Operators and Expressions in Visual Basic

Visual Studio 2015

An *operator* is a code element that performs an operation on one or more code elements that hold values. Value elements include variables, constants, literals, properties, returns from **Function** and **Operator** procedures, and expressions.

An *expression* is a series of value elements combined with operators, which yields a new value. The operators act on the value elements by performing calculations, comparisons, or other operations.

Types of Operators

Visual Basic provides the following types of operators:

- Arithmetic Operators perform familiar calculations on numeric values, including shifting their bit patterns.
- Comparison Operators compare two expressions and return a **Boolean** value representing the result of the comparison.
- Concatenation Operators join multiple strings into a single string.
- Logical and Bitwise Operators in Visual Basic combine **Boolean** or numeric values and return a result of the same data type as the values.

The value elements that are combined with an operator are called *operands* of that operator. Operators combined with value elements form expressions, except for the assignment operator, which forms a *statement*. For more information, see Statements in Visual Basic.

Evaluation of Expressions

The end result of an expression represents a value, which is typically of a familiar data type such as **Boolean**, **String**, or a numeric type.

The following are examples of expressions.

```
5 + 4

' The preceding expression evaluates to 9.

15 * System.Math.Sqrt(9) + x

' The preceding expression evaluates to 45 plus the value of x.

"Concat" & "ena" & "tion"
```

```
' The preceding expression evaluates to "Concatenation".

763 < 23

' The preceding expression evaluates to False.
```

Several operators can perform actions in a single expression or statement, as the following example illustrates.

VB

```
x = 45 + y * z ^ 2
```

In the preceding example, Visual Basic performs the operations in the expression on the right side of the assignment operator (=), then assigns the resulting value to the variable x on the left. There is no practical limit to the number of operators that can be combined into an expression, but an understanding of Operator Precedence in Visual Basic is necessary to ensure that you get the results you expect.

For more information and examples, see Operator Overloading in Visual Basic 2005.

See Also

Operators (Visual Basic)
Efficient Combination of Operators (Visual Basic)
Statements (Visual Basic)

Miscellaneous Operators (Visual Basic)

Visual Studio 2015

The following are miscellaneous operators defined in Visual Basic.

AddressOf Operator (Visual Basic)

Await Operator (Visual Basic)

GetType Operator (Visual Basic)

Function Expression (Visual Basic)

If Operator (Visual Basic)

TypeOf Operator (Visual Basic)

See Also

Operators Listed by Functionality (Visual Basic)

Await Operator (Visual Basic)

Visual Studio 2015

You apply the **Await** operator to an operand in an asynchronous method or lambda expression to suspend execution of the method until the awaited task completes. The task represents ongoing work.

The method in which **Await** is used must have an Async modifier. Such a method, defined by using the **Async** modifier, and usually containing one or more **Await** expressions, is referred to as an *async method*.

Note

The **Async** and **Await** keywords were introduced in Visual Studio 2012. For an introduction to async programming, see Asynchronous Programming with Async and Await (C# and Visual Basic).

Typically, the task to which you apply the **Await** operator is the return value from a call to a method that implements the Task-Based Asynchronous Pattern, that is, a Task or a Task(Of TResult).

In the following code, the HttpClient method GetByteArrayAsync returns `getContentsTask`, a **Task(Of Byte())**. The task is a promise to produce the actual byte array when the operation is complete. The **Await** operator is applied to `getContentsTask` to suspend execution in `SumPageSizesAsync` until `getContentsTask` is complete. In the meantime, control is returned to the caller of `SumPageSizesAsync`. When `getContentsTask` is finished, the **Await** expression evaluates to a byte array.

VB

```
Private Async Function SumPageSizesAsync() As Task

    ' To use the HttpClient type in desktop apps, you must include a using directive and
add a
    ' reference for the System.Net.Http namespace.
    Dim client As HttpClient = New HttpClient()
    ' . . .
    Dim getContentsTask As Task(Of Byte()) = client.GetByteArrayAsync(url)
    Dim urlContents As Byte() = Await getContentsTask

    ' Equivalently, now that you see how it works, you can write the same thing in a
single line.
    'Dim urlContents As Byte() = Await client.GetByteArrayAsync(url)
    ' . . .
End Function
```

Important

For the complete example, see Walkthrough: Accessing the Web by Using Async and Await (C# and Visual Basic). You can download the sample from Developer Code Samples on the Microsoft website. The example is in the AsyncWalkthrough_HttpClient project.

If **Await** is applied to the result of a method call that returns a **Task(Of TResult)**, the type of the **Await** expression is TResult. If **Await** is applied to the result of a method call that returns a **Task**, the **Await** expression doesn't return a value. The following example illustrates the difference.

VB

```
' Await used with a method that returns a Task(Of TResult).
Dim result As TResult = Await AsyncMethodThatReturnsTaskTResult()

' Await used with a method that returns a Task.
Await AsyncMethodThatReturnsTask()
```

An **Await** expression or statement does not block the thread on which it is executing. Instead, it causes the compiler to sign up the rest of the async method, after the **Await** expression, as a continuation on the awaited task. Control then returns to the caller of the async method. When the task completes, it invokes its continuation, and execution of the async method resumes where it left off.

An **Await** expression can occur only in the body of an immediately enclosing method or lambda expression that is marked by an **Async** modifier. The term *Await* serves as a keyword only in that context. Elsewhere, it is interpreted as an identifier. Within the async method or lambda expression, an **Await** expression cannot occur in a query expression, in the **catch** or **finally** block of a Try...Catch...Finally statement, in the loop control variable expression of a **For** or **For Each** loop, or in the body of a SyncLock statement.

Exceptions

Most async methods return a Task or Task(Of TResult). The properties of the returned task carry information about its status and history, such as whether the task is complete, whether the async method caused an exception or was canceled, and what the final result is. The **Await** operator accesses those properties.

If you await a task-returning async method that causes an exception, the **Await** operator rethrows the exception.

If you await a task-returning async method that is canceled, the **Await** operator rethrows an OperationCanceledException.

A single task that is in a faulted state can reflect multiple exceptions. For example, the task might be the result of a call to Task.WhenAll. When you await such a task, the await operation rethrows only one of the exceptions. However, you can't predict which of the exceptions is rethrown.

For examples of error handling in async methods, see Try...Catch...Finally Statement (Visual Basic).

Example

The following Windows Forms example illustrates the use of **Await** in an async method, `WaitAsynchronouslyAsync`.

Contrast the behavior of that method with the behavior of `WaitSynchronously`. Without an **Await** operator, `WaitSynchronously` runs synchronously despite the use of the **Async** modifier in its definition and a call to Thread.Sleep in its body.

VB

```
Private Async Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    ' Call the method that runs asynchronously.
    Dim result As String = Await WaitAsynchronouslyAsync()

    ' Call the method that runs synchronously.
    'Dim result As String = Await WaitSynchronously()

    ' Display the result.
    TextBox1.Text &= result
End Sub

' The following method runs asynchronously. The UI thread is not
' blocked during the delay. You can move or resize the Form1 window
' while Task.Delay is running.
Public Async Function WaitAsynchronouslyAsync() As Task(Of String)
    Await Task.Delay(10000)
    Return "Finished"
End Function

' The following method runs synchronously, despite the use of Async.
' You cannot move or resize the Form1 window while Thread.Sleep
' is running because the UI thread is blocked.
Public Async Function WaitSynchronously() As Task(Of String)
    ' Import System.Threading for the Sleep method.
    Thread.Sleep(10000)
    Return "Finished"
End Function
```

See Also

Asynchronous Programming with Async and Await (C# and Visual Basic)
Walkthrough: Accessing the Web by Using Async and Await (C# and Visual Basic)
Async (Visual Basic)

Function Expression (Visual Basic)

Visual Studio 2015

Declares the parameters and code that define a function lambda expression.

Syntax

```
Function ( [ parameterlist ] ) expression
- or -
Function ( [ parameterlist ] )
  [ statements ]
End Function
```

Parts

Term	Definition
parameterlist	Optional. A list of local variable names that represent the parameters of this procedure. The parentheses must be present even when the list is empty. See Parameter List (Visual Basic).
expression	Required. A single expression. The type of the expression is the return type of the function.
statements	Required. A list of statements that returns a value by using the **Return** statement. (See Return Statement (Visual Basic).) The type of the value returned is the return type of the function.

Remarks

A *lambda expression* is a function without a name that calculates and returns a value. You can use a lambda expression anywhere you can use a delegate type, except as an argument to **RemoveHandler**. For more information about delegates, and the use of lambda expressions with delegates, see Delegate Statement and Relaxed Delegate Conversion (Visual Basic).

Lambda Expression Syntax

The syntax of a lambda expression resembles that of a standard function. The differences are as follows:

- A lambda expression does not have a name.
- Lambda expressions cannot have modifiers, such as **Overloads** or **Overrides**.
- Lambda expressions do not use an **As** clause to designate the return type of the function. Instead, the type is inferred from the value that the body of a single-line lambda expression evaluates to, or the return value of a multiline lambda expression. For example, if the body of a single-line lambda expression is `Where cust.City = "London"`, its return type is **Boolean**.
- The body of a single-line lambda expression must be an expression, not a statement. The body can consist of a call to a function procedure, but not a call to a sub procedure.
- Either all parameters must have specified data types or all must be inferred.
- Optional and Paramarray parameters are not permitted.
- Generic parameters are not permitted.

Example

The following examples show two ways to create simple lambda expressions. The first uses a **Dim** to provide a name for the function. To call the function, you send in a value for the parameter.

VB

```
Dim add1 = Function(num As Integer) num + 1
```

VB

```
' The following line prints 6.
Console.WriteLine(add1(5))
```

Example

Alternatively, you can declare and run the function at the same time.

VB

```
Console.WriteLine((Function(num As Integer) num + 1)(5))
```

Example

Following is an example of a lambda expression that increments its argument and returns the value. The example shows both the single-line and multiline lambda expression syntax for a function. For more examples, see Lambda Expressions (Visual Basic).

VB

```
Dim increment1 = Function(x) x + 1
Dim increment2 = Function(x)
                     Return x + 2
                 End Function

' Write the value 2.
Console.WriteLine(increment1(1))

' Write the value 4.
Console.WriteLine(increment2(2))
```

Example

Lambda expressions underlie many of the query operators in Language-Integrated Query (LINQ), and can be used explicitly in method-based queries. The following example shows a typical LINQ query, followed by the translation of the query into method format.

VB

```
Dim londonCusts = From cust In db.Customers
                      Where cust.City = "London"
                      Select cust

' This query is compiled to the following code:
Dim londonCusts = db.Customers.
                  Where(Function(cust) cust.City = "London").
                  Select(Function(cust) cust)
```

For more information about query methods, see Queries (Visual Basic). For more information about standard query operators, see Standard Query Operators Overview.

See Also

- Function Statement (Visual Basic)
- Lambda Expressions (Visual Basic)
- Operators and Expressions in Visual Basic
- Statements in Visual Basic
- Value Comparisons (Visual Basic)
- Boolean Expressions (Visual Basic)
- If Operator (Visual Basic)
- Relaxed Delegate Conversion (Visual Basic)

Common Tasks Performed with Visual Basic Operators

Visual Studio 2015

Operators perform many common tasks involving one or more expressions called *operands*.

Arithmetic and Bit-shift Tasks

The following table summarizes the available arithmetic and bit-shift operations.

To	See
Add one numeric value to another	+ Operator (Visual Basic)
Subtract one numeric value from another	- Operator (Visual Basic)
Reverse the sign of a numeric value	- Operator (Visual Basic)
Multiply one numeric value by another	* Operator (Visual Basic)
Divide one numeric value into another	/ Operator (Visual Basic)
Find the quotient of one numeric value divided by another (without the remainder)	\ Operator (Visual Basic)
Find the remainder of one numeric value divided by another (without the quotient)	Mod Operator (Visual Basic)
Raise one numeric value to the power of another	^ Operator (Visual Basic)
Shift the bit pattern of a numeric value to the left	< < Operator (Visual Basic)
Shift the bit pattern of a numeric value to the right	> > Operator (Visual Basic)

Comparison Tasks

The following table summarizes the available comparison operations.

To	See

Determine whether two values are equal	= Operator (Comparison Operators in Visual Basic)
Determine whether two values are unequal	<> Operator (Comparison Operators in Visual Basic)
Determine whether one value is less than another	< Operator (Comparison Operators in Visual Basic)
Determine whether one value is greater than another	> Operator (Comparison Operators in Visual Basic)
Determine whether one value is less than or equal to another	<= Operator (Comparison Operators in Visual Basic)
Determine whether one value is greater than or equal to another	>= Operator (Comparison Operators in Visual Basic)
Determine whether two object variables refer to the same object instance	Is Operator (Visual Basic)
Determine whether two object variables refer to different object instances	IsNot Operator (Visual Basic)
Determine whether an object is of a specific type	TypeOf Operator (Visual Basic)

Concatenation Tasks

The following table summarizes the available concatenation operations.

To	See
Join multiple strings into a single string	**&** Operator (Concatenation Operators in Visual Basic)
Join numeric values with string values	**+** Operator (Concatenation Operators in Visual Basic)

Logical and Bitwise Tasks

The following table summarizes the available logical and bitwise operations.

To	See

Perform logical negation on a Boolean value	Not Operator (Visual Basic)
Perform logical conjunction on two Boolean values	And Operator (Visual Basic)
Perform inclusive logical disjunction on two Boolean values	Or Operator (Visual Basic)
Perform exclusive logical disjunction on two Boolean values	Xor Operator (Visual Basic)
Perform short-circuited logical conjunction on two Boolean values	AndAlso Operator (Visual Basic)
Perform short-circuited inclusive logical disjunction on two Boolean values	OrElse Operator (Visual Basic)
Perform bit-by-bit logical conjunction on two integral values	And Operator (Visual Basic)
Perform bit-by-bit inclusive logical disjunction on two integral values	Or Operator (Visual Basic)
Perform bit-by-bit exclusive logical disjunction on two integral values	Xor Operator (Visual Basic)
Perform bit-by-bit logical negation on an integral value	Not Operator (Visual Basic)

See Also

Operators and Expressions in Visual Basic
Operators Listed by Functionality (Visual Basic)

Arithmetic Operators in Visual Basic

Visual Studio 2015

Arithmetic operators are used to perform many of the familiar arithmetic operations that involve the calculation of numeric values represented by literals, variables, other expressions, function and property calls, and constants. Also classified with arithmetic operators are the bit-shift operators, which act at the level of the individual bits of the operands and shift their bit patterns to the left or right.

Arithmetic Operations

You can add two values in an expression together with the + Operator (Visual Basic), or subtract one from another with the - Operator (Visual Basic), as the following example demonstrates.

VB

```
Dim x As Integer
x = 67 + 34
x = 32 - 12
```

Negation also uses the - Operator (Visual Basic), but with only one operand, as the following example demonstrates.

VB

```
Dim x As Integer = 65
Dim y As Integer
y = -x
```

Multiplication and division use the * Operator (Visual Basic) and / Operator (Visual Basic), respectively, as the following example demonstrates.

VB

```
Dim y As Double
y = 45 * 55.23
y = 32 / 23
```

Exponentiation uses the ^ Operator (Visual Basic), as the following example demonstrates.

VB

```
Dim z As Double
z = 23 ^ 3
' The preceding statement sets z to 12167 (the cube of 23).
```

Integer division is carried out using the \ Operator (Visual Basic). Integer division returns the quotient, that is, the integer

that represents the number of times the divisor can divide into the dividend without consideration of any remainder. Both the divisor and the dividend must be integral types (**SByte**, **Byte**, **Short**, **UShort**, **Integer**, **UInteger**, **Long**, and **ULong**) for this operator. All other types must be converted to an integral type first. The following example demonstrates integer division.

VB

```
Dim k As Integer
k = 23 \ 5
' The preceding statement sets k to 4.
```

Modulus arithmetic is performed using the Mod Operator (Visual Basic). This operator returns the remainder after dividing the divisor into the dividend an integral number of times. If both divisor and dividend are integral types, the returned value is integral. If divisor and dividend are floating-point types, the returned value is also floating-point. The following example demonstrates this behavior.

VB

```
Dim x As Integer = 100
Dim y As Integer = 6
Dim z As Integer
z = x Mod y
' The preceding statement sets z to 4.
```

VB

```
Dim a As Double = 100.3
Dim b As Double = 4.13
Dim c As Double
c = a Mod b
' The preceding statement sets c to 1.18.
```

Attempted Division by Zero

Division by zero has different results depending on the data types involved. In integral divisions (**SByte**, **Byte**, **Short**, **UShort**, **Integer**, **UInteger**, **Long**, **ULong**), the .NET Framework throws a DivideByZeroException exception. In division operations on the **Decimal** or **Single** data type, the .NET Framework also throws a DivideByZeroException exception.

In floating-point divisions involving the **Double** data type, no exception is thrown, and the result is the class member representing NaN, PositiveInfinity, or NegativeInfinity, depending on the dividend. The following table summarizes the various results of attempting to divide a **Double** value by zero.

Dividend data type	Divisor data type	Dividend value	Result
Double	**Double**	0	NaN (not a mathematically defined number)
Double	**Double**	> 0	PositiveInfinity

Double	**Double**	< 0	NegativeInfinity

When you catch a DivideByZeroException exception, you can use its members to help you handle it. For example, the Message property holds the message text for the exception. For more information, see Try...Catch...Finally Statement (Visual Basic).

Bit-Shift Operations

A bit-shift operation performs an arithmetic shift on a bit pattern. The pattern is contained in the operand on the left, while the operand on the right specifies the number of positions to shift the pattern. You can shift the pattern to the right with the >> Operator (Visual Basic) or to the left with the << Operator (Visual Basic).

The data type of the pattern operand must be **SByte**, **Byte**, **Short**, **UShort**, **Integer**, **UInteger**, **Long**, or **ULong**. The data type of the shift amount operand must be **Integer** or must widen to **Integer**.

Arithmetic shifts are not circular, which means the bits shifted off one end of the result are not reintroduced at the other end. The bit positions vacated by a shift are set as follows:

- 0 for an arithmetic left shift
- 0 for an arithmetic right shift of a positive number
- 0 for an arithmetic right shift of an unsigned data type (**Byte**, **UShort**, **UInteger**, **ULong**)
- 1 for an arithmetic right shift of a negative number (**SByte**, **Short**, **Integer**, or **Long**)

The following example shifts an **Integer** value both left and right.

VB

```
Dim lResult, rResult As Integer
Dim pattern As Integer = 12
' The low-order bits of pattern are 0000 1100.
lResult = pattern << 3
' A left shift of 3 bits produces a value of 96.
rResult = pattern >> 2
' A right shift of 2 bits produces value of 3.
```

Arithmetic shifts never generate overflow exceptions.

Bitwise Operations

In addition to being logical operators, **Not**, **Or**, **And**, and **Xor** also perform bitwise arithmetic when used on numeric values. For more information, see "Bitwise Operations" in Logical and Bitwise Operators in Visual Basic.

Type Safety

Operands should normally be of the same type. For example, if you are doing addition with an **Integer** variable, you should add it to another **Integer** variable, and you should assign the result to a variable of type **Integer** as well.

One way to ensure good type-safe coding practice is to use the Option Strict Statement. If you set **Option Strict On**, Visual Basic automatically performs *type-safe* conversions. For example, if you try to add an **Integer** variable to a **Double** variable and assign the value to a **Double** variable, the operation proceeds normally, because an **Integer** value can be converted to **Double** without loss of data. Type-unsafe conversions, on the other hand, cause a compiler error with **Option Strict On**. For example, if you try to add an **Integer** variable to a **Double** variable and assign the value to an **Integer** variable, a compiler error results, because a **Double** variable cannot be implicitly converted to type **Integer**.

If you set **Option Strict Off**, however, Visual Basic allows implicit narrowing conversions to take place, although they can result in the unexpected loss of data or precision. For this reason, we recommend that you use **Option Strict On** when writing production code. For more information, see Widening and Narrowing Conversions (Visual Basic).

See Also

Arithmetic Operators (Visual Basic)
Bit Shift Operators (Visual Basic)
Comparison Operators in Visual Basic
Concatenation Operators in Visual Basic
Logical and Bitwise Operators in Visual Basic
Efficient Combination of Operators (Visual Basic)

Comparison Operators in Visual Basic

Visual Studio 2015

Comparison operators compare two expressions and return a **Boolean** value that represents the relationship of their values. There are operators for comparing numeric values, operators for comparing strings, and operators for comparing objects. All three types of operators are discussed herein.

Comparing Numeric Values

Visual Basic compares numeric values using six numeric comparison operators. Each operator takes as operands two expressions that evaluate to numeric values. The following table lists the operators and shows examples of each.

Operator	Condition tested	Examples

= (Equality)	Is the value of the first expression equal to the value of the second?	`23 = 33 ' False`
		`23 = 23 ' True`
		`23 = 12 ' False`

<> (Inequality)	Is the value of the first expression unequal to the value of the second?	`23 <> 33 ' True`
		`23 <> 23 ' False`
		`23 <> 12 ' True`

< (Less than)	Is the value of the first expression less than the value of the second?	`23 < 33 ' True`
		`23 < 23 ' False`
		`23 < 12 ' False`

> (Greater than)	Is the value of the first expression greater than the value of the second?	`23 > 33 ' False` `23 > 23 '`

		`False`
		
		`23 > 12 ' True`

<= (Less than or equal to)	Is the value of the first expression less than or equal to the value of the second?	`23 <= 33 ' True`
		
		`23 <= 23 ' True`
		
		`23 <= 12 ' False`

>= (Greater than or equal to)	Is the value of the first expression greater than or equal to the value of the second?	`23 >= 33 ' False`
		
		`23 >= 23 ' True`
		
		`23 >= 12 ' True`

Comparing Strings

Visual Basic compares strings using the Like Operator (Visual Basic) as well as the numeric comparison operators. The **Like** operator allows you to specify a pattern. The string is then compared against the pattern, and if it matches, the result is **True**. Otherwise, the result is **False**. The numeric operators allow you to compare **String** values based on their sort order, as the following example shows.

```
"73" < "9"

' The result of the preceding comparison is True.
```

The result in the preceding example is **True** because the first character in the first string sorts before the first character in the second string. If the first characters were equal, the comparison would continue to the next character in both strings, and so on. You can also test equality of strings using the equality operator, as the following example shows.

```
"734" = "734"

' The result of the preceding comparison is True.
```

If one string is a prefix of another, such as "aa" and "aaa", the longer string is considered to be greater than the shorter string. The following example illustrates this.

```
"aaa" > "aa"

' The result of the preceding comparison is True.
```

The sort order is based on either a binary comparison or a textual comparison depending on the setting of **Option**

Compare. For more information see Option Compare Statement.

Comparing Objects

Visual Basic compares two object reference variables with the Is Operator (Visual Basic) and the IsNot Operator (Visual Basic). You can use either of these operators to determine if two reference variables refer to the same object instance. The following example illustrates this.

VB

```
Dim x As testClass
Dim y As New testClass()
x = y
If x Is y Then
    ' Insert code to run if x and y point to the same instance.
End If
```

In the preceding example, `x Is y` evaluates to **True**, because both variables refer to the same instance. Contrast this result with the following example.

VB

```
Dim x As New customer()
Dim y As New customer()
If x Is y Then
    ' Insert code to run if x and y point to the same instance.
End If
```

In the preceding example, `x Is y` evaluates to **False**, because although the variables refer to objects of the same type, they refer to different instances of that type.

When you want to test for two objects not pointing to the same instance, the **IsNot** operator lets you avoid a grammatically clumsy combination of **Not** and **Is**. The following example illustrates this.

VB

```
Dim a As New classA()
Dim b As New classB()
If a IsNot b Then
    ' Insert code to run if a and b point to different instances.
End If
```

In the preceding example, `If a IsNot b` is equivalent to `If Not a Is b`.

Comparing Object Type

You can test whether an object is of a particular type with the **TypeOf**...**Is** expression. The syntax is as follows:

```
TypeOf <objectexpression> Is <typename>
```

When `typename` specifies an interface type, then the **TypeOf...Is** expression returns **True** if the object implements the interface type. When `typename` is a class type, then the expression returns **True** if the object is an instance of the specified class or of a class that derives from the specified class. The following example illustrates this.

VB

```
Dim x As System.Windows.Forms.Button
x = New System.Windows.Forms.Button()
If TypeOf x Is System.Windows.Forms.Control Then
    ' Insert code to run if x is of type System.Windows.Forms.Control.
End If
```

In the preceding example, the `TypeOf x Is Control` expression evaluates to **True** because the type of `x` is `Button`, which inherits from `Control`.

For more information, see TypeOf Operator (Visual Basic).

See Also

Value Comparisons (Visual Basic)
Comparison Operators (Visual Basic)
Operators (Visual Basic)
Arithmetic Operators in Visual Basic
Concatenation Operators in Visual Basic
Logical and Bitwise Operators in Visual Basic

How to: Test Whether Two Objects Are the Same (Visual Basic)

Visual Studio 2015

If you have two variables that refer to objects, you can use either the **Is** or **IsNot** operator, or both, to determine whether they refer to the same instance.

To test whether two objects are the same

- Use the Is Operator (Visual Basic) or the IsNot Operator (Visual Basic) with the two variables as operands.

VB

```
Public Sub processControl(ByVal f As System.Windows.Forms.Form,
    ByVal c As System.Windows.Forms.Control)
    Dim active As System.Windows.Forms.Control = f.ActiveControl
    If (active IsNot Nothing) And (c Is active) Then
        ' Insert code to process control c
    End If
    Return
End Sub
```

You might want to take a certain action depending on whether two objects refer to the same instance. The preceding example compares control `c` against the active control on form `f`. If there is no active control, or if there is one but it is not the same control instance as `c`, then the **If** statement fails and the procedure returns without further processing.

Whether you use **Is** or **IsNot** is a matter of personal convenience to you. One might be easier to read than the other in a given expression.

See Also

Comparison Operators in Visual Basic

How to: Match a String against a Pattern (Visual Basic)

Visual Studio 2015

If you want to find out if an expression of the String Data Type (Visual Basic) satisfies a pattern, then you can use the Like Operator (Visual Basic).

Like takes two operands. The left operand is a string expression, and the right operand is a string containing the pattern to be used for matching. **Like** returns a **Boolean** value indicating whether the string expression satisfies the pattern.

You can match each character in the string expression against a specific character, a wildcard character, a character list, or a character range. The positions of the specifications in the pattern string correspond to the positions of the characters to be matched in the string expression.

To match a character in the string expression against a specific character

- Put the specific character directly in the pattern string. Certain special characters must be enclosed in brackets (**[ ]**). For more information, see Like Operator (Visual Basic).

  The following example tests whether `myString` consists exactly of the single character `H`.

  VB

```
Dim sMatch As Boolean = myString Like "H"
```

To match a character in the string expression against a wildcard character

- Put a question mark (**?**) in the pattern string. Any valid character in this position makes a successful match.

  The following example tests whether `myString` consists of the single character `W` followed by exactly two characters of any values.

  VB

```
Dim sMatch As Boolean = myString Like "W??"
```

To match a character in the string expression against a list of characters

- Put brackets (**[ ]**) in the pattern string, and inside the brackets put the list of characters. Do not separate the characters with commas or any other separator. Any single character in the list makes a successful match.

  The following example tests whether `myString` consists of any valid character followed by exactly one of the characters `A`, `C`, or `E`.

  VB

  ```
  Dim sMatch As Boolean = myString Like "?[ACE]"
  ```

  Note that this match is case-sensitive.

To match a character in the string expression against a range of characters

- Put brackets (**[ ]**) in the pattern string, and inside the brackets put the lowest and highest characters in the range, separated by a hyphen (**–**). Any single character within the range makes a successful match.

  The following example tests whether `myString` consists of the characters `num` followed by exactly one of the characters `i`, `j`, `k`, `l`, `m`, or `n`.

  VB

  ```
  Dim sMatch As Boolean = myString Like "num[i-m]"
  ```

  Note that this match is case-sensitive.

Matching Empty Strings

Like treats the sequence `[]` as a zero-length string (**""**). You can use `[]` to test whether the entire string expression is empty, but you cannot use it to test if a particular position in the string expression is empty. If an empty position is one of the options you need to test for, you can use **Like** more than once.

To match a character in the string expression against a list of characters or no character

1. Call the **Like** operator twice on the same string expression, and connect the two calls with either the Or Operator (Visual Basic) or the OrElse Operator (Visual Basic).

2. In the pattern string for the first **Like** clause, include the character list, enclosed in brackets (**[ ]**).

3. In the pattern string for the second **Like** clause, do not put any character at the position in question.

   The following example tests the seven-digit telephone number `phoneNum` for exactly three numeric digits, followed by a space, a hyphen (**–**), a period (**.**), or no character at all, followed by exactly four numeric digits.

   VB

   ```
   Dim sMatch As Boolean =
   ```

```
    (phoneNum Like "###[ -.]####") OrElse (phoneNum Like "#######")
```

See Also

Comparison Operators (Visual Basic)
Operators and Expressions in Visual Basic
Like Operator (Visual Basic)
String Data Type (Visual Basic)

Like Operator (Visual Basic)

Visual Studio 2015

Compares a string against a pattern.

Syntax

```
result = string Like pattern
```

Parts

result
Required. Any **Boolean** variable. The result is a **Boolean** value indicating whether or not the *string* satisfies the *pattern*.

string
Required. Any **String** expression.

pattern
Required. Any **String** expression conforming to the pattern-matching conventions described in "Remarks."

Remarks

If the value in *string* satisfies the pattern contained in *pattern*, *result* is **True**. If the string does not satisfy the pattern, *result* is **False**. If both *string* and *pattern* are empty strings, the result is **True**.

Comparison Method

The behavior of the **Like** operator depends on the Option Compare Statement. The default string comparison method for each source file is **Option Compare Binary**.

Pattern Options

Built-in pattern matching provides a versatile tool for string comparisons. The pattern-matching features allow you to match each character in *string* against a specific character, a wildcard character, a character list, or a character range. The

following table shows the characters allowed in *pattern* and what they match.

Characters in *pattern*	**Matches in *string***
?	Any single character
*****	Zero or more characters
#	Any single digit (0–9)
[*charlist***]**	Any single character in *charlist*
[!*charlist***]**	Any single character not in *charlist*

Character Lists

A group of one or more characters (*charlist*) enclosed in brackets (**[ ]**) can be used to match any single character in *string* and can include almost any character code, including digits.

An exclamation point (**!**) at the beginning of *charlist* means that a match is made if any character except the characters in *charlist* is found in *string*. When used outside brackets, the exclamation point matches itself.

Special Characters

To match the special characters left bracket (**[**), question mark (**?**), number sign (**#**), and asterisk (*****), enclose them in brackets. The right bracket (**]**) cannot be used within a group to match itself, but it can be used outside a group as an individual character.

The character sequence `[]` is considered a zero-length string (**""**). However, it cannot be part of a character list enclosed in brackets. If you want to check whether a position in *string* contains one of a group of characters or no character at all, you can use **Like** twice. For an example, see How to: Match a String against a Pattern (Visual Basic).

Character Ranges

By using a hyphen (**–**) to separate the lower and upper bounds of the range, *charlist* can specify a range of characters. For example, `[A–Z]` results in a match if the corresponding character position in *string* contains any character within the range `A–Z`, and `[!H–L]` results in a match if the corresponding character position contains any character outside the range `H–L`.

When you specify a range of characters, they must appear in ascending sort order, that is, from lowest to highest. Thus, `[A–Z]` is a valid pattern, but `[Z–A]` is not.

Multiple Character Ranges

To specify multiple ranges for the same character position, put them within the same brackets without delimiters. For

example, `[A-CX-Z]` results in a match if the corresponding character position in *string* contains any character within either the range `A`–`C` or the range `X`–`Z`.

Usage of the Hyphen

A hyphen (**-**) can appear either at the beginning (after an exclamation point, if any) or at the end of *charlist* to match itself. In any other location, the hyphen identifies a range of characters delimited by the characters on either side of the hyphen.

Collating Sequence

The meaning of a specified range depends on the character ordering at run time, as determined by **Option Compare** and the locale setting of the system the code is running on. With **Option Compare Binary**, the range `[A-E]` matches `A`, `B`, `C`, `D`, and `E`. With **Option Compare Text**, `[A-E]` matches `A`, `a`, `À`, `à`, `B`, `b`, `C`, `c`, `D`, `d`, `E`, and `e`. The range does not match `Ê` or `ê` because accented characters collate after unaccented characters in the sort order.

Digraph Characters

In some languages, there are alphabetic characters that represent two separate characters. For example, several languages use the character `æ` to represent the characters `a` and `e` when they appear together. The **Like** operator recognizes that the single digraph character and the two individual characters are equivalent.

When a language that uses a digraph character is specified in the system locale settings, an occurrence of the single digraph character in either *pattern* or *string* matches the equivalent two-character sequence in the other string. Similarly, a digraph character in *pattern* enclosed in brackets (by itself, in a list, or in a range) matches the equivalent two-character sequence in *string*.

Overloading

The **Like** operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. If your code uses this operator on such a class or structure, be sure you understand its redefined behavior. For more information, see Operator Procedures (Visual Basic).

Example

This example uses the **Like** operator to compare strings to various patterns. The results go into a **Boolean** variable indicating whether each string satisfies the pattern.

VB

```
Dim testCheck As Boolean
' The following statement returns True (does "F" satisfy "F"?)
testCheck = "F" Like "F"
```

```
' The following statement returns False for Option Compare Binary
'    and True for Option Compare Text (does "F" satisfy "f"?)
testCheck = "F" Like "f"
' The following statement returns False (does "F" satisfy "FFF"?)
testCheck = "F" Like "FFF"
' The following statement returns True (does "aBBBa" have an "a" at the
'    beginning, an "a" at the end, and any number of characters in
'    between?)
testCheck = "aBBBa" Like "a*a"
' The following statement returns True (does "F" occur in the set of
'    characters from "A" through "Z"?)
testCheck = "F" Like "[A-Z]"
' The following statement returns False (does "F" NOT occur in the
'    set of characters from "A" through "Z"?)
testCheck = "F" Like "[!A-Z]"
' The following statement returns True (does "a2a" begin and end with
'    an "a" and have any single-digit number in between?)
testCheck = "a2a" Like "a#a"
' The following statement returns True (does "aM5b" begin with an "a",
'    followed by any character from the set "L" through "P", followed
'    by any single-digit number, and end with any character NOT in
'    the character set "c" through "e"?)
testCheck = "aM5b" Like "a[L-P]#[!c-e]"
' The following statement returns True (does "BAT123khg" begin with a
'    "B", followed by any single character, followed by a "T", and end
'    with zero or more characters of any type?)
testCheck = "BAT123khg" Like "B?T*"
' The following statement returns False (does "CAT123khg"?) begin with
'    a "B", followed by any single character, followed by a "T", and
'    end with zero or more characters of any type?)
testCheck = "CAT123khg" Like "B?T*"
```

See Also

- InStr
- StrComp
- Comparison Operators (Visual Basic)
- Operator Precedence in Visual Basic
- Operators Listed by Functionality (Visual Basic)
- Option Compare Statement
- Operators and Expressions in Visual Basic
- How to: Match a String against a Pattern (Visual Basic)

Concatenation Operators in Visual Basic

Visual Studio 2015

Concatenation operators join multiple strings into a single string. There are two concatenation operators, **+** and **&**. Both carry out the basic concatenation operation, as the following example shows.

```
```

These operators can also concatenate **String** variables, as the following example shows.

VB

```
Dim a As String = "abc"
Dim d As String = "def"
Dim z As String = a & d
Dim w As String = a + d
' The preceding statements set both z and w to "abcdef".
```

Differences Between the Two Concatenation Operators

The + Operator (Visual Basic) has the primary purpose of adding two numbers. However, it can also concatenate numeric operands with string operands. The **+** operator has a complex set of rules that determine whether to add, concatenate, signal a compiler error, or throw a run-time InvalidCastException exception.

The & Operator (Visual Basic) is defined only for **String** operands, and it always widens its operands to **String**, regardless of the setting of **Option Strict**. The **&** operator is recommended for string concatenation because it is defined exclusively for strings and reduces your chances of generating an unintended conversion.

Performance: String and StringBuilder

If you do a significant number of manipulations on a string, such as concatenations, deletions, and replacements, your performance might profit from the StringBuilder class in the System.Text namespace. It takes an extra instruction to create and initialize a StringBuilder object, and another instruction to convert its final value to a **String**, but you might recover this time because StringBuilder can perform faster.

See Also

Option Strict Statement

Types of String Manipulation Methods in Visual Basic
Arithmetic Operators in Visual Basic
Comparison Operators in Visual Basic
Logical and Bitwise Operators in Visual Basic

Logical and Bitwise Operators in Visual Basic

Visual Studio 2015

Logical operators compare **Boolean** expressions and return a **Boolean** result. The **And**, **Or**, **AndAlso**, **OrElse**, and **Xor** operators are *binary* because they take two operands, while the **Not** operator is *unary* because it takes a single operand. Some of these operators can also perform bitwise logical operations on integral values.

Unary Logical Operator

The Not Operator (Visual Basic) performs logical *negation* on a **Boolean** expression. It yields the logical opposite of its operand. If the expression evaluates to **True**, then **Not** returns **False**; if the expression evaluates to **False**, then **Not** returns **True**. The following example illustrates this.

VB

```
Dim x, y As Boolean
x = Not 23 > 14
y = Not 23 > 67
' The preceding statements set x to False and y to True.
```

Binary Logical Operators

The And Operator (Visual Basic) performs logical *conjunction* on two **Boolean** expressions. If both expressions evaluate to **True**, then **And** returns **True**. If at least one of the expressions evaluates to **False**, then **And** returns **False**.

The Or Operator (Visual Basic) performs logical *disjunction* or *inclusion* on two **Boolean** expressions. If either expression evaluates to **True**, or both evaluate to **True**, then **Or** returns **True**. If neither expression evaluates to **True**, **Or** returns **False**.

The Xor Operator (Visual Basic) performs logical *exclusion* on two **Boolean** expressions. If exactly one expression evaluates to **True**, but not both, **Xor** returns **True**. If both expressions evaluate to **True** or both evaluate to **False**, **Xor** returns **False**.

The following example illustrates the **And**, **Or**, and **Xor** operators.

VB

```
Dim a, b, c, d, e, f, g As Boolean

a = 23 > 14 And 11 > 8
b = 14 > 23 And 11 > 8
' The preceding statements set a to True and b to False.

c = 23 > 14 Or 8 > 11
d = 23 > 67 Or 8 > 11
```

```
' The preceding statements set c to True and d to False.

e = 23 > 67 Xor 11 > 8
f = 23 > 14 Xor 11 > 8
g = 14 > 23 Xor 8 > 11
' The preceding statements set e to True, f to False, and g to False.
```

Short-Circuiting Logical Operations

The AndAlso Operator (Visual Basic) is very similar to the **And** operator, in that it also performs logical conjunction on two **Boolean** expressions. The key difference between the two is that **AndAlso** exhibits *short-circuiting* behavior. If the first expression in an **AndAlso** expression evaluates to **False**, then the second expression is not evaluated because it cannot alter the final result, and **AndAlso** returns **False**.

Similarly, the OrElse Operator (Visual Basic) performs short-circuiting logical disjunction on two **Boolean** expressions. If the first expression in an **OrElse** expression evaluates to **True**, then the second expression is not evaluated because it cannot alter the final result, and **OrElse** returns **True**.

Short-Circuiting Trade-Offs

Short-circuiting can improve performance by not evaluating an expression that cannot alter the result of the logical operation. However, if that expression performs additional actions, short-circuiting skips those actions. For example, if the expression includes a call to a **Function** procedure, that procedure is not called if the expression is short-circuited, and any additional code contained in the **Function** does not run. Therefore, the function might run only occasionally, and might not be tested correctly. Or the program logic might depend on the code in the **Function**.

The following example illustrates the difference between **And**, **Or**, and their short-circuiting counterparts.

VB

```
Dim amount As Integer = 12
Dim highestAllowed As Integer = 45
Dim grandTotal As Integer
```

VB

```
If amount > highestAllowed And checkIfValid(amount) Then
    ' The preceding statement calls checkIfValid().
End If
If amount > highestAllowed AndAlso checkIfValid(amount) Then
    ' The preceding statement does not call checkIfValid().
End If
If amount < highestAllowed Or checkIfValid(amount) Then
    ' The preceding statement calls checkIfValid().
End If
If amount < highestAllowed OrElse checkIfValid(amount) Then
    ' The preceding statement does not call checkIfValid().
End If
```

VB

```
Function checkIfValid(ByVal checkValue As Integer) As Boolean
    If checkValue > 15 Then
        MsgBox(CStr(checkValue) & " is not a valid value.")
        ' The MsgBox warning is not displayed if the call to
        ' checkIfValid() is part of a short-circuited expression.
        Return False
    Else
        grandTotal += checkValue
        ' The grandTotal value is not updated if the call to
        ' checkIfValid() is part of a short-circuited expression.
        Return True
    End If
End Function
```

In the preceding example, note that some important code inside `checkIfValid()` does not run when the call is short-circuited. The first **If** statement calls `checkIfValid()` even though `12 > 45` returns **False**, because **And** does not short-circuit. The second **If** statement does not call `checkIfValid()`, because when `12 > 45` returns **False**, **AndAlso** short-circuits the second expression. The third **If** statement calls `checkIfValid()` even though `12 < 45` returns **True**, because **Or** does not short-circuit. The fourth **If** statement does not call `checkIfValid()`, because when `12 < 45` returns **True**, **OrElse** short-circuits the second expression.

Bitwise Operations

Bitwise operations evaluate two integral values in binary (base 2) form. They compare the bits at corresponding positions and then assign values based on the comparison. The following example illustrates the **And** operator.

VB

```
Dim x As Integer
x = 3 And 5
```

The preceding example sets the value of `x` to 1. This happens for the following reasons:

- The values are treated as binary:

  3 in binary form = 011

  5 in binary form = 101

- The **And** operator compares the binary representations, one binary position (bit) at a time. If both bits at a given position are 1, then a 1 is placed in that position in the result. If either bit is 0, then a 0 is placed in that position in the result. In the preceding example this works out as follows:

  011 (3 in binary form)

  101 (5 in binary form)

001 (The result, in binary form)

- The result is treated as decimal. The value 001 is the binary representation of 1, so x = 1.

The bitwise **Or** operation is similar, except that a 1 is assigned to the result bit if either or both of the compared bits is 1. **Xor** assigns a 1 to the result bit if exactly one of the compared bits (not both) is 1. **Not** takes a single operand and inverts all the bits, including the sign bit, and assigns that value to the result. This means that for signed positive numbers, **Not** always returns a negative value, and for negative numbers, **Not** always returns a positive or zero value.

The **AndAlso** and **OrElse** operators do not support bitwise operations.

Note
Bitwise operations can be performed on integral types only. Floating-point values must be converted to integral types before bitwise operation can proceed.

See Also

Logical/Bitwise Operators (Visual Basic)
Boolean Expressions (Visual Basic)
Arithmetic Operators in Visual Basic
Comparison Operators in Visual Basic
Concatenation Operators in Visual Basic
Efficient Combination of Operators (Visual Basic)

Efficient Combination of Operators (Visual Basic)

Visual Studio 2015

Complex expressions can contain many different operators. The following example illustrates this.

```
x = (45 * (y + z)) ^ (2 / 85) * 5 + z
```

Creating complex expressions such as the one in the preceding example requires a thorough understanding of the rules of operator precedence. For more information, see Operator Precedence in Visual Basic.

Parenthetical Expressions

Often you want operations to proceed in a different order from that determined by operator precedence. Consider the following example.

```
x = z * y + 4
```

The preceding example multiplies `z` by `y`, then adds the result to `4`. But if you want to add `y` and `4` before multiplying the result by `z`, you can override normal operator precedence by using parentheses. By enclosing an expression in parentheses, you force that expression to be evaluated first, regardless of operator precedence. To force the preceding example to do the addition first, you could rewrite it as in the following example.

```
x = z * (y + 4)
```

The preceding example adds `y` and `4`, then multiplies that sum by `z`.

Nested Parenthetical Expressions

You can nest expressions in multiple levels of parentheses to override precedence even further. The expressions most deeply nested in parentheses are evaluated first, followed by the next most deeply nested, and so on to the least deeply nested, and finally the expressions outside parentheses. The following example illustrates this.

```
x = (z * 4) ^ (y * (z + 2))
```

In the preceding example, `z + 2` is evaluated first, then the other parenthetical expressions. Exponentiation, which normally has higher precedence than addition or multiplication, is evaluated last in this example because the other expressions are enclosed in parentheses.

See Also

Arithmetic Operators in Visual Basic

Comparison Operators in Visual Basic
Logical and Bitwise Operators in Visual Basic
Logical/Bitwise Operators (Visual Basic)
Boolean Expressions (Visual Basic)
Value Comparisons (Visual Basic)
How to: Calculate Numeric Values (Visual Basic)
Operator Precedence in Visual Basic

How to: Calculate Numeric Values (Visual Basic)

Visual Studio 2015

You can calculate numeric values through the use of numeric expressions. A *numeric expression* is an expression that contains literals, constants, and variables representing numeric values, and operators that act on those values.

Calculating Numeric Values

To calculate a numeric value

- Combine one or more numeric literals, constants, and variables into a numeric expression. The following example shows some valid numeric expressions.

```
93.217

System.Math.PI

counter

4 * (67 + i)
```

The first three lines show a literal, a constant, and a variable. Each one forms a valid numeric expression by itself. The final line shows a combination of a variable with two literals.

Note that a numeric expression does not form a complete Visual Basic statement by itself. You must use the expression as part of a complete statement.

To store a numeric value

- You can use an assignment statement to assign the value represented by a numeric expression to a variable, as the following example demonstrates.

VB

```
Dim i As Integer = 2
Dim j As Integer
j = 4 * (67 + i)
```

In the preceding example, the value of the expression on the right side of the equal operator (**=**) is assigned to the variable `j` on the left side of the operator, so `j` evaluates to 276.

For more information, see Statements (Visual Basic).

Multiple Operators

If the numeric expression contains more than one operator, the order in which they are evaluated is determined by the rules of operator precedence. To override the rules of operator precedence, you enclose expressions in parentheses, as in the above example; the enclosed expressions are evaluated first.

To override normal operator precedence

- Use parentheses to enclose the operations you want to be performed first. The following example shows two different results with the same operands and operators.

VB

```
Dim i As Integer = 2
Dim j, k As Integer
j = 4 * (67 + i)
k = 4 * 67 + i
```

In the preceding example, the calculation for `j` performs the addition operator (**+**) first because the parentheses around `(67 + i)` override normal precedence, and the value assigned to `j` is 276 (4 times 69). The calculation for `k` performs the operators in their normal precedence (***** before **+**), and the value assigned to `k` is 270 (268 plus 2).

For more information, see Operator Precedence in Visual Basic.

See Also

Operators and Expressions in Visual Basic
Value Comparisons (Visual Basic)
Statements (Visual Basic)
Operator Precedence in Visual Basic
Arithmetic Operators (Visual Basic)
Efficient Combination of Operators (Visual Basic)

Value Comparisons (Visual Basic)

Visual Studio 2015

Comparison operators can be used to construct expressions that compare the values of numeric variables. These expressions return a **Boolean** value based on whether the comparison is true or false. Examples of such an expression are as follows.

```
45 > 26

26 > 45
```

The first expression evaluates to **True**, because 45 is greater than 26. The second example evaluates to **False**, because 26 is not greater than 45.

You can also compare numeric expressions in this fashion. The expressions you compare can themselves be complex expressions, as in the following example.

```
x / 45 * (y +17) >= System.Math.Sqrt(z) / (p - (x * 16))
```

The preceding complex expression includes literals, variables, and function calls. The expressions on both sides of the comparison operator are evaluated, and the resulting values are then compared using the **>=** comparison operator. If the value of the expression on the left side is greater than or equal to the value of the expression on the right, the entire expression evaluates to **True**; otherwise, it evaluates to **False**.

Expressions that compare values are most commonly used in **If...Then** constructions, as in the following example.

VB

```
If x > 50 Then
    ' Insert code to run if x is greater than 50.
Else
    ' Insert code to run if x is less than or equal to 50.
End If
```

The **=** sign is a comparison operator as well as an assignment operator. When used as a comparison operator, it evaluates whether the value on the left is equal to the value on the right, as shown in the following example.

VB

```
If x = 50 Then
    ' Insert code to continue program.
End If
```

You can also use a comparison expression anywhere a **Boolean** value is needed, such as in an **If**, **While**, **Loop**, or **ElseIf** statement, or when assigning to or passing a value to a **Boolean** variable. In the following example, the value returned by the comparison expression is assigned to a **Boolean** variable.

VB

```
Dim x As Boolean
x = 50 < 30
' The preceding statement assigns False to x.
```

See Also

Boolean Expressions (Visual Basic)
Operators and Expressions in Visual Basic
Comparison Operators in Visual Basic
How to: Calculate Numeric Values (Visual Basic)
Operator Precedence in Visual Basic

Boolean Expressions (Visual Basic)

Visual Studio 2015

A *Boolean expression* is an expression that evaluates to a value of the Boolean Data Type: **True** or **False**. **Boolean** expressions can take several forms. The simplest is the direct comparison of the value of a **Boolean** variable to a **Boolean** literal, as shown in the following example.

VB

```
If newCustomer = True Then
    ' Insert code to execute if newCustomer = True.
Else
    ' Insert code to execute if newCustomer = False.
End If
```

Two Meanings of the = Operator

Notice that the assignment statement `newCustomer = True` looks the same as the expression in the preceding example, but it performs a different function and is used differently. In the preceding example, the expression `newCustomer = True` represents a Boolean value, and the **=** sign is interpreted as a comparison operator. In a stand-alone statement, the **=** sign is interpreted as an assignment operator and assigns the value on the right to the variable on the left. The following example illustrates this.

VB

```
If newCustomer = True Then
    newCustomer = False
End If
```

For further information, see Value Comparisons (Visual Basic) and Statements (Visual Basic).

Comparison Operators

Comparison operators such as **=**, **<**, **>**, **<>**, **<=**, and **>=** produce Boolean expressions by comparing the expression on the left side of the operator to the expression on the right side of the operator and evaluating the result as **True** or **False**. The following example illustrates this.

```
42 < 81
```

Because 42 is less than 81, the Boolean expression in the preceding example evaluates to **True**. For more information on this kind of expression, see Value Comparisons (Visual Basic).

Comparison Operators Combined with Logical Operators

Comparison expressions can be combined using logical operators to produce more complex Boolean expressions. The following example demonstrates the use of comparison operators in conjunction with a logical operator.

```
x > y And x < 1000
```

In the preceding example, the value of the overall expression depends on the values of the expressions on each side of the **And** operator. If both expressions are **True**, then the overall expression evaluates to **True**. If either expression is **False**, then the entire expression evaluates to **False**.

Short-Circuiting Operators

The logical operators **AndAlso** and **OrElse** exhibit behavior known as *short-circuiting*. A short-circuiting operator evaluates the left operand first. If the left operand determines the value of the entire expression, then program execution proceeds without evaluating the right expression. The following example illustrates this.

VB

```
If 45 < 12 AndAlso testFunction(3) = 81 Then
    ' Add code to continue execution.
End If
```

In the preceding example, the operator evaluates the left expression, `45 < 12`. Because the left expression evaluates to **False**, the entire logical expression must evaluate to **False**. Program execution thus skips execution of the code within the **If** block without evaluating the right expression, `testFunction(3)`. This example does not call `testFunction()` because the left expression falsifies the entire expression.

Similarly, if the left expression in a logical expression using **OrElse** evaluates to **True**, execution proceeds to the next line of code without evaluating the right expression, because the left expression has already validated the entire expression.

Comparison with Non-Short-Circuiting Operators

By contrast, both sides of the logical operator are evaluated when the logical operators **And** and **Or** are used. The following example illustrates this.

VB

```
If 45 < 12 And testFunction(3) = 81 Then
    ' Add code to continue execution.
End If
```

The preceding example calls `testFunction()` even though the left expression evaluates to **False**.

Parenthetical Expressions

You can use parentheses to control the order of evaluation of Boolean expressions. Expressions enclosed by parentheses evaluate first. For multiple levels of nesting, precedence is granted to the most deeply nested expressions. Within parentheses, evaluation proceeds according to the rules of operator precedence. For more information, see Operator Precedence in Visual Basic.

See Also

Logical and Bitwise Operators in Visual Basic
Value Comparisons (Visual Basic)
Statements in Visual Basic
Comparison Operators (Visual Basic)
Efficient Combination of Operators (Visual Basic)
Operator Precedence in Visual Basic
Boolean Data Type (Visual Basic)

Arithmetic Operators (Visual Basic)

Visual Studio 2015

The following are the arithmetic operators defined in Visual Basic.

^ Operator

* Operator

/ Operator

\ Operator

Mod Operator

+ Operator (unary and binary)

- Operator (unary and binary)

See Also

Operator Precedence in Visual Basic
Arithmetic Operators in Visual Basic

Assignment Operators (Visual Basic)

Visual Studio 2015

The following are the assignment operators defined in Visual Basic.

= Operator

^= Operator

*= Operator

/= Operator

\= Operator

+= Operator

-= Operator

<<= Operator

>>= Operator

&= Operator

See Also

Operator Precedence in Visual Basic
Operators Listed by Functionality (Visual Basic)
Statements (Visual Basic)

Comparison Operators (Visual Basic)

Visual Studio 2015

The following are the comparison operators defined in Visual Basic.

< operator

<= operator

> operator

>= operator

= operator

<> operator

Is Operator (Visual Basic)

IsNot Operator (Visual Basic)

Like Operator (Visual Basic)

These operators compare two expressions to determine whether or not they are equal, and if not, how they differ. **Is**, **IsNot**, and **Like** are discussed in detail on separate Help pages. The relational comparison operators are discussed in detail on this page.

Syntax

```
      result = expression1 comparisonoperator expression2
result = object1 [Is | IsNot] object2
result = string Like pattern
```

Parts

result
Required. A **Boolean** value representing the result of the comparison.

expression
Required. Any expression.

comparisonoperator

Required. Any relational comparison operator.

object1, *object2*
Required. Any reference object names.

string
Required. Any **String** expression.

pattern
Required. Any **String** expression or range of characters.

Remarks

The following table contains a list of the relational comparison operators and the conditions that determine whether *result* is **True** or **False**.

Operator	True if	False if
< (Less than)	*expression1* < *expression2*	*expression1* >= *expression2*
<= (Less than or equal to)	*expression1* <= *expression2*	*expression1* > *expression2*
> (Greater than)	*expression1* > *expression2*	*expression1* <= *expression2*
>= (Greater than or equal to)	*expression1* >= *expression2*	*expression1* < *expression2*
= (Equal to)	*expression1* = *expression2*	*expression1* <> *expression2*
<> (Not equal to)	*expression1* <> *expression2*	*expression1* = *expression2*

Note

The = Operator (Visual Basic) is also used as an assignment operator.

The **Is** operator, the **IsNot** operator, and the **Like** operator have specific comparison functionalities that differ from the operators in the preceding table.

Comparing Numbers

When you compare an expression of type **Single** to one of type **Double**, the **Single** expression is converted to **Double**. This behavior is opposite to the behavior found in Visual Basic 6.

Similarly, when you compare an expression of type **Decimal** to an expression of type **Single** or **Double**, the **Decimal** expression is converted to **Single** or **Double**. For **Decimal** expressions, any fractional value less than *1E-28* might be lost. Such fractional value loss may cause two values to compare as equal when they are not. For this reason, you

should take care when using equality (=) to compare two floating-point variables. It is safer to test whether the absolute value of the difference between the two numbers is less than a small acceptable tolerance.

Floating-point Imprecision

When you work with floating-point numbers, keep in mind that they do not always have a precise representation in memory. This could lead to unexpected results from certain operations, such as value comparison and the Mod Operator (Visual Basic). For more information, see Troubleshooting Data Types (Visual Basic).

Comparing Strings

When you compare strings, the string expressions are evaluated based on their alphabetical sort order, which depends on the **Option Compare** setting.

Option Compare Binary bases string comparisons on a sort order derived from the internal binary representations of the characters. The sort order is determined by the code page. The following example shows a typical binary sort order.

```
A < B < E < Z < a < b < e < z < À < Ê < Ø < à < ê < ø
```

Option Compare Text bases string comparisons on a case-insensitive, textual sort order determined by your application's locale. When you set **Option Compare Text** and sort the characters in the preceding example, the following text sort order applies:

```
(A=a) < (À= à) < (B=b) < (E=e) < (Ê= ê) < (Ø = ø) < (Z=z)
```

Locale Dependence

When you set **Option Compare Text**, the result of a string comparison can depend on the locale in which the application is running. Two characters might compare as equal in one locale but not in another. If you are using a string comparison to make important decisions, such as whether to accept an attempt to log on, you should be alert to locale sensitivity. Consider either setting **Option Compare Binary** or calling the StrComp, which takes the locale into account.

Typeless Programming with Relational Comparison Operators

The use of relational comparison operators with **Object** expressions is not allowed under **Option Strict On**. When **Option Strict** is **Off**, and either *expression1* or *expression2* is an **Object** expression, the run-time types determine how they are compared. The following table shows how the expressions are compared and the result from the comparison, depending on the runtime type of the operands.

If operands are	**Comparison is**
Both **String**	Sort comparison based on string sorting characteristics.
Both numeric	Objects converted to **Double**, numeric comparison.

One numeric and one **String**	The **String** is converted to a **Double** and numeric comparison is performed. If the **String** cannot be converted to **Double**, an InvalidCastException is thrown.
Either or both are reference types other than **String**	An InvalidCastException is thrown.

Numeric comparisons treat **Nothing** as 0. String comparisons treat **Nothing** as **""** (an empty string).

Overloading

The relational comparison operators (**<**. **<=**, **>**, **>=**, **=**, **<>**) can be *overloaded*, which means that a class or structure can redefine their behavior when an operand has the type of that class or structure. If your code uses any of these operators on such a class or structure, be sure you understand the redefined behavior. For more information, see Operator Procedures (Visual Basic).

Notice that the = Operator (Visual Basic) can be overloaded only as a relational comparison operator, not as an assignment operator.

Example

The following example shows various uses of relational comparison operators, which you use to compare expressions. Relational comparison operators return a **Boolean** result that represents whether or not the stated expression evaluates to **True**. When you apply the **>** and **<** operators to strings, the comparison is made using the normal alphabetical sorting order of the strings. This order can be dependent on your locale setting. Whether the sort is case-sensitive or not depends on the Option Compare setting.

VB

```
Dim testResult As Boolean
testResult = 45 < 35
testResult = 45 = 45
testResult = 4 <> 3
testResult = "5" > "4444"
```

In the preceding example, the first comparison returns **False** and the remaining comparisons return **True**.

See Also

InvalidCastException
= Operator (Visual Basic)
Operator Precedence in Visual Basic
Operators Listed by Functionality (Visual Basic)
Troubleshooting Data Types (Visual Basic)
Comparison Operators in Visual Basic

Concatenation Operators (Visual Basic)

Visual Studio 2015

The following are the concatenation operators defined in Visual Basic.

& Operator

+ Operator

See Also

System.Text
StringBuilder
Operator Precedence in Visual Basic
Concatenation Operators in Visual Basic

Logical/Bitwise Operators (Visual Basic)

Visual Studio 2015

The following are the logical/bitwise operators defined in Visual Basic.

And Operator (Visual Basic)

Not Operator (Visual Basic)

Or Operator (Visual Basic)

Xor Operator (Visual Basic)

AndAlso Operator (Visual Basic)

OrElse Operator (Visual Basic)

IsFalse Operator (Visual Basic)

IsTrue Operator (Visual Basic)

See Also

Operator Precedence in Visual Basic
Logical and Bitwise Operators in Visual Basic

Bit Shift Operators (Visual Basic)

Visual Studio 2015

The following are the bit shift operators defined in Visual Basic.

<< Operator

>> Operator

See Also

Operators Listed by Functionality (Visual Basic)