we get the authors with sales. By using simple SQL joins, we select authors whose books have been ordered today:

```
strAu_Sales = "Select Distinct Authors.* from " & _
```

```
"Authors,Sales, TitleAuthor Where " & _
```

```
"TitleAuthor.au_id = Authors.au_id and " & _
```

```
"TitleAuthor.Title_id = Sales.Title_id and " & _
```

```
"Sales.ord_date = " & Todays_Date
```

Now we convert the SQL statement into an ADO recordset with the following call:

```
Set rstAuthors = ExecuteSQL(strAu_Sales)
```

Next, we make sure that rstAuthors is in fact not empty before we iterate through to send notifications to the authors:

```
If Not (rstAuthors.EOF and rstAuthors.Bof ) Then
```

```
While Not rstAuthors.EOF
```

We extract other information, such as the store where the order was placed, order number, quantity, payment terms, and the title of the books from various tables such as Stores, Sales, TitleAuthors, and Titles. This is a simple join, as you can see:

```
strAu_Sales = "SELECT distinct stores.stor_name as [Store Name], " & _
```

```
"sales.ord_num as [Order Number], sales.qty as [Quantity], " & _
```

```
"sales.payterms as [Pay Terms], Titles.Title FROM Stores, Sales, " & _
```

```
"TitleAuthor, Titles " & "Where TitleAuthor.au_id = '" & _
```

```
"rstAuthors("au_id") & "'" and Sales.ord_date = " & Todays_Date " & _
```

```
" & "and Sales.Title_id = Titles.Title_id and sales.stor_id = " & _
```

```
"stores.stor_id "
```

We convert the SQL statement into the ADO recordset by calling the ExecuteSQL statement:

```
Set rstSales = ExecuteSQL(strAu_Sales )
```

Now we construct the body of the message with the author's name and address. The HTML tag <br> is used because we will embed the message as HTML in the e-mail:

```
strTable = rstAuthors("au_fname") & " " &  
rstAuthors("au_lname") & "<br>" & rstAuthors("Address")  
& "<br>" & rstAuthors("city") & ", " &  
rstAuthors("state") & " " & rstAuthors("Zip")
```

Now that the body of the message is ready, we extend this body with an embedded HTML table by including the sales information previously extracted. FormatRecordset will convert the rstSales ADO recordset into an HTML table:

```
strTable = strTable & FormatRecordset(rstSales)
```

Then we call the send\_e-mail function that accepts the subject, the author's e-mail address, and the HTML message body:

```
Call send_e-mail("Book Sales Report For: " & _  
& Todays_Date, rstAuthors("au_email"), strTable)
```

We then move on to the next author:

```
rstAuthors.movenext
```

```
Wend
```

The Execute\_SQL function connects directly to the database. Inside this function we access the database, execute the SQL string provided by the caller, and return the results, if there are any. First we create an ADO connection:

```
Set myConn = CreateObject("ADODB.Connection")
```

Then we create an ADO recordset:

```
set myRecordset = CreateObject("ADODB.Recordset" )
```

In the next step, we open the connection using the DB\_CONNECT\_STRING constant:

```
myConn.Open = DB_CONNECT_STRING
```

After the connection is open, we open the recordset using the connection and the SQL:

```
myRecordset.Open mySQLCmdText, myConn
```

And finally we return the results of opening the recordset:

```
Set ExecuteSQL = myRecordset
```

If you have spent any time working with ADO, you will see that all of these steps are very straightforward.

The FormatRecordset function accepts an ADO recordset as a parameter and returns an HTML table (in the form of a string variable). It has two loops—one inside the other. First, it steps through the list of fields and then it loops through the number of rows. These iterations are used to convert the recordset to an HTML table.

The HTML table is created as a string and a simple table is defined as the following:

```
strTable = "<table border=1 width=500>"
```

Then we move to the first record and create the table row with the <tr> tag:

```
rstTable.MoveFirst
```

```
strTable = strTable & "<tr>"
```

We then iterate through the number of fields and add the name to the <td> tag with nice colors:

```
For Index = 0 To rstTable.Fields.Count - 1
```

```
    strTable = strTable & "<td bgcolor='blue'><" & _
```

```
    "font color='white'>"
```

```
    strTable = strTable & _
```

```
    rstTable.Fields.Item(Index).Name
```

```
    strTable = strTable & "</font></td>"
```

```
Next
```

The <tr> tag is closed; this marks the heading for the table:

```
strTable = strTable & "</tr>"
```

Now we continue on to the data part of the recordset to fill the rest of the HTML table. We loop through all records and fill the HTML table tags <tr> and <td>:

```
While (Not rstTable.EOF)
```

```
    strTable = strTable & "<tr>"
```

```
    For Index = 0 To rstTable.Fields.Count - 1
```

```
        strTable = strTable & "<td>"
```

```
        strTable = strTable & _
```

```
        rstTable(rstTable.Fields.Item(Index).Name).Value
```

```
        strTable = strTable & "<br>"
```

```

        strTable = strTable & "</td>"

    Next

    strTable = strTable & "</tr>"

    rstTable.MoveNext

Wend

```

Finally, the table is complete and the function value is returned as a string variable type:

```

strTable = strTable & "</table>"

FormatRecordset = strTable

```

## Sending Reports

The Send\_Mail function is used to send the reports to the authors. Notice that the constants SMTP\_SERVER and SENDER\_E-MAIL are being used in this function. Send\_Mail uses the Microsoft® Collaboration Data Objects, sometimes referred to as CDO 2.0 or CDOSYS.DLL. CDO provides an object model for the development of messaging applications on Windows® 2000. CDOSYS is based on the SMTP and NNTP standards and is available as a system component on Windows 2000 Server installations. It is the standard API for building bulk-mailing and Web-based messaging applications on Windows 2000 Server.

The Send\_Mail function has three parameters: subject, rcpt, and msgHTML. Subject is the subject of the e-mail being sent, rcpt is the receiver's e-mail address, and msgHTML is the HTMLBody of the message. The report is sent to the authors in HTML format.

The Send\_Mail function uses two constants. The cdoSendUsingPickup constant indicates that the message should be sent using the local SMTP service pickup directory. The cdoSendUsingPort constant indicates that the message is using the network (SMTP over the network).

If the SMTP service is installed on the local computer, then the constant's value defaults to cdoSendUsingPickup. Otherwise, if Outlook® Express is installed, the constant's value defaults to cdoSendUsingPort and the settings from the default account are used. For this article, we use cdoSendUsingPickup.

Next, we use COM to create Message and Configuration objects as shown here:

```

set iMsg = CreateObject("CDO.Message")

set iConf = CreateObject("CDO.Configuration")

```

Configuration has several fields. Before setting the values of these fields, they are linked to the configuration by:

```

Set Flds = iConf.Fields

```

The <http://schemas.microsoft.com/cdo/configuration/> namespace defines the majority of fields used to set configurations for various CDO objects. We set and update the following three fields (SendUsing, SMTP\_SERVER, and TimeOut) of the Configuration object:

```

With Flds

.Item("http://schemas.microsoft.com/cdo/configuration/sendusing") = _

```

```
        cdoSendUsingPickup
    .Item("http://schemas.microsoft.com/cdo/configuration/smtpserver") = _
        SMTP_SERVER
    .Item("http://schemas.microsoft.com/cdo/configuration/ " & _
        "smtpconnectiontimeout") = 10
    .Update
End With
```

Finally, the message is configured with the configuration we've just defined and with the rest of the e-mail message information before it is sent:

```
With iMsg
    Set .Configuration = iConf
    .To = rcpt
    .From = SENDER_EMAIL
    .Subject = subject
    .HTMLBody = msgHTML
    .Send
End With
```