Chapter 21: Operators

Section 21.1: Concatenation Operators

VBA supports 2 different concatenation operators, `+` and `&` and both perform the exact same function when used with `String` types - the right-hand `String` is appended to the end of the left-hand `String`.

If the `&` operator is used with a variable type other than a `String`, it is implicitly cast to a `String` before being concatenated.

Note that the `+` concatenation operator is an overload of the `+` addition operator. The behavior of `+` is determined by the variable types of the operands and precedence of operator types. If both operands are typed as a `String` or `Variant` with a sub-type of `String`, they are concatenated:

```
Public Sub Example()
    Dim left As String
    Dim right As String

    left = "5"
    right = "5"

    Debug.Print left + right    'Prints "55"
End Sub
```

If *either* side is a numeric type and the other side is a `String` that can be coerced into a number, the type precedence of mathematical operators causes the operator to be treated as the addition operator and the numeric values are added:

```
Public Sub Example()
    Dim left As Variant
    Dim right As String

    left = 5
    right = "5"

    Debug.Print left + right    'Prints 10
End Sub
```

This behavior can lead to subtle, hard to debug errors - especially if `Variant` types are being used, so only the `&` operator should typically be used for concatenation.

Section 21.2: Comparison Operators

Token	Name	Description
`=`	Equal to	Returns `True` if the left-hand and right-hand operands are equal. Note that this is an overload of the assignment operator.
`<>`	Not equal to	Returns `True` if the left-hand and right-hand operands are not equal.
`>`	Greater than	Returns `True` if the left-hand operand is greater than the right-hand operand.
`<`	Less than	Returns `True` if the left-hand operand is less than the right-hand operand.
`>=`	Greater than or equal	Returns `True` if the if the left-hand operand is greater than or equal to the right-hand operand.
`<=`	Less than or equal	Returns `True` if the if the left-hand operand is less than or equal to the right-hand operand.

`Is`	Reference equity	Returns **`True`** if the left-hand object reference is the same instance as the right-hand object reference. It can also be used with **`Nothing`** (the null object reference) on either side. **Note:** The `Is` operator will attempt to coerce both operands into an `Object` before performing the comparison. If either side is a primitive type *or* a `Variant` that does not contain an object (either a non-object subtype or `vtEmpty`), the comparison will result in a Run-time error 424 - "Object required". If either operand belongs to a different *interface* of the same object, the comparison will return **`True`**. If you need to test for equity of both the instance *and* the interface, use `ObjPtr(left) = ObjPtr(right)` instead.

Notes

The VBA syntax allows for "chains" of comparison operators, but these constructs should generally be avoided. Comparisons are always performed from left to right on only 2 operands at a time, and each comparison results in a `Boolean`. For example, the expression...

```
a = 2: b = 1: c = 0
expr = a > b > c
```

...may be read in some contexts as a test of whether `b` is between `a` and `c`. In VBA, this evaluates as follows:

```
a = 2: b = 1: c = 0
expr = a > b > c
expr = (2 > 1) > 0
expr = True > 0
expr = -1 > 0 'CInt(True) = -1
expr = False
```

Any comparison operator other than `Is` used with an `Object` as an operand will be performed on the return value of the `Object`'s default member. If the object does not have a default member, the comparison will result in a Run-time error 438 - "Object doesn't support his property or method".

If the `Object` is unintitialized, the comparison will result in a Run-time error 91 - "Object variable or With block variable not set".

If the literal **`Nothing`** is used with any comparison operator other than `Is`, it will result in a Compile error - "Invalid use of object".

If the default member of the `Object` is *another* *`Object`*, VBA will continually call the default member of each successive return value until a primitive type is returned or an error is raised. For example, assume `SomeClass` has a default member of `Value`, which is an instance of `ChildClass` with a default member of `ChildValue`. The comparison...

```
Set x = New SomeClass
Debug.Print x > 42
```

...will be evaluated as:

```
Set x = New SomeClass
Debug.Print x.Value.ChildValue > 42
```

If either operand is a numeric type and the *other* operand is a `String` or `Variant` of subtype `String`, a numeric comparison will be performed. In this case, if the `String` cannot be cast to a number, a Run-time error 13 - "Type mismatch" will result from the comparison.

If ***both*** operands are a `String` or a `Variant` of subtype `String`, a string comparison will be performed based on the

Option Compare setting of the code module. These comparisons are performed on a character by character basis. Note that the *character representation* of a `String` containing a number is ***not*** the same as a comparison of the numeric values:

```
Public Sub Example()
    Dim left As Variant
    Dim right As Variant

    left = "42"
    right = "5"
    Debug.Print left > right              'Prints False
    Debug.Print Val(left) > Val(right)    'Prints True
End Sub
```

For this reason, make sure that `String` or `Variant` variables are cast to numbers before performing numeric inequity comparisons on them.

If one operand is a `Date`, a numeric comparison on the underlying Double value will be performed if the other operand is numeric or can be cast to a numeric type.

If the other operand is a `String` or a `Variant` of subtype `String` that can be cast to a `Date` using the current locale, the `String` will be cast to a `Date`. If it cannot be cast to a `Date` in the current locale, a Run-time error 13 - "Type mismatch" will result from the comparison.

Care should be taken when making comparisons between `Double` or `Single` values and Booleans. Unlike other numeric types, non-zero values cannot be assumed to be **`True`** due to VBA's behavior of promoting the data type of a comparison involving a floating point number to `Double`:

```
Public Sub Example()
    Dim Test As Double

    Test = 42        Debug.Print CBool(Test)             'Prints True.
    'True is promoted to Double - Test is not cast to Boolean
    Debug.Print Test = True              'Prints False

    'With explicit casts:
    Debug.Print CBool(Test) = True       'Prints True
    Debug.Print CDbl(-1) = CDbl(True)    'Prints True
End Sub
```

Section 21.3: Bitwise \ Logical Operators

All of the logical operators in VBA can be thought of as "overrides" of the bitwise operators of the same name. Technically, they are *always* treated as bitwise operators. All of the comparison operators in VBA return a Boolean, which will always have none of its bits set (**`False`**) or *all* of its bits set (**`True`**). But it will treat a value with *any* bit set as **`True`**. This means that the result of the casting the bitwise result of an expression to a `Boolean` (see Comparison Operators) will always be the same as treating it as a logical expression.

Assigning the result of an expression using one of these operators will give the bitwise result. Note that in the truth tables below, `0` is equivalent to **`False`** and `1` is equivalent to **`True`**.

`And`

Returns **`True`** if the expressions on both sides evaluate to **`True`**.

Left-hand Operand	Right-hand Operand	Result

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

`Or`

Returns `True` if either side of the expression evaluates to `True`.

Left-hand Operand	Right-hand Operand	Result
0	0	0
0	1	1
1	0	1
1	1	1

`Not`

Returns `True` if the expression evaluates to `False` and `False` if the expression evaluations to `True`.

Right-hand Operand	Result
0	1
1	0

`Not` is the only operand without a Left-hand operand. The Visual Basic Editor will automatically simplify expressions with a left hand argument. If you type...

```
Debug.Print x Not y
```

...the VBE will change the line to:

```
Debug.Print Not x
```

Similar simplifications will be made to any expression that contains a left-hand operand (including expressions) for `Not`.

`Xor`

Also known as "exclusive or". Returns `True` if both expressions evaluate to different results.

Left-hand Operand	Right-hand Operand	Result
0	0	0
0	1	1
1	0	1
1	1	0

Note that although the `Xor` operator can be *used* like a logical operator, there is absolutely no reason to do so as it gives the same result as the comparison operator `<>`.

`Eqv`

Also known as "equivalence". Returns `True` when both expressions evaluate to the same result.

Left-hand Operand	Right-hand Operand	Result
0	0	1

0	1	0
1	0	0
1	1	1

Note that the `Eqv` function is *very* rarely used as `x Eqv y` is equivalent to the much more readable `Not (x Xor y)`.

`Imp`

Also known as "implication". Returns `True` if both operands are the same *or* the second operand is `True`.

Left-hand Operand	Right-hand Operand	Result
0	0	1
0	1	1
1	0	0
1	1	1

Note that the `Imp` function is very rarely used. A good rule of thumb is that if you can't explain what it means, you should use another construct.

Section 21.4: Mathematical Operators

Listed in order of precedence:

Token	Name	Description
`^`	Exponentiation	Return the result of raising the left-hand operand to the power of the right-hand operand. Note that the value returned by exponentiation is *always* a `Double`, regardless of the value types being divided. Any coercion of the result into a variable type takes place ***after*** the calculation is performed.
`/`	Division1	Returns the result of dividing the left-hand operand by the right-hand operand. Note that the value returned by division is *always* a `Double`, regardless of the value types being divided. Any coercion of the result into a variable type takes place ***after*** the calculation is performed.
`*`	Multiplication1	Returns the product of 2 operands.
`\`	Integer Division	Returns the integer result of dividing the left-hand operand by the right-hand operand ***after*** rounding both sides with .5 rounding down. Any remainder of the division is ignored. If the right-hand operand (the divisor) is `0`, a Run-time error 11: Division by zero will result. Note that this is ***after*** all rounding is performed - expressions such as `3 \ 0.4` will also result in a division by zero error.
`Mod`	Modulo	Returns the integer remainder of dividing the left-hand operand by the right-hand operand. The operand on each side is rounded to an integer *before* the division, with .5 rounding down. For example, both `8.6 Mod 3` and `12 Mod 2.6` result in `0`. If the right-hand operand (the divisor) is `0`, a Run-time error 11: Division by zero will result. Note that this is ***after*** all rounding is performed - expressions such as `3 Mod 0.4` will also result in a division by zero error.
`-`	Subtraction2	Returns the result of subtracting the right-hand operand from the left-hand operand.
`+`	Addition2	Returns the sum of 2 operands. Note that this token also treated as a concatenation operator when it is applied to a `String`. See **Concatenation Operators**.

1 Multiplication and division are treated as having the same precedence.

2 Addition and subtraction are treated as having the same precedence.