VBScript – The Basics

When users interact with computers, they usually get to some point where the computer asks them for information. That information is stored or manipulated by the computer in some way. Suppose, for example, that you want to keep a record of the number of times the user has clicked the button. In that case, you would want to store a value in memory. In any case, you need a "container" in which to store information. Programmers commonly call these containers variables.

What is a variable?

A variable is a virtual container in the computer's memory or 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. 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.** A computer program can store information in a variable and then access that information later by referring to the variable's name**.**

Variables 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.
- Make sure you never create variables that have the same name as keywords already used by **VBScript**. These keywords are called reserved words and include terms such as **Date**, **Minute**, **Second**, **Time**, and so on.

How Do I Create a Variable?

When you create a variable, you have to give it a name. That way, when you need to find out what's contained in the variable, you use its name to let the computer know which variable you are referring to. You have two ways to create a variable. The first way, called the explicit method, is where you use the **Dim** keyword to tell **VBScript** you are about to create a variable. You then follow this keyword with the name of the variable. If, for example, you want to create a variable called Quantity, you would enter

```
Dim nQuantity
```

And the variable will then exist.

Declaration Statements and Highiligts

Dim Statement

Description

The Dim statement declares and allocates storage space in memory for variables. The **Dim** statement is used either at the start of a procedure or the start of a global script block. In the first case, the variable declared using **Dim** is local to the procedure. In the second, it's available throughout the module.

Syntax

```
Dim varname[([subscripts])][, varname[([subscripts])]] . . .
```

Arguments

Parameter	Description
varname	Name of the variable; follows standard variable naming conventions.
subcripts	An array and optionally specifies the number and extent of array dimensions up to 60 multiple dimensions may be declared.

Notes

- When variables are first initialized with the **Dim** statement, they have a value of **Empty**. In addition, if a variable has been initialized but not assigned a value, the following expressions will both evaluate to True:

```
If vVar = 0 Then
If vVar = "" Then
```

Overriding Standard Variable Naming Conventions

You can override standard variable naming conventions by placing your variable name in brackets. This allows you to use reserved words or illegal characters in variable names. For example:

```
Dim [Option Explicit]
[Option Explicit] = 6
Msgbox  [Option Explicit]
```

In a common usage, is not recommended to use this syntax. However, is possible. Those overriding naming convention are useful for user custom classes.
For more information about classes, in the advanced topic of **VBScript.**

Declaring Variables Explicit and Implicit

You declare variables **explicitly** in your script using the **Dim** statement, the **Public** statement, and the **Private** statement.

```
Dim nDegreesFahrenheit
```

You declare variables **explicitly** in your script using the **Dim** statement, the **Public** statement**,** and the **Private** statement**.**

```
Dim nDegreesFahrenheit
```

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

```
Dim nTop, nBottom, nLeft, nRight
```

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.

For that reason, and for coding standards proposes, it is recommended to declare variables explicitly, and force other programmers the same by adding the Option **Explicit** statement in the head of any action, reusable action and vbs files.

Option Explicit Statement

Forces explicit declaration of all variables in a script.
If used, the **Option Explicit** statement must appear in a script before any other statements. When you use the **Option Explicit** statement, you must explicitly declare all variables using the **Dim**, **Private**, **Public**, or **ReDim** statements. If you attempt to use an undeclared variable name, an error occurs like in Figure 1 .

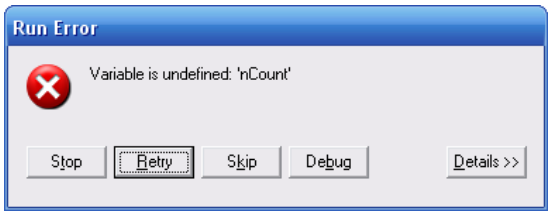

Figure 1 – Undefined variable error.

Tip

Use **Option Explicit** to avoid incorrectly typing the name of an existing variable or to avoid confusion in code where the scope of the variable is not clear.
The following example illustrates use of the **Option Explicit** statement.

```
Option Explicit        ' Force explicit variable declaration.
Dim MyVar              ' Declare variable.
MyInt = 10             ' Undeclared variable generates error.
MyVar = 10             ' Declared variable does not generate error
```

Working with Arrays

So far you've learned what a variable is, how to create one, and what you can store inside one, a variable containing a single value is a scalar variable. You might be wondering if there is some easy way to group variables together in a set.
you can create a variable that can contain a series of values. This is called an array variable. Array variables and scalar variables are declared in the same way, except that the declaration of an array variable uses parentheses ( ) following the variable name. Arrays are useful when you're storing sets of similar data because they often make it easier to manipulate the data together. If you wanted to manipulate a list of ten coordinates, you would have to execute ten different statements to handle each one. Besides, how can you be sure you have ten? What if you have only six at the moment? How can your code handle this kind of situation where you really don't know ahead of time how many pieces of information you have? Here is where the array comes to the rescue!

Scalar Variables and Array Variables

The beauty of an array is that it enables you to store and use a series of data using one variable name and an index to distinguish the individual items. By using an index, you can often make your code simpler and more efficient so that it's easier for you to put a script together and change it later. With **VBScript,** the elements inside your array can hold any kind of data. The elements don't have to be all integers or all strings, for example. The array can hold a combination of data types

Creating Arrays

You create arrays using the same keyword you use when creating variables-the **Dim** keyword. An array created with the **Dim** keyword exists as long as the procedure does and is destroyed once the procedure ends. If you create the array in the main script, outside the procedure, the values will persist as long as the page is loaded.

You can create two types of arrays using **VBScript**: fixed arrays and dynamic arrays. Fixed arrays have a specific number of elements in them, whereas dynamic arrays can vary in the number of elements depending on how many are stored in the array. Both types of arrays are useful, and both have advantages and disadvantages.

Fixed Length Arrays

In the following example, a single-dimension array containing 11 elements is declared:

```
Dim arrArray(10)
```

Although the number shown in the parentheses is 10, all arrays in **VBScript** are zero-based, so this array actually contains 11 elements.
In a zero-based array, the number of array elements is always the number shown in parentheses plus one.
This kind of array is called a fixed-size array.

Dynamic Arrays

The second type of array you can create is the dynamic array. The benefit of a dynamic array is that if you don't know how large the array will be when you write the code, you can create code that sets or changes the size while the **VBScript** code is running. A dynamic array is created in the same way as a fixed array, but you don't put any bounds in the declaration. As a result, your statement becomes

```
Dim arrNames()
```

Eventually, you need to tell **VBScript** how many elements the array will contain. You can do this with the **ReDim** function**. ReDim** tells VBScript to "re-dimension" the array to however many elements you specify**. ReDim** takes dimensions the same way **Dim** can. The syntax is

```
ReDim arrName(nCount - 1)
```

So, if you enter

```
ReDim arrNames(9)
```

You will create an array that has room to store ten elements. This way, you can set the size of the array while the code is running rather than when you write the code. This can be useful when the user gets to decide how many names he will enter**.**

Resizing a dynamic array without Preserve.

To use a dynamic array, you must subsequently use **ReDim** to determine the number of dimensions and the size of each dimension.

```
Dim arrMyArray()
ReDim arrMyArray(25)
Redim arrArray2(10,10,10)
```

Declares dynamic-array variables, and allocates or reallocates storage space at procedure level.
The **ReDim** statement is used to size or resize a dynamic array that has already been formally declared using a **Private**, **Public**, or **Dim** statement with empty parentheses (without dimension subscripts). You can use the **ReDim** statement repeatedly to change the number of elements and dimensions in an array.

Resizing a dynamic array with Preserve

The **Preserve** keyword is very important when using **ReDim.** Suppose, for example, that you create a dynamic array, specifying its storage space by using **ReDim,** fill it with data, and then later decide to make it larger so you can fill it with more information without losing your original data.
In the following example**, ReDim** sets the initial size of the dynamic array to 25. A subsequent **ReDim** statement resizes the array to 30, but uses the **Preserve** keyword to preserve the contents of the array as the resizing takes place.

```
ReDim MyArray(25)
 . . .
ReDim Preserve MyArray(30)
```

There is no limit to the number of times you can resize a dynamic array, although if you make an array smaller, you lose the data in the eliminated elements.
Figure 3 describes the memory allocations, when using arrays.

```
Dim arr(5)
```

the system allocates 5 cells in the memory to store the array. This type of declaration is static; the array size cannot be changed.

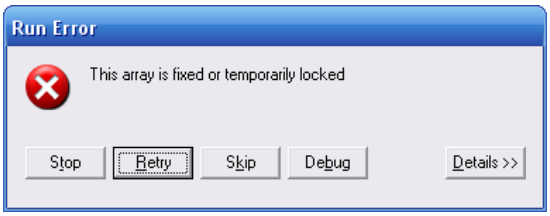

Figure 2 – Resizing static arrays

Resize a dynamic array, every time you resize an array without using the keyword Preserve, the system allocates a new place in the memory with the required size, and the last information will be lost.

Using the keyword preserve, the size of the array will be extended, and the information will not be lost

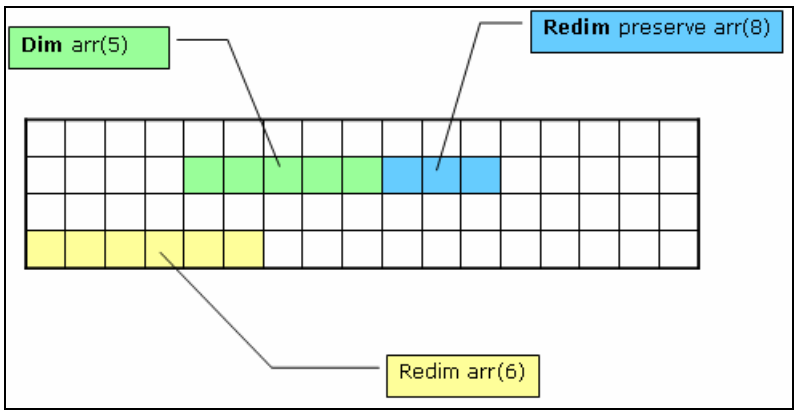

Figure 3 - Memory diagram

Array Iterations

- You can iterate an array using the **For…Next** statement. The loop is based on the array index
- You can iterate an array using the **For Each…Next** statement.
- More information about loops in Flow Control

VBScript Constants

Sometimes in writing code, you will want to refer to values that never change**.** The values for **True** and **False,** for example, are always -1 and 0, respectively. Values that never change usually have some special meaning, such as defining **True** and **False.** These values that never change are called constants. The constants **True** and **False** are sometimes referred to as implicit constants because you do not need to do anything to reference their constant names. They are immediately available in any code you write.

Working with constants

A constant is a variable within a program that never changes in value. Users can

create their own constants by initializing variables accordingly and then not changing their value. **VBScript** defines the special **True** and **False** constants, as well.

When you create the simulated constant, name the constant with uppercase letters if the constant represents a value unique to your program. If the constant is a special value expected by **VBScript**, prefix the constant name with the letters vb. This naming convention is not required, but it will help you distinguish the constant from all the other variables.
See Constant Naming Conventions on page 45

Declaring constants

Constants perform a similar function to variables, in that they are a symbolic replacement for an item of data in memory. The difference is that a constant keeps the same value throughout its lifetime.

Constants perform a similar function to variables, in that they are a symbolic replacement for an item of data in memory. The difference is that a constant keeps the same value throughout its lifetime.

Values are assigned to constants using the same method used for variables, and can contain the same range of data subtypes. In most respects, therefore, a constant is the same as a variable. In fact, it could be described as a variable whose value doesn't vary! But because confusion can arise about whether you are dealing with a constant or a variable within the script, it is safest to use a different naming convention for constants. The accepted method of denoting a constant is to use all capitals for the name. In this case, the use of underscores improves their readability and is highly recommended.

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 TIMEOUT = 54
Const DUE_DATE = #6-1-97#
Const MY_STRING_CONSTANT = "Hello World"
```

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.

Remark

Like the constant declaration in **VB,** Const in **VBScript** cannot be used to assign non-constant values or the values returned by **VBScript** functions. This means that a statement like the following:

```
Const LONG_INT_LEN = Len(lNum)   ' Invalid
```

Is also invalid, since it attempts to assign the value of a variable to a constant. It also means that a statement like:

```
Const LONG_INT_LEN = 4 + x  ' Invalid
```

Is also invalid, since it relies on the value returned by **the VBScript Len** function. Finally, unlike **VB** or **VBA,** you are not allowed to use any value which includes an operator in defining a constant. For example, the following declaration, which is valid in **VB**, generates a syntax error in **VBScript:**

```
Const ADDED_CONST = 4 + 1    ' Invalid
```

Is also invalid, since it relies on the value returned by **the VBScript Len** function. Finally, unlike **VB** or **VBA**, you are not allowed to use any value which includes an operator in defining a constant. For example, the following declaration, which is valid in **VB**, generates a syntax error in **VBScript**:

Remark

Also is possible to declare non decimal values for Constants and variables specially hexadecimal values, that are so used in programming languages, like:

```
Const MY_HEXA_CONST = &H02FF
```

The statement is legal in **VBScript,** also **QuickTest**, at run-time will transfer the value to 767. but there is a syntax checking bug in **QuickTest** that will display the follow message:

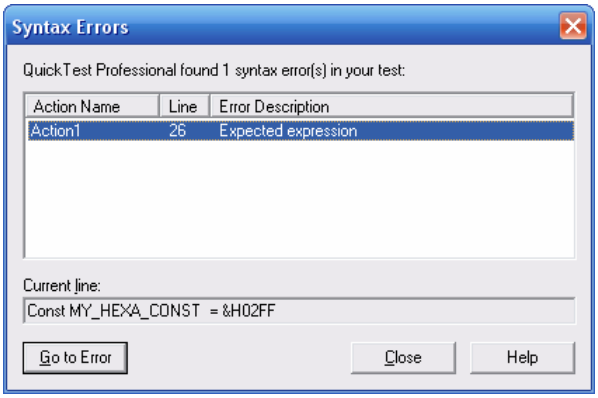

Figure 4 - QuickTest Syntax Error bug

VBScript pre-defined constants

A number of useful constants you can use in your code are built into **VBScript**. Constants provide a convenient way to use specific values without actually having to remember the value itself. Using constants also makes your code more maintainable should the value of any constant ever change. Because these constants are already defined in **VBScript**, you don't need to explicitly declare them in your code. Simply use them in place of the values they represent.

Here are the various categories of constants provided in VBScript and a brief description of each:

- Color Constants - Defines eight basic colors that can be used in scripting.
- Date and Time Constants - Defines date and time constants used by various date and time functions.
- Date Format Constants - Defines constants used to format dates and times.
- Miscellaneous Constants - Defines constants that don't conveniently fit into any other category.
- MsgBox Constants - Defines constants used in the **MsgBox** function to describe button visibility, labeling, behavior, and return values.
- String Constants - Defines a variety of non-printable characters used in string manipulation.
- Tristate Constants - Defines constants used with functions that format numbers.
- VarType Constants - Defines the various Variant subtypes.

Note

A complete listing of the constant values shown above can be found in **VBScript** Constants topic, on the **QuickTest Professional** Help.

More information about **VBScript** can be found in **Error! Reference source not found.**

The Scope and Lifetime of a Variable

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.

Private Statement

Description

The **Private** statement declares private variables and allocates storage space. Declares, in a **Class** block, a private variable**.**

Syntax

```
Private varname[([subscripts])][, varname[([subscripts])]] . . .
```

Arguments

Parameter	Description
varname	Name of the variable; follows standard variable naming conventions.
subcripts	An array and optionally specifies the number and extent of array dimensions up to 60 multiple dimensions may be declared.

Notes

- A Private variable's visibility is limited to the script in which it's created for global variables and to the class in which it is declared for class-level variables. Elsewhere, the **Private** keyword generates an error.
- varname follows standard **VB** naming conventions. It must begin with an alphabetic character, can't contain embedded periods or spaces, can't be the same as a **VBScript** reserved word, must be shorter than 255 characters, and must be unique within its scope.
- You can override standard variable naming conventions by placing your variable name in brackets. This allows you to use reserved words or illegal characters in variable names. For example:

```
Private [me]
Private [1Var]
Private [2-Var]
Private [Left]
```

- Using the subscripts argument, you can declare up to 60 multiple dimensions for the array**.**
- If the subscripts argument isn't used (i.e., the variable name is followed by empty parentheses), the array is declared dynamic. You can change both the number of dimensions and the number of elements of a dynamic array using the **ReDim** statement**.**
- In **QuickTest,** the **Public/Private** Scopes are ignored. Any variable you declare inside a reusable action will remain **Private.**

Tips

- All variables created at procedure level (that is, in code within a **Sub...End Sub, Function...End Function,** or **Property**...**End Property** construct are local by default. That is, they don't have scope outside the procedure in which they are created. The use of the **Private** keyword in these cases generates a runtime error.
- You cannot change the dimensions of arrays that were defined to be dynamic arrays while preserving their original data.
- It's good practice to always use **Option Explicit** at the beginning of a module to prevent misnamed variables from causing hard-to-find errors.

Public Statement

Description

The **Public** statement declares public variables and allocates **storage** space. Declares, in a **Class** block, a private variable.

Syntax

```
Public varname[([subscripts])][, varname[([subscripts])]] . . .
```

Arguments

Parameter	Description
varname	Name of the variable; follows standard variable naming conventions.
subcripts	An array and optionally specifies the number and extent of array dimensions up to 60 multiple dimensions may be declared.

Notes

- The behavior of a **Public** variable depends on where it's declared, as the following table shows

Variable declared in...	Scope
Any procedure, Function or Property statement	Illegal; generates a syntax error; use the Dim statement instead.
Global code (in external vbs file)	Variable is available throughout the script.
Global code (in a reusable action)	Variable is available throughout the Reusable action only (Private).
Class block declarations section	Variable is available as a property of the class to all code within the script.

- *varname* follows standard VB naming conventions. It must begin with an alphabetic character, can't contain embedded periods or spaces, can't be the same as a VBScript reserved word, must be shorter than 255 characters, and must be unique within its scope.
- You can override standard variable naming conventions by placing your variable name in brackets. This allows you to use reserved words or illegal characters in variable names. For example:

```
Public [me]
Public [1Var]
Public [2-Var]
Public [Left]
```

- Using the subscripts argument, you can declare up to 60 multiple dimensions for the array.
- If the subscripts argument isn't used (i.e., the variable name is followed by empty parentheses), the array is declared dynamic. You can change both the number of dimensions and the number of elements of a dynamic array using the **ReDim** statement**.**
- In **QuickTest, the Public/Private** Scopes are ignored. Any variable **you** declare inside a reusable action will remain **Private.**

Tips

- Instead of declaring a variable as **Public** within a class construct, you
- should create **Property Let** and **Property Get** procedures that assign and retrieve the value of a private variable, respectively.
- You cannot change the dimensions of arrays that were defined to be dynamic arrays while preserving their original data.
- It's good practice to always use **Option Explicit** at the beginning of a module to prevent misnamed variables from causing hard-to-find errors.
- To use a global variable, which is not recommended, you can use a **Public** *MyVar* statement on an external **vbs** file.
  To use global variables in your testing suite, use the Environment object, or/and the global datasheet or a global dictionary pre-defined in the repository. For more information see:

- *QuickTest Professional User's Guide > Creating Tests or Components > Working with Actions > Sharing Action Information > Sharing Values Using the Dictionary Object*

VBScript Operators

- Arithmetic Operators
  Operators used to perform mathematical calculations.
- Assignment Operator
  Operator used to assign a value to a property or variable.
- Comparison Operators
  Operators used to perform comparisons.
- Concatenation Operators
  Operators used to combine strings.
- Logical Operators
  Operators used to perform logical operations.

Working with Operators

As you begin to write **VBScript** code, you will use operators so much that their use will become natural to you. In this section, you will learn about each of the operators available to you in **VBScript**, as well as how you might go about using them.

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 Operators

The first major class of operators is arithmetic operators. Arithmetic operators enable you to perform simple arithmetic on one or more variables. Most of these operators will be familiar to you because you have been exposed to them in everyday life. Few people will be surprised to find, for example, that the + operator performs addition! Some operators, however, might be new to you. In any case, you need to understand how to apply these operators to variables and literals in **VBScript** code.

Operators fit into three separate categories: arithmetic operators, comparison operators, and logical operators. Each of these categories has a special use in a VBScript program. Operators are executed in a specific order when they are combined. Programmers must take this order into account when they write code.

Description	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Integer Division	\
Exponentiation	^
Modulus arithmetic	Mod
Unary negation	-
String concatenation	&

Table 1 - Arithmetic operators

Addition (+)

The first arithmetic operator is the addition operator. You already used this operator yesterday and probably intuitively understood its purpose because it is so commonly used and easy to understand. The addition operator is used to add values, whether they are stored in variables, constants, or literal numbers. You also use the + operator to concatenate strings.

Subtraction (-)

This operator works the same way the addition operator does except that it subtracts one or more numbers rather than add them. Otherwise, the syntax is the same.

Multiplication (*)

Addition and subtraction are important, but you also need to be able to multiply values together. In most computer languages, the * symbol is used to indicate multiplication, not the x symbol.

Division (/) and Integer Division (\)

The division operator is the last of the four commonly used arithmetic operators. Among the common arithmetic operators, division is the most complicated arithmetic operation a computer performs.
VBScript has two types of division operators. The first operator handles numbers with decimal points. Usually referred to as the floating-point division operator, it's represented by the / symbol in code listings.
The floating-point division operator is designed to divide values with decimal points, but it can also divide numbers without decimals.

Exponentiation (^)

Raises a number to the power of an exponent.

Modulus Arithmetic (Mod)

The Mod operator is another powerful arithmetic operator. Essentially, the Mod operator returns the remainder after dividing one number into another.

Unary Negation (-)

The last of the arithmetic operators is the negation operator. Simply put, this operator changes the sign of a value contained in a variable or creates a negative number.

String Concatenation (&)(+)

Forces string concatenation of two expressions.
You use the string concatenation operator to merge two strings together.

Comparison Operators

The first set of operators **VBScript** provides are arithmetic operators. This section discusses the second type: comparison operators. As the name implies, you use comparison operators to compare one or more variables, numbers, constants, or a combination of the three. **VBScript** has many different types of comparison operators, and each check for a different comparison condition.

Description	Symbol
Equality	=
Inequality	<>
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=

Object equivalence	Is

Table 2 - Comparison Operators

Equality (=)

You use the equality operator to see if a variable, constant, or number is equal to another. It's common to mistake the equality operator for the assignment operator, which is also represented by an equal sign. You use the assignment operator to set a variable equal to another variable, number, string, constant, or other data entity. For comparing strings use the **StrComp** function.

Inequality (<>)

Another important comparison operator is the inequality operator. You use this operator to test whether a variable is not equal to another variable or some data element. For comparing strings use the **StrComp** function.

Less Than (<) and Greater Than (>)

You might have a condition where you don't care whether a variable is equal to another, but you do want to know whether it is greater than or less than another variable, number, or constant. In such a case, you need the greater-than and less-than operators. For comparing strings use the **StrComp** function.

Less than or equal (<=), greater than or equal to (>=)

Sometimes, you also might want to see whether a variable is greater than *or equal to* some other variable, constant, or number. Perhaps you want to know if it is *less than or equal to* the entity. Then, you can combine operators to use the less-than-or-equal and greater-than-or-equal operators, <= and >=.
For comparing strings use the **StrComp** function.

Object Equivalence (Is)

The last comparison operator is designed for objects.
This operator does not compare one object to another, nor does it compare values. This special operator simply checks to see if the two object references in the expression refer to the same object. Suppose, for example, you have assigned a command button in your script. You have another variable, *myObject* that is set to reference different objects at different points in your program. Assume that a statement has been carried out that assigns the variable *myObject* to reference this button

Logical Operators

The last category of operators in **VBScript** is logical operators. The logical operators might require a more significant amount of understanding to appreciate. In some cases, the way you use logical operators seems to run counter to your

intuitive thinking. If you've ever taken a course in logic, you have first-hand experience with this.

Because logical operators are such an important part of **VBScript** in particular, and every programming language in general, it's important to gain a good understanding, starting with the basics, so that you can use them effectively.

Description	Symbol
Logical negation	Not
Logical conjunction	And
Logical disjunction	Or
Logical exclusion	Xor
Logical equivalence	Eqv
Logical implication	Imp

Table 3 - Logical Operators

Logical Negation (Not)

The operator performs logical negation on an expression. In addition, the Not operator inverts the bit values of any variable and sets the corresponding bit This operator has the following results

If Expression is	Then result is
True	False
False	True
Null	Null
Bit Expression is	**Bit Result is**
0	1
1	0

Table 4 - Logical Negation

Logical Conjunction (And)

The operator performs a logical conjunction on two expressions.
In order to obtain **True** in the result variable, both *expression1* and *expression2* must be **True**. You often use this operator to make sure two or more conditions are true before performing some action.

If Expression1 is	And Expression2 is	Then result is
True	True	True
True	False	False
False	True	False
False	False	False
If bit in expression1 is	**and bit in expression2 is**	**Then result is**

0	0	0
0	1	0
1	0	0
1	1	1

Table 5 - Logical conjuction

Logical Disjunction (Or)

The operator performs a logical disjunction on two expressions.
If either or both expressions evaluate to **True**, *result* is **True**.
The **Or** operator also performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result.*

If Expression1 is	**And Expression2 is**	**Then result is**
True	True	True
True	False	True
False	True	True
False	False	False
If bit in expression1 is	**and bit in expression2 is**	**Then result is**
0	0	0
0	1	1
1	0	1
1	1	1

Table 6 – Logical Disjunction

Logical Exclusion (Xor)

The operator performs a logical exclusion on two expressions.
The exclusion operator is another in the family of logical operators that you can use to make decisions based on the contents of one or more expressions. It checks whether one and only one condition is exclusively **True**.

If Expression1 is	**And Expression2 is**	**Then result is**
True	True	False
True	False	True
False	True	True
False	False	False
If bit in expression1 is	**and bit in expression2 is**	**Then result is**
0	0	0
0	1	1
1	0	1

1	1	0

Table 7 - Logical exclusion

Logical Equivalence (Eqv)

The operator performs a logical equivalence on two expressions.
The equivalence operator checks whether two expressions are bitwise equivalent to each other.
If either expression is **Null**, *result* is also **Null**. When neither expression is **Null**

If Expression1 is	And Expression2 is	Then result is
True	True	True
True	False	False
False	True	False
False	False	True
If bit in expression1 is	**and bit in expression2 is**	**Then result is**
0	0	1
0	1	0
1	0	0
1	1	1

Table 8 - Logical equivalence

Logical Implication (Imp)

The operator performs a logical implication on two expressions.

If Expression1 is	And Expression2 is	Then result is
True	True	True
True	False	True
False	True	False
False	False	True
If bit in expression1 is	**and bit in expression2 is**	**Then result is**
0	0	1
0	1	1
1	0	0
1	1	1

Table 9 – Logical Implication

Like all the logical operators covered here, the **Imp** and **Eqv** comparisons are performed on a bit-by-bit basis and the results are set on a bit-by-bit basis. For the other operators, the functionality it provides and the expected results are intuitively obvious. For **Imp** and **Eqv**, the results are less intuitive. Of course, any

expression is ultimately represented in bits on a computer, and to understand these operators, you must think in terms of these bits.

Flow Control

Controlling the Flow of VBScript Code

Both variables and operators are fundamental building blocks you need to understand to write useful code. This subject is very important when you want your programs to make on the spot decisions or execute differently based on what the user wants to do.

You will learn all the **VBScript** control structures and see several examples of how they can be applied. You will also learn which structures are more applicable and more useful under which situations. You'll learn not only how to control the flow of code, but also how best to do so.
Once you've learned how to control the flow of code within a procedure, you'll see how you can control the flow of code within the entire application.

Why Control the Flow of Code?

Before you begin learning how to control the flow of your code, you might be wondering why you should do so in the first place. What is the "flow of code" anyway? To get this answer, you need to step back for a moment and take a look at the big picture.

Stripped of all the complex software languages and the big technical words used to describe application development, a computer program is really quite simple. A program is simply a list of instructions that a computer executes. Although we tend to give computers a lot of credit, a computer is actually quite stupid. It can only do exactly what you tell it to do; it has only a limited subset of the capabilities of the human mind without many important traits, such as the ability to draw inferences, use intuition, or use emotion to help make decisions.

When a computer executes a list of instructions, it executes them one at a time in the order it is told. Depending on the type of computer and the language of the program, some computers can jump around a list of instructions if told to do so.

At times, you might want to execute the same instructions over again. Suppose, for instance, you have to ask the user for a set of data again because he didn't enter the data correctly the first time. Other times, you might need to make a decision in your program to execute one line of code under one condition and another line of code in some other condition.

In both cases, depending on the user, your code can execute in a different order each time. As you can see, it is very important, if not fundamental, that you have this capability in your programs. That's why knowing how to control the flow of your code is important

Using Control Structures to Make Decisions

VBScript gives you a variety of ways to direct the flow of your code. The mechanisms used to accomplish this in **VBScript** are called control structures. They are called structures because you construct your code around them, much like you build and finish a house around its structure. Control structures are like the wood beams and boards in your house that all of your rooms are built upon. You can use each control structure to make your code travel in different ways, depending on how you want the decision to be made. In this section, you will learn about the two control structures used in **VBScript** to make decisions. Later, you will see the structures used to make code repeat based on criteria you specify.

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

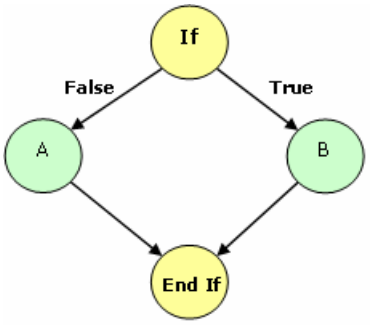

Figure 5 - If..Then.Else.End If flow

Description

Executes a statement or block of statements based on the Boolean (**True** or **False**) value of an expression.

Syntax

```
If condition Then statements [Else elsestatements ]
```

```
If condition Then
    [statements]
[ElseIf condition-n Then
    [elseifstatements] ...
[Else
    [elsestatements]]
End If
```

Arguments

Argument	Description
condition	An expression returning either **True** or **False** or an object type.
statements	One or more statements separated by colons; executed if condition is **True**.

condition-n	Same as condition.
elseifstatements	One or more statements executed if the associated *condition-n* is **True**.
elsestatements	One or more statements executed if no previous condition or condition-n expression is **True**.

If...Then

The first control structure you should know about is **If...Then**. The syntax for this control structure is given as

```
If condition = True Then
    ... the code that executes if the condition is satisfied
End If
```

Where *condition* is some test you want to apply to the conditional structure. If the condition is **true**, the code within the **If** and **End If** statements is executed. If the condition is not **true**, the code within these statements is skipped over and does not get executed.

Suppose you have a **Boolean** variable named *bShowDetails*. This variable is set to **True** at some point in your code if the user wants to see the specific details of a calculation. You could set up a simple conditional statement that gives the user the help he or she needs by entering:

```
If bShowDetail Then MsgBox "More details. . . "
```

This way, if the user doesn't want to see the details and the variable hasn't been set previously, the code in between the two statements is ignored. The condition expression is typically a test, such as whether one variable is equal to another or whether a variable equals a certain value. The condition always comes out either **True** or **False**, but the conditional structure only executes the code within it if the condition is **True**. When using an **If...Then** structure, make sure your condition is expressed properly.

- To run only one statement when a condition is **True**, use the single-line syntax for the **If...Then...Else** statement.
- 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:

```
sMsg = "Either you're not born yet or you're getting too old for this stuff!"
If nAge <= 0 Or nAge > 120 Then
   MsgBox sMsg
   bFail = True
End If
```

If...Then...Else

The **If...Then** structure is quite useful, but it has one limitation. Oftentimes, when people make decisions, they want to do one thing if a condition is true; otherwise, they want to do something different.

For example, you may have imagined a simple decision structure in your mind that if your favorite restaurant is opens; you will go there to eat. Otherwise, you will go

home and cook your own meal. You're certain to carry out either one of the two options because you're hungry, it's time for dinner, and hey-you deserve it after an afternoon of programming.

Apply this decision process to some code. Assume you had a variable that was previously set to indicate the state of whether your favorite restaurant is open or closed based on the time a script is run. You want to have your code check this variable and put up a message box to let you know whether you can hit your favorite restaurant. In this case, you wouldn't want to use the logic.

```
If MyFavoriteRestaurantOpen = True Then
   Msgbox "Go To My Favorite Restaurant!"
End If
```

Because that leaves out the alternative. You could use two statements:

```
 If MyFavoriteRestaurantOpen = True Then
   Msgbox "Go To My Favorite Restaurant!"
End If
If MyFavoriteRestaurantOpen = False Then
   Msgbox "Go Home and Cook!"
End If
```

But wouldn't it be nice if you didn't have to repeat the test where you check the negative instead of positive condition? Fortunately, you have another control structure available to you in **VBScript** that makes this process easier. The control structure, called **If...Then...Else**, is represented as:

```
If condition = True Then
    ...this is the code that executes if the condition is satisfied
Else
    ...this is the code that executes if the condition is not satisfied
End If
```

You could enter the expression you've formed in your mind as you drive toward My Favorite Restaurant as

```
If MyFavoriteRestaurantOpen = True Then
   Msgbox "Go To My Favorite Restaurant!"
Else
   Msgbox "Go Home and Cook!"
End If
```

This is certainly much simpler to understand, and it's equally helpful when writing your programs.

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:

```
If condition1 = True Then
```

```
   ...the code that executes for condition1
ElseIf condition2 = True Then
   ...the code that executes for condition2
ElseIf condition3 = True Then
   ...the code that executes for condition3
End If
```

A few comments are in order about this conditional structure. Notice that only one of the conditions can be **true**. If you want to execute code for more than one of these conditions, you cannot use this control structure. In other words, this structure can only execute one of the conditions. Once it finds the first condition in the list that is true, it executes the code off that branch and ignores all the rest.

This would work fine as long as either all the variables are **true** or **False**. What if the user didn't specify which one and both variables were set to **False**? In that case, neither would get executed and the user wouldn't see anything. You have to be careful how you use these structures. If the logic of your problems demands that at least some action take place for a given condition, either make sure one of the two variables is set to true before you perform these tests, or be sure to provide an **Else** conditional at the end. The **Else** at the end will executed if none of the other conditions is true. The **Else** condition acts as a fallback position just in case none of the other conditions is **True**. This might be valuable to you in cases where you want to make sure something happens in the conditional structure. After all, when you write code, it's best to take all possibilities into consideration; you never know when a pesky bug might enter or some unforeseen condition might take place, and you wouldn't want it to mess up your code.

```
If condition1 = True Then
   ...the code that executes for condition1
ElseIf condition2 = True Then
   ...the code that executes for condition2
ElseIf condition3 = True Then
   ...the code that executes for condition3
Else
   ...none condition match
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.
The following example shows a simple sampling of the **If...Then...Else** control structure. Here, you get a variable that shows a user age, just like the first example. Only this time, rather than perform one check, this code tests a variety of conditions and responds differently to each one.

```
If nAge = 0 Then
   MsgBox "Welcome to the human race!"
ElseIf nAge < 0 Then
    MsgBox "You have to grow up a bit before you start using VBScript!"
ElseIf nAge > 0 And Age < 10 Then
   MsgBox "If you're bold enough, you must be old enough."
ElseIf nAge > 120 Then
   MsgBox "You're getting too old for this stuff!"
```

```
Else
    MsgBox "You're at the perfect age to get started!"
End If
```

In this case, you see that you can do more specific checks for various age groups and even provide an **Else** clause for the default case. You couldn't do that using **If...Then** statements unless you want to write a lot of excess code.

Select Case

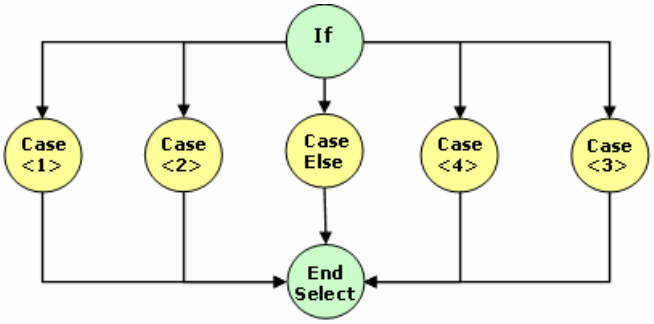

Figure 6 - Select Case Flow

Description

Allows for conditional execution of a block of code, typically out of three or more code blocks, based on some condition. Use the **Select Case** statement as an alternative to complex nested **If...Then...Else** statements.

Syntax

```
Select Case testexpression
   [Case expressionlist-n
      [statements-n]] ...
   [Case Else
      [elsestatements]]
End Select
```

Arguments

Argument	Description
testexpression	Any numeric or string expression.
epressionlist-n	Required if **Case** appears. Delimited list of one or more expressions.
satements-n	One or more statements executed if *testexpression* matches any part of *expressionlist-n*.
elsestatements	One or more statements executed if *testexpression* doesn't match any of the **Case** clauses.

Notes

- Any number of **Case** clauses can be included in the **Select Case** statement.
- If a match between *testexpression* and any part of *expressionlist* is found, the program statements following the matched *expressionlist* are executed. When program execution encounters the next **Case** clause or the **End Select** clause, execution continues with the statement immediately following the **End Select** clause.
- Both *expressionlist* and *testexpression* must be a valid expression that can consist of one or more of the following: a literal value, a variable, an arithmetic or comparison operator, or the value returned by an intrinsic or user-defined function.
- If used, the **Case Else** clause must be the last **Case** clause. Program execution encounters the **Case Else** clause, and thereby executes, the *elsestatements*—only if all other *expressionlist* comparisons fail.
- **Select Case** statements can also be nested, resulting in a successful match between *testexpression* and *expressionlist* being another **Select Case** statement.

Tips

- The **Select Case** statement is the **VBA/VBScript** equivalent of the **Switch** construct found in C and C++.
- The **Case Else** clause is optional. However, as with **If...Then...Else** statements, **it's often good practice** to provide a **Case Else** to catch the exceptional instance when, perhaps unexpectedly, a match can't be found in any of the *expressionlists* you have provided.
- If *testexpression* satisfies more than one *expressionlist* comparison, only the code in the first is executed.
- A **Select Case** structure works with a single *testexpression* that is evaluated once, at the top of the structure. The result of the *testexpression* is then compared with the values for each **Case** in the structure.
- 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. **Select Case** structure is defined as follows:

```
Select Case expression
Case exp-1
   ...this is the code that executes if exp-1 matches expression
Case exp-2, exp-3
   ...this is the code that executes if exp-2 or exp-3 matches expression
Case exp-4
   ...this is the code that executes if exp-4 matches expression
    .
    .
    .
Case Else
   ...this is the code that executes if none matches expression
End Select
```

Example

The following example uses **Select Case** to read a variable populated by the user and determine the name of the user's operating system:

```
Select Case Left(Environment.Value("OSVersion"), 1)
  Case 1 :    varOSDesc = "Windows NT"
  Case 2 :    varOSDesc = "Windows 98"
  Case 3 :    varOSDesc = "Windows 95"
  Case 4 :    varOSDesc = "Windows 3.11"
  Case 5 :    varOSDesc = "Windows 2000"
  Case 6 :    varOSDesc = "Windows ME"
  Case 7 :    varOSDesc = "Windows XP"
  Case Else : varOSDesc = "OS is unknown"
End Select
```

Using Control Structures to Make Code Repeat

On occasion, you will need to write code that repeats some set of statements. Oftentimes, this will occur when you need to perform some calculation over and over or when you have to apply the same calculations or processing to more than one variable, such as changing the values in an array. This section shows you all the control structures you can use in **VBScript** to control code in your program that repeats.

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 For...Next Statement

Description

Defines a loop that executes a given number of times, as determined by a loop counter. To use the **For...Next** loop, you must assign a numeric value to a counter variable. This counter is either incremented or decremented automatically with each iteration of the loop. In the **For** statement, you specify the value that is to be assigned to the counter initially and the maximum value the counter will reach for the block of code to be executed. The **Next** statement marks the end of the block of code that is to execute repeatedly, and also serves as a kind of flag that indicates the counter variable is to be modified.

Syntax

```
For counter = start To end [Step stepcounter]
    [statements]
    [Exit For]
    [statements]
Next
```

Arguments

Argument	Description
counter	Numeric variable used as a loop counter. The variable can't be an array element or an element of a user-defined type.
start	Initial value of counter.
end	Final value of counter.
step	Amount counter is changed each time through the loop. If not specified, step defaults to one.
statements	One or more statements between **For** and **Next** that are executed the specified number of times.

Notes

- If *start* is greater than *end*, and no *step* keyword is used or the *stepcounter* counter is positive, the **For...Next** loop is ignored and execution commences with the first line of code immediately following the **Next** statement.
- If *start* and *end* are equal and *stepcounter* is one, the loop executes once.
- *counter* can't be a variable of type **Boolean** or an array element.
- *counter* is incremented by one with each iteration unless the **Step** keyword is used.
- If the **Step** keyword is used, *step* specifies the amount *stepcounter* is incremented if *stepcounter* is positive or decremented if it's negative.
- If the **Step** keyword is used, and *stepcounter* is negative, *start* should be greater than *end*. If this isn't the case, the loop doesn't execute.
- The **For...Next** loop can contain any number of **Exit For** statements. When the **Exit For** statement is executed, program execution commences with the first line of code immediately following the **Next** statement.

```
For Each subObject In myObj
    sName = subObject.NameProperty
    If Len(Sname) < 10 Then
       Exit For
    End if
Next
```

Tips

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 as default**.**

```
For x = 1 To 50
   MyProc
Next
```

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.

```
Dim j, total
For j = 1 To 10 Step 2
   total = total + j
Next
MsgBox "The total is " & total
```

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.

```
Dim j, total
For j = 16 To 2 Step -2
   total = total + j
Next
MsgBox "The total is " & total
```

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.

For...Next loops can also be nested.

```
For nDay = 1 To 365
   For nHour = 1 To 23
      For nMinute = 1 To 59
         . . .
      Next
   Next
Next
```

- When you use a positive step value, make sure the finish value is greater than the start value, or the loop will not execute at all.

```
For i = 10 to 1 Step 2        ' Incorrect
For i = 1 to 10 Step 2        ' Correct
```

- When you use a negative step value, make sure the start value is greater than the finish value, or the loop won't execute at all.

```
For i = 1 to 10 Step -1        ' Incorrect
For i = 10 to 1 Step -1        ' Correct
```

- Never use a step value of zero. In this case, **VBScript** will enter an infinite loop, and your program might run indefinitely.

```
For i = 1 to 10 Step 0        ' Incorrect
For i = 1 to 10 Step 3        ' Correct
```

Using For Each...Next Statement

Description

Repeats a group of statements for each element in an array or an object collection.

Syntax

```
For Each element In group
    [statements]
    [Exit For]
    [statements]
Next
```

Arguments

Argument	Description
element	The *string* argument is any valid string expression. If *string* contains **Null**, **Null** is returned.
group	Name of an object collection or array.
statements	One or more statements that are executed on each item in group.

Notes

- The **For...Each** code block is executed only if group contains at least one element.
- All statements are executed for each element in group in turn until either there are no more elements in group, or the loop is exited prematurely using the **Exit For** statement. Program execution then continues with the line of code following **Next**.
- **For Each...Next** loops can be nested, but each element must be unique. For example:

```
For Each myObj In anObject
   For Each subObject In myObject
      sName(ctr) = subObject.NameProperty
      ctr = ctr + 1
   Next
Next
```

- Uses a nested **For Each...Next** loop, but two different variables, *myObj* and *subObject*, represent element.
- Any number of **Exit For** statements can be placed with the **For Each...Next** loop to allow for conditional exit of the loop prematurely. On exiting the loop, execution of the program continues with the line immediately following the **Next** statement.
- For example, the following loop terminates once the program finds a name

in the *myObj* collection that has fewer than 10 characters:

```
For Each subObject In myObj
    sName = subObject.NameProperty
    If Len(Sname) < 10 Then
       Exit For
    End if
Next
```

Tips

- Each time the loop executes when iterating the objects in a collection, an implicit **Set** statement is executed. The following code reflects the "longhand" method that is useful for explaining what is actually happening during each iteration of the **For Each...Next** loop:

```
For i = 1 to MyObject.Count
    Set myObjVar = MyObject.Item(i)
    MsgBox myObjVar.Name
Next
```

- Because the elements of an array are assigned to element by value, element is a local copy of the array element and not a reference to the array element itself. This means that you can't make changes to the array element using **For Each...Next** and expect them to be reflected in the array once the **For Each...Next** loop terminates, as demonstrated in the example shown next.

```
Dim strNameArray(1)
Dim intCtr
strNameArray(0) = "Paul"
strNameArray(1) = "Bill"
intCtr = 0
For Each varName In strNameArray
   varName = "Changed"
   Msgbox strNameArray(intCtr)
   intCtr = intCtr + 1
Next
```

- For example, on the first iteration of the loop, although *varName* has been changed from "Paul" to "Changed," the underlying array element, *strNameArray*(0), still reports a value of "Paul."
- This proves that a referential link between the underlying array and object variable isn't present; instead, the value of the array element is passed to element by value.

Using Do...Loops Statement

Description

Repeatedly executes a block of code while or until a condition becomes **True**.

Syntax

```
Do [{While | Until} condition]
   [statements]
[Exit Do]
   [statements]
Loop
```

```
Do
   [statements]
[Exit Do]
   [statements]
Loop [{While | Until} condition]
```

Arguments

Argument	Description
condition	Numeric or string expression that is **True** or **False**. If condition is **Null**, condition is treated as **False**.
statements	One or more statements that are repeated while or until condition is **True**.

Notes

- On its own, **Do...Loop** repeatedly executes the code that is contained within its boundaries indefinitely. You therefore need to specify under what conditions the loop is to stop repeating. Sometimes, this requires modifying the variable that controls loop execution within the loop. For example:

```
Do
   nCtr = nCtr + 1    ' Modify loop control variable
   MsgBox "Iteration " & nCtr & " of the Do loop..." & vbCrLf
   ' Compare to upper limit
   If nCtr = 10  Then Exit Do
Loop
```

- Failure to do this results in the creation of an endless loop.
- Adding the **Until** keyword after **Do** instructs your program to **Do** something **Until** the condition is **True**. Its syntax is:

```
Do Until condition
   code to execute
Loop
```

- If condition is **True** before your code gets to the **Do** statement, the code within the **Do...Loop** is ignored.
- Adding the **While** keyword after **Do** repeats the code while a particular condition is **True**. When the condition becomes **False**, the loop is automatically exited. The syntax of the **Do While** statement is:

```
Do While condition
   code to execute
Loop
```

- Again, the code within the **Do...Loop** construct is ignored if condition is **False** when the program arrives at the loop.
- In some cases, you may need to execute the loop at least once. You might, for example, evaluate the values held within an array and terminate the
- loop if a particular value is found. In that case, you'd need to execute the loop at least once. To do this, place the **Until** or **While** keyword along with the condition after the **Loop** statement. **Do...Loop Until** always executes the code in the loop at least once and continues to loop until the condition is **True**. Likewise, **Do...Loop While** always executes the code at least once, and continues to loop while the condition is **True**. The syntax of these two statements is as follows:

```
Do
 code to execute
Loop Until condition

Do
 code to execute
Loop While condition
```

- A **Null** condition is treated as **False**.
- Your code can exit the loop at any point by executing the **Exit Do** statement.

Using While. . .Wend Statement

Description

The **While...Wend** statement executes a series of statements as long as a given condition is **True**.

Syntax

```
While condition
   Version [statements]
Wend
```

Arguments

Argument	Description
condition	Numeric variable used as a loop counter. The variable can't be an array element or an element of a user-defined type.
statements	One or more statements between For and Next that are executed the specified number of times.

Notes

- A **Null** condition evaluated as **False**.
- If *condition* evaluates to **True**, the program code between the **While** and **Wend** statements executed. After the **Wend** statement is executed, control is passed back up to the **While** statement, where condition is evaluated again. **When** *condition* evaluates to **False**, program execution skips to the

first statement following the **Wend** statement.

- You can nest **While…Wend** loops within each other.

Tips

- The **While…Wend** statement remains in **VBScript** for backward compatibility only. It has been superseded by the much more flexible **Do…Loop** statement.

Putting all together

Now that you've seen all the decision structures at your command, you know that you have quite a wide variety of choices. You might be wondering at this point, "How do I know which control structure to use?" Oftentimes, you can structure your code where you can choose from one or more of the techniques explained today. I can't provide a specific set of rules to follow for every case. Often, it just boils down to using the structure that expresses what you want to do the most clearly. **The If…Then…Else** structure is fairly intuitive and straightforward. Still, you might want to keep the following points in mind:

- Do you want to take a single action based on a decision, or do you want to take more than one possible action based on a decision?
- If you only have one action to consider, the **If…Then** structure is best for you. If, however, you have several tests to do and actions to match each, you might want to use a series of **If…Then…ElseIf** statements. Often, it helps to write a flowchart or a graph of what you want the program to do under what circumstances.
- Sometimes, you might wonder whether you should use the **For…Next** control structure. Ask yourself the following question: Do you want the code in your loop to execute a certain number of times based on a counter that can be adjusted each time through the loop?
- If so, and this is often the case, use the **For…Next** loop. If, for instance, you have to set a series of elements within an array or perhaps perform some calculation a certain number of times, the **For…Next** loop is the structure to use. If you can't find a connection between a counter, start value, and stop value for repeating the code in your conditional, you might have to consider a **Do…Loop.**

You might hesitate when deciding what type of **Do…Loop** to use for the case in question. You should keep the following tips in mind when making your decision:

- Do you want the code within your loop to always execute at least once?
- If so, you should use either the **Do…Loop Until** or **Do…Loop While** structures.
- If you want to loop through code until something happens, such as setting a variable or completing some procedure successfully, you should use the **Do…Loop Until** or **Do Until…Loop** structures**.**
- Any time you want to repeat the code in your loop while some condition is true or while something is happening, the **Do…Loop While** or **Do While…Loop** structures are probably best.

Take some time to think of the right looping approach. At the same time, realize that there is often no one best approach. Even experienced programmers might

each choose different looping constructs to solve the same problem.

A control structure is a set of code statements you can use to enable blocks of code to repeat or to execute or not execute based on decisions made in those statements. You have seen the syntax of each control structure available in **VBScript**. With your knowledge of variables, operators, and control structures, you are becoming an increasingly competent and knowledgeable **QuickTest** programmer.

Control structures are at the heart of **VBScript** programming. A program usually has to make many decisions.

Subroutines and Function Procedures

A procedure is a grouping of code statements that can be called by an associated name to accomplish a specific task or purpose in a program. Once a procedure is defined, it can be called from different locations in the program as needed.

Functions and **procedures** (or subroutines) are central to modern programming**.** Dividing our script into subroutines helps us to maintain and write programs by segregating related code into smaller, manageable sections. It also helps to reduce the number of lines of code we have to write by allowing us to reuse the same subroutine or function many times in different situations and from different parts of the program. In this section, we'll examine the different types of subroutines, how and why they are used, and how using subroutines helps to optimize code.

In **VBScript**, there are two kinds of procedures; the **Sub** procedure and the **Function** procedure.

What Are Functions?

Functions are a way of performing a task and getting something back. For example, **VBScript** has a function named **Date**(), which simply looks up and provides the current date according to your computer's internal clock. Functions are used to perform special calculations, retrieve information, look up information, convert data types, manipulate data, and much more.

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 ().

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.**

Declaring Subroutines/Functions

There are several very straightforward rules to remember when giving names to your subroutines/functions:

- The name can contain any alphabetical or numeric characters and the underscore character.
- The name cannot start with a numeric character.
- The name cannot contain any spaces. Use the underscore character to separate words to make them easier to read.

For example:

```
Sub 123MySub( )          ' Illegal
Function My Func( )      ' Illegal
```

Both contain illegal subroutine names. However:

```
Sub MySub123( )          ' Legal
Function MyFunc( )       ' Legal
```

Both are legal **Subroutine/Functions** names.

- A subroutine/function can be scoped as **Public, Private** or **Default**
  - **Public -** indicates that the **Sub/Function** procedure is accessible to all other procedures in all scripts.
  - **Default -** Used only with the **Public** keyword in a **Class** block to indicate that the **Sub/Function** procedure is the default method for the class. An error occurs if more than one **Default** procedure is specified in a class**.**
  - **Private** - Indicates that the **Sub/Function** procedure is accessible only to other procedures in the script where it is declared.
  - If not explicitly specified using either **Public** or **Private**, **Sub** procedures are public by default.
- **Sub/Function** Subroutines can receive 0 (zero) to n argument parameters
  - The list of variables representing arguments that are passed to the **Subroutine** when it is called.
  - Commas separate multiple variables.
  - Arguments can be passed in two ways
    - **ByVal -** Indicates that the argument passed by value.
    - **ByRef** - Indicates that the argument passed by reference.

    - varname - Name of the variable representing the argument; follows standard variable naming conventions.
- You can't define a **Sub/Function** procedure inside any other procedure.
- The **Exit Sub/Exit Function** statement causes an immediate exit from a **Sub/Function** procedure. Program execution continues with the statement that follows the statement that called the **Sub/Function** procedure. Any number of **Exit Sub/Exit Function statements** can appear anywhere in a **Sub/Function procedure.**
- **Sub/Function** procedures can be recursive, that is, they can call themselves to perform a given task. However, recursion can lead to stack overflow.
- Variables used in **Sub/Function** procedures fall into two categories
    - Those that are explicitly declared within the procedure and those that not.
    - Variables that are explicitly declared in a procedure are always local to the procedure.
    - Variables that are used but not explicitly declared in a procedure are also local, unless they are explicitly declared at some higher level outside the procedure.
- A procedure can use a variable that is not explicitly declared in the procedure, but a naming conflict can occur if anything you have defined at the script level has the same name. To avoid this kind of conflict, use an **Option Explicit** statement to force explicit declaration of variables.
- **VBScript** may rearrange arithmetic expressions to increase internal efficiency. Avoid using a **Function** procedure in an arithmetic expression when the function changes the value of variables in the same expression.
- To return a value from a **Function** subroutine, assign the value to the function name.

Calling a Subroutine

Now that you've learned how to create a subroutine, how do you call one? You can call a subroutine throughout the rest of the application once you've declared and created it. You can call subroutines by using the Call keyword or just entering the name of the subroutine on a line of code. For example, to call a subroutine called *ShowMessage*, you could enter

```
ShowMessage "This is the message."
```

You could also use the Call keyword and enter

```
Call ShowMessage("This is the message.")
```

Notice that in the first method, you do not place parentheses around the arguments of the subroutine. On the other hand, if you use Call, you must enclose the arguments in parentheses. This is simply a convention that **VBScript** requires. What if a subroutine has no arguments? To call the subroutine ShowAboutMessage, you could enter

```
ShowAboutMessage
```

Or

```
Call ShowAboutMessage()
```

Or you could use

```
Call ShowAboutMessage
```

The first method simply lists the name of the subroutine. The second method uses **Call** but doesn't require parentheses because the subroutine has no arguments. Whether you use the parentheses when you call or declare a subroutine with no arguments is a personal preference about writing code. When you call a subroutine without the **Call** statement, it can be more difficult to figure out the difference between a subroutine and a variable in your code, especially if your code is lengthy. Although the choice is up to you, it is generally recommended that you always use the **Call** statement when calling subroutines for the sake of readability.

Call Statement

Description

The **Call** statement passes program control to an explicitly named procedure or function.

Syntax

```
[Call] procedurename [argumentlist]
```

Arguments

Parameter	Description
Call	Required. The name of the subroutine being called.
argumentlist	Optional. A comma-delimited list of arguments to pass to the subroutine being called.

Notes

- The **Call** statement requires that the procedure being called be named explicitly.
- You cannot assign the subroutine name to a variable and provide that as an argument to the **Call** statement. For example, the following is an illegal use of Call:

```
Dim sProc
sProc = "PrintRoutine"
Call sProc(sReport)       ' Illegal: sProc is a variable
```

- The following code fragment shows a valid use of the Call statement:

```
Call PrintRoutine(sReport)          ' Legal usage
```

- You aren't required to use the **Call** keyword when calling a function procedure. However, if you use the **Call** keyword to call a procedure that requires arguments, *argumentlist* must be enclosed in parentheses. If you omit the **Call** keyword from the procedure call, you must also omit the parentheses around *argumentlist*.

Tips

- You can use the **Call** keyword to call a function when you're not interested in the function's return value.
- The use of the **Call** keyword is considered outdated. We suggest not using the keyword, as it is unnecessary and provides no value.
- If you remove the **Call** statement but fail to remove the parentheses from a call to a subroutine with a single argument, then that argument is passed by value rather than by reference. This can have unintended consequences.

Calling a Function

Now that you've seen how to declare a function, you need to know how to call it. The benefit of using a function is that you can pass back a piece of data to the caller. The subroutine does not enable you to do this because it does not return anything. You will see a way to change variables in the calling code with a subroutine later today, but the function is a better way to get data back and forth. To call a function, you simply use the syntax

```
return_variable = function_name(argument1, argument2, …, argumentn)
```

Notice that in this case, the syntax is quite a bit different from the subroutine. Here, you can assign the function to a variable (or another expression that can be updated with a value, such as a property, which will be covered in later lessons), or you needn't assign it to anything. The parentheses are optional only when no arguments are passed to the function.

For an example of its use, suppose you have a function called `GetAge`. To use the `GetAge` function, you could enter the statement

```
UserAge = GetAge()
```

Notice that this function doesn't need any arguments, and the result is assigned to a variable named UserAge. The following function requires three arguments-hours, minutes, and seconds-and returns the number of seconds:

```
Function GetSeconds(Hrs, Min, Sec)
   GetSeconds = Hrs * 3600 + Min * 60 + Sec
End Function
```

You could then call this function using a statement like

```
NumSeconds = GetSeconds(2, 34, 25)
```

Where, the total number of seconds is returned to the variable NumSeconds. The statement

```
Call GetSeconds(2, 34, 25)
```

Would also be valid, but it wouldn't be very useful because you're not retrieving the number of seconds from the function! This simply calls a function as if it were a subroutine, without handling the return value. You can also utilize a function within an expression, such as

```
MsgBox "There are " & GetSeconds(2,34,25) & " seconds."
```

You don't need to assign a variable to the return of the function because the return value is automatically used within the statement. Although this is certainly legal, it is not always the best programming practice. If you want to use the result of the function more than once, you must store the result in a variable. Otherwise, you will have to call the function again and waste the computer's resources in doing the calculation all over again. Likewise, storing the value in a variable to avoid repeated calls makes the code more readable and maintainable.

Exiting a Subroutine/Function

The code within your subroutine/function will execute until one of two things happens. First, the subroutine/function might get down to the last line, the End **Sub/End Function** line, which terminates the subroutine and passes the baton back to the caller. This statement can appear only once at the end of the subroutine declaration. The second possibility is that **VBScript** could execute the following code **statement:**

```
Exit Sub (for subroutine)
Exit Function (for functions)
```

When, placed inside the subroutine/function. You might use this statement if you need to provide more than one exit point for the subroutine/function. However, you shouldn't need to use this very often if your subroutine is constructed properly. Consider the following function:

```
Public Function ConvertFeetToInches(ByVal nFeet)
   If nFeet < 0 Then
      Exit Function
   Else
      ConvertFeetToInches = nFeet * 12
   End If
End Function
```

This function contains an **Exit Function** statement, which could be avoided by changing the Function to:

```
Public Function ConvertFeetToInches(ByVal nFeet)
   If nFeet >= 0 Then
      ConvertFeetToInches = nFeet * 12
   End If
End Function
```

Passing Arguments into Procedures

Procedures are an essential part of almost every program. When you define a procedure, whether it's a **Function** or a **Sub** procedure, you need to decide whether the procedure arguments are passed by reference or by value.

What difference does it make?

VBScript default is to pass arguments by reference. You can include the **ByRef** keyword in an argument list if desired but, because this is the default, it has no effect:

```
Sub Foo(ByRef Arg1, ByRef Arg2)
```

The procedure is passed the address of the argument variable (in other words, a reference to the variable):

```
Dim Total
Call MySub(Total)
```

In this example, *MySub* receives a reference to Total. The practical consequence of this is that code in *MySub* can change Total. Here's an example. First, the procedure:

```
Sub MySub(Total)
    Total = 50
End Sub
```

Now the code that calls the procedure

```
Dim Total : Total = 100
Call MySub(Total)
```

After this code executes, the variable Total equals 50 because the code in the procedure changed its value. To pass an argument by value, use the **ByVal** keyword

```
Sub MySub(ByVal Total)
```

When you use **ByVal**, the procedure is passed a copy of the argument variable and not a reference to the argument variable itself. Code in the procedure cannot change the variable's value**.**

```
Sub MySub(ByVal Total)
    Total = 50
End Sub
```

Now the code that calls the procedure:

```
Dim Total
Total = 100
Call MySub(Total)
```

After this code executes, Total is still equal to 100.

Note that array arguments and user-defined type arguments cannot be passed **ByVal.** Also, using **ByVal** or **ByRef** doesn't have any effect when the argument is a literal constant--only when it's a variable.

For most procedures, the default **ByRef** argument passing is fine. You can use **ByVal** when you want to ensure that code in the procedure cannot change the variable that was passed as an argument.

Why Are Procedures Useful?

If you're new to programming, you might be wondering why procedures are useful

in the first place. There are three primary reasons: readability, maintainability, and correctness

Procedures are useful any time you have a task that must be accomplished many times, perhaps in many places in your code, throughout your program. Suppose you request an order number from the user, and each time the number is entered, you want to make sure it's valid. One option is to write code that checks each time an order is entered. The following code **should be placed in a function.**

```
SpouseOrder = InputBox("What order would you like for your spouse?")
If SpouseOrder < 0 Then
   MsgBox "The order number is invalid."
End If
YourOrder = InputBox("What order would you like for yourself?")
If YourOrder < 0 Then
   MsgBox "The order number is invalid."
End If
ChildOrder = InputBox("What order would you like for your children?")
If ChildOrder < 0 Then
   MsgBox "The order number is invalid."
End If
```

As you can see from this example, the same check is repeated three times in your code, this results in code that is not only more difficult to read, but also more difficult to maintain.
Rather than type the same code three times, wouldn't it make your code more readable if you created a function and placed the repeating code within that function? Suppose you call the function *VerifyOrderNumber* and place the common code in that function.

```
Public Function VerifyOrderNumber(ByVal nOrderNumber)
    VerifyOrderNumber = False      ' initializing return value
    If IsNumeric(nOrderNumber) = False Then
      MsgBox "Not s number."
      Exit Function
    ElseIf nOrderNumber < 0 Then
       MsgBox "The order number is invalid."
    Else
       VerifyOrderNumber = True
    End If
End Function
```

Using a function to improve the program.

```
Do
   sSpouseOrder = InputBox("What order would you like for your spouse?")
Loop Until VerifyOrderNumber(sSpouseOrder) = True

Do
   sYourOrder = InputBox("What order would you like for yourself?")
Loop Until VerifyOrderNumber(sYourOrder) = True

Do
   sChildOrder = InputBox("What order would you like for your children?")
Loop Until VerifyOrderNumber(sChildOrder) = True
```

If the user enters values greater than zero, the function returns with a **Boolean** variable indicating the result is valid. Otherwise, the function returns the **Boolean** value for false, and the loop continues to prompt the user until an order number entered, calling the function again each time through.

In this case, you have placed all the repeating code within a function so that the code within the function appears just once rather than several times throughout the program. Doing this has several advantages. First of all, it's a lot easier for the reader of the code. He can see what the code does in one word rather than having to wade through all the details. Furthermore, it cuts down on the size of the code listing. Perhaps most important, it makes the code more maintainable.

Exit Statement

Description

The **Exit s**tatement exits a block of **Do...Loop**, **For...Next**, **Function**, or **Sub** code.

Syntax

```
Exit Do
Exit For
Exit Function
Exit Property
Exit Sub
```

The **Exit** statement syntax has these forms:

Statement	Description
Exit Do	Exits a **Do...Loop** statement. If the current **Do...Loop** is within a nested **Do...Loop**, execution continues with the next **Loop** statement wrapped around the current one. If, however, the **Do...Loop** is standalone, program execution continues with the first line of code after the Loop statement.
Exit For	Exits a **For...Next** loop. If the current **For...Next** is within a nested **For...Next** loop, execution continues with the next **Next** statement wrapped around the current one. If, however, the **For...Next** loop is standalone, program execution continues with the first line of code after the **Next** statement**.**
Exit Function	Immediately exits the **Function** procedure in which it appears. Execution continues with the statement following the statement that called the **Function.**
Exit Sub	Immediately exits the **Sub** procedure in which it appears. Execution continues with the statement following the statement that called the **Sub.**
Exit Property	Immediately exits the **Property** procedure in which it appears. Execution continues with the statement following the statement that called the **Property** procedure**.**

Tips

Traditional programming theory recommends one entry point and one exit point for each procedure. However, you can improve the readability of long routines by using the **Exit** statement. Using **Exit Sub** can save having to wrap almost an entire subroutine (which could be tens of lines long) within an **If...Then** statement.

With Exit Sub:

```
Sub MyTestSub(nNumber)
    If nNumber = 10 Then
        Exit Sub
    End If
    ...'code
End Sub
```

Without **Exit Sub**:

```
Sub MyTestSub(nNumber)
    If nNumber <> 10 Then
        ...'code
    End If
End Sub
```

In the case of the **Exit Function**, **Exit Property**, and **Exit Sub** statements, the point in the program to which program flow returns depends on the caller of the **Property**, **Function**, or **Sub**, respectively, and not on the **Property**, **Function**, or **Sub** itself.

Coding Conventions

Coding conventions are suggestions that may help you write code using Microsoft Visual Basic Scripting Edition. Coding conventions can include the following:

- Naming conventions for objects, variables, and procedures
- Commenting conventions
- Text formatting and indenting guidelines

The main reason for using a consistent set of coding conventions is to standardize the structure and coding style of a script or set of scripts so that you and others can easily read and understand the code.

Using good coding conventions results in precise, readable, and unambiguous source. code that is consistent with other language conventions and as intuitive as possible.

Constant Naming Conventions

Constant names should be uppercase with underscores (_) between words. For example:

```
USER_LIST_MAX
NEW_LINE
```

Variable Naming Conventions

For purposes of readability and consistency, use the prefixes listed in the following table, along with descriptive names for variables in your VBScript code.

Subtype	Prefix	Example
Boolean	b	bExist
DateTime	d	dNow
String	s	sName
Object	o	oFile
Integer, Byte, Long	n (numeric)	nCounter
Single, Double	f	fPrice
Error	err	errResponse
Array	arr	arrLabels
Currency	c	cDollar

Table 10 – Variable naming conventions

Descriptive Variable and Procedure Names

The body of a variable or procedure name should use mixed case and should be as complete as necessary to describe its purpose. In addition, procedure names should begin with a verb, such as *InitNameArray* or *CloseDialog*.

For frequently used or long terms, standard abbreviations are recommended to help keep name length reasonable. In general, variable names greater than 32 characters can be difficult to read.

Object Naming Conventions

The recommended conventions for the various objects you may encounter while programming VBScript is the prefix "o", in some cases, for known **ActiveX** objects i recommend the following prefixes:

ActiveX object	prefix
ADODB.Connection	oConn
ADODB.Recordset	oRst
Scipting.FileSystemObject	oFso
Scipting.TextStream	oTxt
Scripting.Dictionary	oDic
Shell.Application	oWsh
Excel.Application	oXls

Table 11 - Object naming conventions

Code Commenting Conventions

All procedures, actions and reusable actions, should begin with a brief comment describing what they do. This description should not describe the implementation details (how it does it) because these often change over time, resulting in unnecessary comment maintenance work, or worse yet, erroneous comments. The code itself and any necessary inline comments describe the implementation. Arguments passed to a procedure should be described when their purpose is not obvious and when the procedure expects the arguments to be in a specific range.

The follow example demonstrates a **QuickTest** action header template convention for a reusable action.

```
'TEST HEADER
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'
'@Name          : <Action name>
'@Description   : <Propose>
'@Author        : <Author Name>
'@CreationDate  : <dd-mmm-yyyy>
'@inParameter   : <inparam1:>, <type:>, <default:>, <description:>
'@inParameter   : <inparam2:>, <type:>, <default:>, <description:>
'@outParameter  : <outparam1:>, <type:>, <description:>
'@outParameter  : <outparam1:>, <type:>, <description:>
'@PreCondition  : <list of pre conditions>
'@Libraries     : <List of required libraries>
'@DataFiles     : <List of required data files>
'@DtParam       : <Name> <Description> <Format/Valid values>
'@DtParam       : <Name> <Description> <Format/Valid values>
'@DtParam       : <Name> <Description> <Format/Valid values>
'@Modification  : <By <Name>, Date: <dd-mmm-yyyy>>
'                 Description: <modification description>
'@Remarks       : <Additional information>
'
'TEST HEADER
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Option Explicit

'--- Constant Declarations
    'declare here your constants
'--- Variable Declarations
                               'declare here your variables
```

Functions and Sub Standards

Procedure Header

The importance of documentation I have described on chapter Table 11 - Object naming conventions

Code Commenting Conventions on page 46.

Here I want to show how documentation of functions can enhance the **QuickTest** auto documentation feature the keyword view. The Keyword View can contain any of the following columns: **Item**, **Operation**, **Value**, **Assignment**, **Comment**, and **Documentation.** Where the **Documentation** column is a Read-only auto-documentation of what the step does, in an easy-to-understand sentence.
For Example, I have this a simple Sum function on an external **vbs** file.

```
Function Sum(ByVal a, ByVal b)
    Sum = a+ b
End Function
```

Then, in my script I perform the follow call

```
c = Sum(2,8)
```

The keyword view will show the follow:

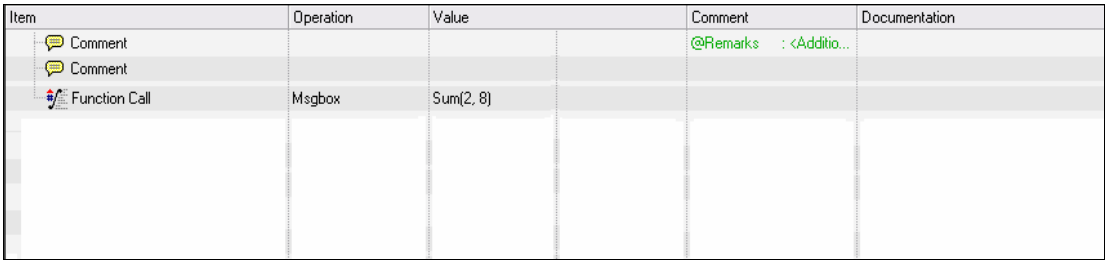

Figure 7 – Without auto documentation

```
'@Description Return the mathematical sum of two numeric values
'@Documentation Return the sum between <a> and <b>
```

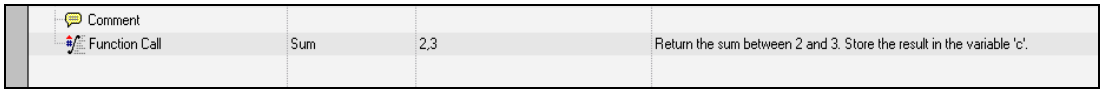

Figure 8 - Applying auto documentation

```
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'@Documentation   : <QTP auto documentation feature>
'@Description    : <Your Description>
'@ModuleName     : <module name>
'@Author         : <author name, e-mail>
'@Date           : <dd-mmm-yyyy>
'@InParameter    : <<[in/out]> Name, <type>, <description>
'@ReturnValue    : <<[in/out]> Name, <type>, <description>
'@Modification   : By <Name>, Date: <dd-mmm-yyyy>
'@ReturnValue    : <If the function success it returns <description, value>
'                   <If the function fails it returns <description, value>.
'@Remarks        : <additional info>
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
```

The QuickTest Editor Options

Fonts and Colors

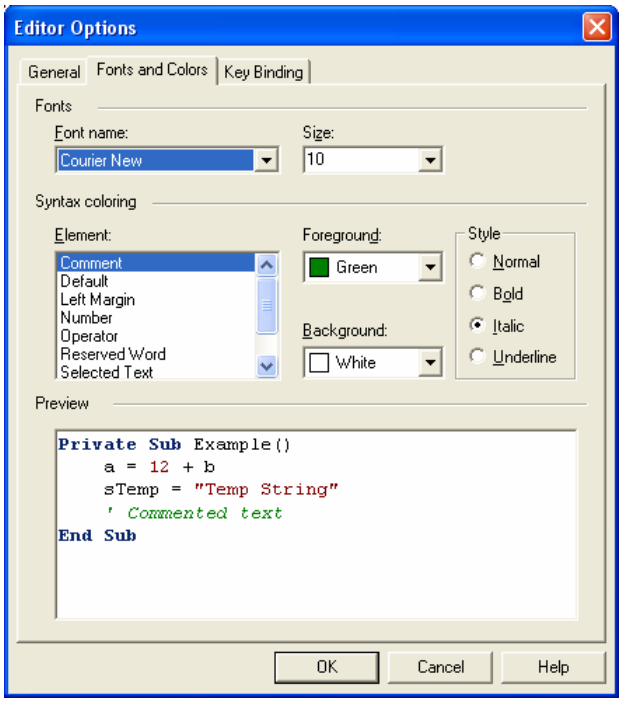

Figure 9 - Editor Options, Fonts and Colors

The First time you install **QuickTest Professional** on your computer, the default font for the Expert View is "Microsoft Sans Serif", It was designed to be metrically compatible with the original MS Sans bitmap font that shipped in early versions of Microsoft Windows. The original MS Sans was in the inflexible .fon bitmap format and could not be scaled. Microsoft Sans Serif is much more flexible and legible as it supports font antialiasing and scalable user interfaces. It's a nice font, but when displaying BLANKS (Spaces) he's very confusing. By a simple look, is hard to verify if you have 1 or 2 blanks. When using the object repository, is critical.
So the recommended font is "Courier New", because is a wide font, and the width is equal for every character.

General

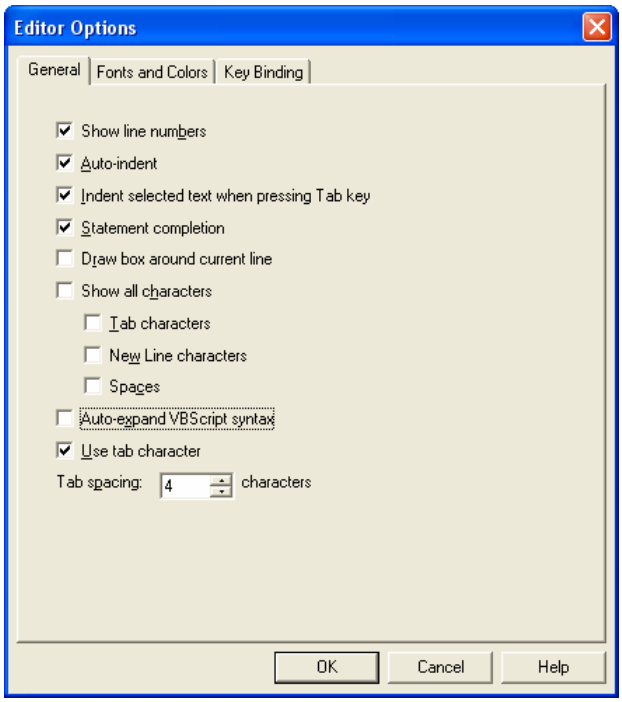

Figure 10 - Editor Options, General

Note that "Show line numbers" is checked, very usefull for debbuging your scripts.
"Tab Spacing" was set to 4 (default), for indetations
"Auto-expand VBScript syntax" was removed. If you are an experience user, auto expand, will only disturb your work.

Code Indentation

Code Indentation is, almost as important as writing code itself.
Code blocks are defined by their indentation. By "code block", I mean functions, if statements, for loops, while loops, and so forth. Indenting starts a block and unindenting ends it. There are no explicit braces, brackets, or keywords. This means that whitespace is significant, and must be consistent.

- Indent standard nested blocks four spaces.
- Indent the overview comments of a procedure one space.
- Indent the highest level statements that follow the overview comments four spaces, with each nested block indented an additional four spaces.
- The following code adheres to **VBScript** coding conventions.

```
Public Function FindUser (ByRef sUserList, ByVal sTargetUser)

'*********************************************************
' Purpose: Locates the first occurrence of a specified user
'          in the UserList array.
' Inputs: sUserList(): the list of users to be searched.
'         sTargetUser: the name of the user to search for.
' Returns: The index of the first occurrence of the sTargetUser
'          in the sUserList array.
'          If the target user is not found, return -1.
'*********************************************************
    Dim i        ' Loop counter.
    Dim bFound   ' Target found flag
    FindUser = -1
    i = 0   ' Initialize loop counter
    Do While i <= Ubound(sUserList) and Not bFound
      If sUserList(i) = sTargetUser Then
         bFound = True   ' Set flag to True
         FindUser = i   ' Set return value to loop count
      End If
      i = i + 1   ' Increment loop counter
    Loop
End Function
```

QuickTest Reusable Action Header

```
'************************************************************************
'@Author: <Author Name>
'@Name: <Action Name>
'@Description: <Description>
'@Param_In:  <Name>, <Type>, <Default>, <Description>
                      :
'@Param_In:  <Name>, <Type>, <Default>, <Description>
'@Param_Out:  <Name>, <Type>, <Description>
                      :
'@Param_Out:  <Name>, <Type>, <Description>
'@Excel_Param_In: <Name>, <Description>
'@Modifications: <#n By <Name>, Date: <dd-mmm-yyyy>> (Later modification on top)
'                <#n-1 By <Name>, Date: <dd-mmm-yyyy>>
'                Description: <modification description>
'************************************************************************
```

Creating an Action Template[1]

If you want to include one or more statements in every new action in your test, you can create an action template. For example, if you always enter your name as the author of an action, you can add this comment line to your action template. An

[1] QTP User's Guide > Creating Tests or Components > Working with Actions > Creating an Action Template

action template applies only to actions created on your computer.

To create an action template:

Create a text file containing the comments, function calls, and other statements that you want to include in your action template. The text file must be in the structure and format used in the **Expert View**.
Save the text file as *ActionTemplate.mst* in your <QuickTest Installation Folder>\dat folder. All new actions you create contain the script lines from the action template.
Note: Only the file name *ActionTemplate.mst* is recognized as an action template.