

Contents

[Visual Basic Guide](#)

[What's New for Visual Basic](#)

[Get Started](#)

[Visual Basic Breaking Changes in Visual Studio](#)

[Additional Resources for Visual Basic Programmers](#)

[Developing Applications](#)

[Programming in Visual Basic](#)

[Accessing Computer Resources](#)

[Logging Information from the Application](#)

[Accessing User Data](#)

[Accessing Application Forms](#)

[Accessing Application Web Services](#)

[How to: Call a Web Service Asynchronously](#)

[Accessing Application Settings](#)

[Processing Drives, Directories, and Files](#)

[Development with My](#)

[Performing Tasks with My.Application, My.Computer, and My.User](#)

[Default Object Instances Provided by My.Forms and My.WebServices](#)

[Rapid Application Development with My.Resources and My.Settings](#)

[Overview of the Visual Basic Application Model](#)

[How My Depends on Project Type](#)

[Accessing Data](#)

[Creating and Using Components](#)

[Windows Forms Application Basics](#)

[Customizing Projects and Extending My with Visual Basic](#)

[Extending the My Namespace](#)

[Packaging and Deploying Custom My Extensions](#)

[Extending the Visual Basic Application Model](#)

[Customizing Which Objects are Available in My](#)

Programming Concepts

Asynchronous Programming with Async and Await

Attributes

Caller Information

Collections

Covariance and Contravariance

Expression Trees

Iterators

Language-Integrated Query (LINQ)

Object-Oriented Programming

Reflection

Serialization

Program Structure and Code Conventions

Structure of a Program

Main Procedure

References and the Imports Statement

Namespaces

Naming Conventions

Coding Conventions

Conditional Compilation

How to: Break and Combine Statements in Code

How to: Collapse and Hide Sections of Code

How to: Label Statements

Special Characters in Code

Comments in Code

Keywords as Element Names in Code

Me, My, MyBase, and MyClass

Limitations

Language Features

Arrays

Collection Initializers

Constants and Enumerations

[Control Flow](#)

[Data Types](#)

[Type Characters](#)

[Elementary Data Types](#)

[Numeric Data Types](#)

[Character Data Types](#)

[Miscellaneous Data Types](#)

[Composite Data Types](#)

[How to: Hold More Than One Value in a Variable](#)

[Generic Types](#)

[How to: Define a Class That Can Provide Identical Functionality on Different Data Types](#)

[How to: Use a Generic Class](#)

[Generic Procedures](#)

[Nullable Value Types](#)

[Value Types and Reference Types](#)

[Type Conversions](#)

[Widening and Narrowing Conversions](#)

[Implicit and Explicit Conversions](#)

[Conversions Between Strings and Other Types](#)

[How to: Convert an Object to Another Type](#)

[Array Conversions](#)

[Structures](#)

[How to: Declare a Structure](#)

[Structure Variables](#)

[Structures and Other Programming Elements](#)

[Structures and Classes](#)

[Tuples](#)

[Efficient Use of Data Types](#)

[Troubleshooting Data Types](#)

[Declared Elements](#)

[Delegates](#)

[Early and Late Binding](#)

- [Error Types](#)
- [Events](#)
- [Interfaces](#)
 - [Walkthrough: Creating and Implementing Interfaces](#)
- [LINQ](#)
- [Objects and Classes](#)
- [Operators and Expressions](#)
- [Procedures](#)
- [Statements](#)
- [Strings](#)
- [Variables](#)
- [XML](#)
- [COM Interop](#)
 - [Introduction to COM Interop](#)
 - [How to: Reference COM Objects](#)
 - [How to: Work with ActiveX Controls](#)
 - [Walkthrough: Calling Windows APIs](#)
 - [How to: Call Windows APIs](#)
 - [How to: Call a Windows Function that Takes Unsigned Types](#)
 - [Walkthrough: Creating COM Objects](#)
 - [Troubleshooting Interoperability](#)
 - [COM Interoperability in .NET Framework Applications](#)
 - [Walkthrough: Implementing Inheritance with COM Objects](#)
- [Language Reference](#)
 - [Configure language version](#)
 - [Typographic and Code Conventions](#)
 - [Visual Basic Runtime Library Members](#)
 - [Keywords](#)
 - [Arrays Summary](#)
 - [Collection Object Summary](#)
 - [Control Flow Summary](#)
 - [Conversion Summary](#)

[Data Types Summary](#)

[Dates and Times Summary](#)

[Declarations and Constants Summary](#)

[Directories and Files Summary](#)

[Errors Summary](#)

[Financial Summary](#)

[Information and Interaction Summary](#)

[Input and Output Summary](#)

[Math Summary](#)

[Derived Math Functions](#)

[My Reference](#)

[Operators Summary](#)

[Registry Summary](#)

[String Manipulation Summary](#)

[Attributes](#)

[Constants and Enumerations](#)

[Data Type Summary](#)

[Boolean Data Type](#)

[Byte Data Type](#)

[Char Data Type](#)

[Date Data Type](#)

[Decimal Data Type](#)

[Double Data Type](#)

[Integer Data Type](#)

[Long Data Type](#)

[Object Data Type](#)

[SByte Data Type](#)

[Short Data Type](#)

[Single Data Type](#)

[String Data Type](#)

[UInteger Data Type](#)

[ULong Data Type](#)

[User-Defined Data Type](#)

[UShort Data Type](#)

[Directives](#)

[#Const Directive](#)

[#ExternalSource Directive](#)

[#If...Then...#Else Directives](#)

[#Region Directive](#)

[Functions](#)

[Conversion Functions](#)

[Math Functions](#)

[String Functions](#)

[Type Conversion Functions](#)

[Return Values for the CStr Function](#)

[CType Function](#)

[Modifiers](#)

[Ansi](#)

[Assembly](#)

[Async](#)

[Auto](#)

[ByRef](#)

[ByVal](#)

[Default](#)

[Friend](#)

[In \(Generic Modifier\)](#)

[Iterator](#)

[Key](#)

[Module](#)

[MustInherit](#)

[MustOverride](#)

[Narrowing](#)

[NotInheritable](#)

[NotOverridable](#)

- [Optional](#)
- [Out \(Generic Modifier\)](#)
- [Overloads](#)
- [Overridable](#)
- [Overrides](#)
- [ParamArray](#)
- [Partial](#)
- [Private](#)
- [Private Protected](#)
- [Protected](#)
- [Protected Friend](#)
- [Public](#)
- [ReadOnly](#)
- [Shadows](#)
- [Shared](#)
- [Static](#)
- [Unicode](#)
- [Widening](#)
- [WithEvents](#)
- [WriteOnly](#)
- [Modules](#)
- [Nothing](#)
- [Objects](#)
- [My.Application Object](#)
 - [My.Application.Info Object](#)
 - [My.Application.Log Object](#)
- [My.Computer Object](#)
 - [My.Computer.Audio Object](#)
 - [My.Computer.Clipboard Object](#)
 - [My.Computer.Clock Object](#)
 - [My.Computer.FileSystem Object](#)
 - [My.Computer.FileSystem.SpecialDirectories Object](#)

[My.Computer.Info Object](#)
[My.Computer.Keyboard Object](#)
[My.Computer.Mouse Object](#)
[My.Computer.Network Object](#)
[My.Computer.Ports Object](#)
[My.Computer.Registry Object](#)
[My.Forms Object](#)
[My.Log Object](#)
[My.Request Object](#)
[My.Response Object](#)
[My.Resources Object](#)
[My.Settings Object](#)
[My.User Object](#)
[My.WebServices Object](#)
[TextFieldParser Object](#)

Operators

[Operator Precedence](#)
[Data Types of Operator Results](#)
[Operators Listed by Functionality](#)
[Arithmetic Operators](#)
[Assignment Operators](#)
[Bit Shift Operators](#)
[Comparison Operators](#)
[Concatenation Operators](#)
[Logical-Bitwise Operators](#)
[Miscellaneous Operators](#)

List of Visual Basic Operators

[& Operator](#)
[&= Operator](#)
[* Operator](#)
[*= Operator](#)
[+ Operator](#)

+ $=$ Operator
= $=$ Operator
- Operator
- $=$ Operator
< Operator
< $=$ Operator
> Operator
> $=$ Operator
<< Operator
<< $=$ Operator
>> Operator
>> $=$ Operator
 $/$ Operator
 $/=$ Operator
 \backslash Operator
 $\backslash=$ Operator
 $^$ Operator
 $^=$ Operator
? $()$ Operator
?. Operator
AddressOf Operator
And Operator
AndAlso Operator
Await Operator
DirectCast Operator
Function Expression
GetType Operator
GetXmlNamespace Operator
If Operator
Is Operator
IsFalse Operator
 IsNot Operator

- [IsTrue Operator](#)
- [Like Operator](#)
- [Mod Operator](#)
- [New Operator](#)
- [Not Operator](#)
- [Or Operator](#)
- [OrElse Operator](#)
- [Sub Expression](#)
- [TryCast Operator](#)
- [TypeOf Operator](#)
- [Xor Operator](#)
- [Properties](#)
- [Queries](#)
 - [Aggregate Clause](#)
 - [Distinct Clause](#)
 - [Equals Clause](#)
 - [From Clause](#)
 - [Group By Clause](#)
 - [Group Join Clause](#)
 - [Join Clause](#)
 - [Let Clause](#)
 - [Order By Clause](#)
 - [Select Clause](#)
 - [Skip Clause](#)
 - [Skip While Clause](#)
 - [Take Clause](#)
 - [Take While Clause](#)
 - [Where Clause](#)
- [Statements](#)
 - [A-E Statements](#)
 - [AddHandler Statement](#)
 - [Call Statement](#)

- [Class Statement](#)
- [Const Statement](#)
- [Continue Statement](#)
- [Declare Statement](#)
- [Delegate Statement](#)
- [Dim Statement](#)
- [Do...Loop Statement](#)
- [Else Statement](#)
- [End Statement](#)
- [End <keyword> Statement](#)
- [Enum Statement](#)
- [Erase Statement](#)
- [Error Statement](#)
- [Event Statement](#)
- [Exit Statement](#)
- [F-P Statements](#)
- [For Each...Next Statement](#)
- [For...Next Statement](#)
- [Function Statement](#)
- [Get Statement](#)
- [GoTo Statement](#)
- [If...Then...Else Statement](#)
- [Implements Statement](#)
- [Imports Statement \(.NET Namespace and Type\)](#)
- [Imports Statement \(XML Namespace\)](#)
- [Inherits Statement](#)
- [Interface Statement](#)
- [Mid Statement](#)
- [Module Statement](#)
- [Namespace Statement](#)
- [On Error Statement](#)
- [Operator Statement](#)

[Option <keyword> Statement](#)

[Option Compare Statement](#)

[Option Explicit Statement](#)

[Option Infer Statement](#)

[Option Strict Statement](#)

[Property Statement](#)

[Q-Z Statements](#)

[RaiseEvent Statement](#)

[ReDim Statement](#)

[REM Statement](#)

[RemoveHandler Statement](#)

[Resume Statement](#)

[Return Statement](#)

[Select...Case Statement](#)

[Set Statement](#)

[Stop Statement](#)

[Structure Statement](#)

[Sub Statement](#)

[SyncLock Statement](#)

[Then Statement](#)

[Throw Statement](#)

[Try...Catch...Finally Statement](#)

[Using Statement](#)

[While...End While Statement](#)

[With...End With Statement](#)

[Yield Statement](#)

[Clauses](#)

[Alias Clause](#)

[As Clause](#)

[Handles Clause](#)

[Implements Clause](#)

[In Clause](#)

[Into Clause](#)

[Of Clause](#)

[Declaration Contexts and Default Access Levels](#)

[Attribute List](#)

[Parameter List](#)

[Type List](#)

[XML Comment Tags](#)

[`<c>`](#)

[`<code>`](#)

[`<example>`](#)

[`<exception>`](#)

[`<include>`](#)

[`<list>`](#)

[`<para>`](#)

[`<param>`](#)

[`<paramref>`](#)

[`<permission>`](#)

[`<remarks>`](#)

[`<returns>`](#)

[`<see>`](#)

[`<seealso>`](#)

[`<summary>`](#)

[`<typeparam>`](#)

[`<value>`](#)

[XML Axis Properties](#)

[XML Attribute Axis Property](#)

[XML Child Axis Property](#)

[XML Descendant Axis Property](#)

[Extension Indexer Property](#)

[XML Value Property](#)

[XML Literals](#)

[XML Element Literal](#)

[XML Document Literal](#)

[XML CDATA Literal](#)

[XML Comment Literal](#)

[XML Processing Instruction Literal](#)

Error Messages

'#ElseIf' must be preceded by a matching '#If' or '#ElseIf'

'#Region' and '#End Region' statements are not valid within method bodies-multiline lambdas

'<attribute>' cannot be applied because the format of the GUID '<number>' is not correct

'<classname>' is not CLS-compliant because the interface '<interfacename>' it implements is not CLS-compliant

'<elementname>' is obsolete (Visual Basic Warning)

'<eventname>' is an event, and cannot be called directly

'<expression>' cannot be used as a type constraint

'<functionname>' is not declared (Smart Device-Visual Basic Compiler Error)

'<interfacename>.<membername>' is already implemented by the base class '<baseclassname>'. Re-implementation of <type> assumed

'<keyword>' is valid only within an instance method

'<membername>' cannot expose type '<typename>' outside the project through <containertype> '<containertypename>'

'<membername>' is ambiguous across the inherited interfaces '<interfacename1>' and '<interfacename2>'

<message> This error could also be due to mixing a file reference with a project reference to assembly '<assemblyname>'

'<methodname>' has multiple definitions with identical signatures

'<name>' is ambiguous in the namespace '<namespacename>'

'<name1>' is ambiguous, imported from the namespaces or types '<name2>'

<proceduresignature1> is not CLS-compliant because it overloads <proceduresignature2> which differs from it only by array of array parameter types or by the rank of the array parameter types

'<type1>'<typename>' must implement '<membername>' for interface '<interfacename>'

'<type1>'<typename>' must implement '<methodname>' for interface '<interfacename>'

'<typename>' cannot inherit from <type> '<basetypename>' because it expands the access of the base <type> outside the assembly

'<typename>' is a delegate type

'<typename>' is a type and cannot be used as an expression

A double quote is not a valid comment token for delimited fields where EscapeQuote is set to True

A property or method call cannot include a reference to a private object, either as an argument or as a return value

A reference was created to embedded interop assembly '<assembly1>' because of an indirect reference to that assembly from assembly '<assembly2>'

A startup form has not been specified

Access of shared member through an instance; qualifying expression will not be evaluated

'AddressOf' operand must be the name of a method (without parentheses)

An unexpected error has occurred because an operating system resource required for single instance startup cannot be acquired

Anonymous type member name can be inferred only from a simple or qualified name with no arguments

Argument not optional

Array bounds cannot appear in type specifiers

Array declared as for loop control variable cannot be declared with an initial size

Array subscript expression missing

Arrays declared as structure members cannot be declared with an initial size

'As Any' is not supported in 'Declare' statements

Attribute '<attributename>' cannot be applied multiple times

Automation error

Bad checksum value, non hex digits or odd number of hex digits

Bad DLL calling convention

Bad file mode

Bad file name or number

Bad record length

Because this call is not awaited, the current method continues to run before the call is completed

Cannot convert anonymous type to expression tree because it contains a field that is used in the initialization of another field

Cannot create ActiveX Component

Cannot refer to '<name>' because it is a member of the value-typed field '<name>' of class '<classname>' which has 'System.MarshalByRefObject' as a base class

Cannot refer to an instance member of a class from within a shared method or shared member initializer without an explicit instance of the class

Can't create necessary temporary file

Can't open '<filename>' for writing

Class '<classname>' cannot be found

Class does not support Automation or does not support expected interface

'Class' statement must end with a matching 'End Class'

Clipboard format is not valid

Constant expression not representable in type '<typename>'

Constants must be of an intrinsic or enumerated type, not a class, structure, type parameter, or array type

Constructor '<name>' cannot call itself

Copying the value of 'ByRef' parameter '<parametername>' back to the matching argument narrows from type '<typename1>' to type '<typename2>'

'Custom' modifier is not valid on events declared without explicit delegate types

Data type(s) of the type parameter(s) cannot be inferred from these arguments

Declaration expected

Default property '<propertyname1>' conflicts with default property '<propertyname2>' in '<classname>' and so should be declared 'Shadows'

Default property access is ambiguous between the inherited interface members '<defaultpropertyname>' of interface '<interfacename1>' and '<defaultpropertyname>' of interface '<interfacename2>'

Delegate class '<classname>' has no Invoke method, so an expression of this type cannot be the target of a method call

Derived classes cannot raise base class events

Device I/O error

'Dir' function must first be called with a 'PathName' argument

End of statement expected

Error creating assembly manifest: <error message>

Error creating Win32 resources: <error message>

Error in loading DLL

Error saving temporary Win32 resource file '<filename>': <error message>

Errors occurred while compiling the XML schemas in the project

Evaluation of expression or statement timed out

Event '<eventname1>' cannot implement event '<eventname2>' on interface '<interface>' because their delegate types '<delegate1>' and '<delegate2>' do not match

Events cannot be declared with a delegate type that has a return type

Events of shared WithEvents variables cannot be handled by non-shared methods

Expression does not produce a value

Expression has the type '<typename>' which is a restricted type and cannot be used to access members inherited from 'Object' or 'ValueType'

Expression is a value and therefore cannot be the target of an assignment

Expression of type <type> is not queryable

Expression recursively calls the containing property '<propertyname>'

Expression too complex

'Extension' attribute can be applied only to 'Module', 'Sub', or 'Function' declarations

File already open

File is too large to read into a byte array

File name or class name not found during Automation operation

File not found (Visual Basic Run-Time Error)

First operand in a binary 'If' expression must be nullable or a reference type

First statement of this 'Sub New' must be a call to ' MyBase.New' or ' MyClass.New' (No Accessible Constructor Without Parameters)

First statement of this 'Sub New' must be an explicit call to ' MyBase.New' or ' MyClass.New' because the '<constructorname>' in the base class '<baseclassname>' of '<derivedclassname>' is marked obsolete: '<errormessage>'

'For Each' on type '<typename>' is ambiguous because the type implements multiple instantiations of 'System.Collections.Generic.IEnumerable(Of T)'

Friend assembly reference <reference> is invalid

Function '<procedurename>' doesn't return a value on all code paths

Function evaluation is disabled because a previous function evaluation timed out

Generic parameters used as optional parameter types must be class constrained

'Get' accessor of property '<propertyname>' is not accessible

Handles clause requires a WithEvents variable defined in the containing type or one

of its base types

Identifier expected

Identifier is too long

Initializer expected

Input past end of file

Internal error happened at <location>

Implicit conversion from '<typename1>' to '<typename2>' in copying the value of 'ByRef' parameter '<parametername>' back to the matching argument.

'Is' requires operands that have reference types, but this operand has the value type '<typename>'

' IsNot' operand of type 'typename' can only be compared to 'Nothing', because 'typename' is a nullable type

Labels that are numbers must be followed by colons

Lambda expression will not be removed from this event handler

Lambda expressions are not valid in the first expression of a 'Select Case' statement

Late bound resolution; runtime errors could occur

Latebound overload resolution cannot be applied to '<procedurename>' because the accessing instance is an interface type

Leading '.' or '!' can only appear inside a 'With' statement

Line is too long

'Line' statements are no longer supported (Visual Basic Compiler Error)

Method does not have a signature compatible with the delegate

Methods of 'System.Nullable(Of T)' cannot be used as operands of the 'AddressOf' operator

'Module' statements can occur only at file or namespace level

Name <membername> is not CLS-compliant

Name '<name>' is not declared

Name <namespacename> in the root namespace <fullnamespacename> is not CLS-compliant

Namespace or type specified in the Imports '<qualifiedelementname>' doesn't contain any public member or cannot be found

Namespace or type specified in the project-level Imports '<qualifiedelementname>' doesn't contain any public member or cannot be found

Need property array index

Nested function does not have a signature that is compatible with delegate ''

No accessible 'Main' method with an appropriate signature was found in '<name>'

Non-CLS-compliant <membername> is not allowed in a CLS-compliant interface

Nullable type inference is not supported in this context

Number of indices exceeds the number of dimensions of the indexed array

Object or class does not support the set of events

Object required

Object variable or With block variable not set

Operator declaration must be one of: +,-,*,-,^, &, Like, Mod, And, Or, Xor, Not, <<, >>, =, <>, <, <=, >, >=, CType, IsTrue, IsFalse

'Optional' expected

Optional parameters must specify a default value

Ordinal is not valid

Out of memory (Visual Basic Compiler Error)

Out of stack space

Out of string space

Overflow (Visual Basic Error)

Overflow (Visual Basic Run-Time Error)

Path not found

Path-File access error

Permission denied

Procedure call or argument is not valid

Property '<propertyname>' doesn't return a value on all code paths

Property array index is not valid

Property let procedure not defined and property get procedure did not return an object

Property not found

Property or method not found

Range variable <variable> hides a variable in an enclosing block, a previously defined range variable, or an implicitly declared variable in a query expression

Range variable name can be inferred only from a simple or qualified name with no arguments

Reference required to assembly '<assemblyidentity>' containing type '<typename>'

but a suitable reference could not be found due to ambiguity between projects '`<projectname1>`' and '`<projectname2>`'

Reference required to assembly '`<assemblyname>`' containing the base class '`<classname>`'

Requested operation is not available (BC35000)

Resume without error

Return type of function '`<procedurename>`' is not CLS-compliant

'Set' accessor of property '`<propertyname>`' is not accessible

Some subkeys cannot be deleted

Statement cannot end a block outside of a line 'If' statement

Statement is not valid in a namespace

Statement is not valid inside a method-multiline lambda

String constants must end with a double quote

Structure '`<structurename>`' must contain at least one instance member variable or at least one instance event declaration not marked 'Custom'

'Sub Main' was not found in '`<name>`'

Sub or Function not defined

Subscript out of range

TextFieldParser is unable to complete the read operation because maximum buffer size has been exceeded

The type for variable '`<variablename>`' will not be inferred because it is bound to a field in an enclosing scope

This array is fixed or temporarily locked

This key is already associated with an element of this collection

Too many files

Type '`<typename>`' has no constructors

Type `<typename>` is not CLS-compliant

Type '`<typename>`' is not defined

Type arguments could not be inferred from the delegate

Type mismatch

Type of '`<variablename>`' cannot be inferred because the loop bounds and the step variable do not widen to the same type

Type of member '`<membername>`' is not CLS-compliant

Type of optional value for optional parameter `<parametername>` is not CLS-

compliant

Type of parameter '<parametername>' is not CLS-compliant

Type parameters cannot be used as qualifiers

Unable to create strong-named assembly from key file '<filename>': <error>

Unable to embed resource file '<filename>': <error message>

Unable to emit assembly: <error message>

Unable to find required file '<filename>'

Unable to get serial port names because of an internal system error

Unable to link to resource file '<filename>': <error message>

Unable to load information for class '<classname>'

Unable to write output to memory

Unable to write temporary file because temporary path is not available

Unable to write to output file '<filename>': <error>

Underlying type <typename> of Enum is not CLS-compliant

Using the iteration variable in a lambda expression may have unexpected results

Value of type '<typename1>' cannot be converted to '<typename2>'

Value of type '<typename1>' cannot be converted to '<typename2>' (Multiple file references)

Value of type 'type1' cannot be converted to 'type2'

Variable '<variablename>' hides a variable in an enclosing block

Variable '<variablename>' is used before it has been assigned a value

Variable uses an Automation type not supported in Visual Basic

XML axis properties do not support late binding

XML comment exception must have a 'cref' attribute

XML entity references are not supported

XML literals and XML properties are not supported in embedded code within ASP.NET

XML namespace URI '<uri>' can be bound only to 'xmlns'

Reference

[Command-Line Compiler](#)

[Building from the Command Line](#)

[How to: Invoke the Command-Line Compiler](#)

[Sample Compilation Command Lines](#)

Compiler Options Listed Alphabetically

@ (Specify Response File)

-addmodule

-baseaddress

-bugreport

-codepage

-debug

-define

-delaysign

-deterministic

-doc

-errorreport

-filealign

-help, /?

-highentropyva

-imports

-keycontainer

-keyfile

-langversion

-libpath

-link

-linkresource

-main

-moduleassemblyname

-netcf

-noconfig

-nologo

-nostdlib

-nowarn

-nowin32manifest

-optimize

-optioncompare

- optionexplicit
- optioninfer
- optionstrict
- out
- platform
- quiet
- recurse
- reference
- refonly
- refout
- removeintchecks
- resource
- rootnamespace
- sdkpath
- target
- subsystemversion
- utf8output
- vbruntime
- verbose
- warnaserror
- win32icon
- win32manifest
- win32resource

[Compiler Options Listed by Category](#)

[.NET Framework Reference Information](#)

[Language Specification](#)

[Sample Applications](#)

[Walkthroughs](#)

Visual Basic Guide

10/18/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic is engineered for productively building type-safe and object-oriented applications. Visual Basic enables developers to target Windows, Web, and mobile devices. As with all languages targeting the Microsoft .NET Framework, programs written in Visual Basic benefit from security and language interoperability.

This generation of Visual Basic continues the tradition of giving you a fast and easy way to create .NET Framework-based applications.

If you don't already have Visual Basic, you can download a free version of Visual Studio that includes Visual Basic from the [Visual Studio](#) site.

In This Section

- [Getting Started](#)
Helps you begin working by listing what is new and what is available in various editions of the product.
- [Programming Concepts](#)
Presents the language concepts that are most useful to Visual Basic programmers.
- [Program Structure and Code Conventions](#)
Contains documentation on the basic structure and code conventions of Visual Basic such as naming conventions, comments in code, and limitations within Visual Basic.
- [Visual Basic Language Features](#)
Provides links to topics that introduce and discuss important features of Visual Basic, including LINQ and XML literals.
- [Visual Basic Reference](#)
Contains the Visual Basic language and compiler information.
- [Developing Applications with Visual Basic](#)
Discusses various aspects of development in Visual Basic, such as security, exception handling, and using the .NET Framework class library.
- [COM Interop](#)
Explains the interoperability issues associated with creating and using component object model (COM) objects with Visual Basic.
- [Samples](#)
Contains information about samples.
- [Walkthroughs](#)
Provides links to step-by-step instructions for common scenarios.

Related Sections

- [Get Started Developing with Visual Studio](#)
Provides links to topics that help you learn about the basics of Visual Studio.
- [.NET API Browser](#)
Provides entry to the library of classes, interfaces, and value types that are included in the Microsoft .NET Framework SDK.

What's new for Visual Basic

10/18/2019 • 12 minutes to read • [Edit Online](#)

This topic lists key feature names for each version of Visual Basic, with detailed descriptions of the new and enhanced features in the latest versions of the language.

Current version

Visual Basic 16.0 / Visual Studio 2019 Version 16.0

For new features, see [Visual Basic 16.0](#).

Previous versions

Visual Basic 15.8 / Visual Studio 2017 Version 15.8

For new features, see [Visual Basic 15.8](#).

Visual Basic 15.5 / Visual Studio 2017 Version 15.5

For new features, see [Visual Basic 15.5](#).

Visual Basic 15.3 / Visual Studio 2017 Version 15.3

For new features, see [Visual Basic 15.3](#).

Visual Basic 2017 / Visual Studio 2017

For new features, see [Visual Basic 2017](#).

Visual Basic / Visual Studio 2015

For new features, see [Visual Basic 14](#).

Visual Basic / Visual Studio 2013

Technology previews of the .NET Compiler Platform ("Roslyn")

Visual Basic / Visual Studio 2012

`Async` and `await` keywords, iterators, caller info attributes

Visual Basic, Visual Studio 2010

Auto-implemented properties, collection initializers, implicit line continuation, dynamic, generic co/contra variance, global namespace access

Visual Basic / Visual Studio 2008

Language Integrated Query (LINQ), XML literals, local type inference, object initializers, anonymous types, extension methods, local `var` type inference, lambda expressions, `if` operator, partial methods, nullable value types

Visual Basic / Visual Studio 2005

The `My` type and helper types (access to app, computer, files system, network)

Visual Basic / Visual Studio .NET 2003

Bit-shift operators, loop variable declaration

Visual Basic / Visual Studio .NET 2002

The first release of Visual Basic .NET

Visual Basic 16.0

Visual Basic 16.0 focuses on supplying more of the features of the Visual Basic Runtime (`microsoft.visualbasic.dll`) to .NET Core and is the first version of Visual Basic focused on .NET Core. Many portions of the Visual Basic Runtime depend on WinForms and these will be added in a later version of Visual Basic.

Comments allowed in more places within statements

In Visual Basic 15.8 and earlier versions, comments are only allowed on blank lines, at the end of a statement, or in specific places within a statement where an implicit line continuation is allowed. Starting with Visual Basic 16.0, comments are also allowed after explicit line continuations and within a statement on a line beginning with a space followed by an underscore.

```
Public Sub Main()
    cmd.CommandText = ' Comment is allowed here without _
                      "SELECT * FROM Titles JOIN Publishers " _ ' This is a comment
                      & "ON Publishers.PubId = Titles.PubID " _
                      - ' This is a comment on a line without code
                      & "WHERE Publishers.State = 'CA'"
End Sub
```

Visual Basic 15.8

Optimized floating-point to integer conversion

In previous versions of Visual Basic, conversion of `Double` and `Single` values to integers offered relatively poor performance. Visual Basic 15.8 significantly enhances the performance of floating-point conversions to integers when you pass the value returned by any of the following methods to one of the [intrinsic Visual Basic integer conversion functions](#) (`CByte`, `CShort`, `CInt`, `CLng`, `CSByte`, `CUShort`, `CUInt`, `CULng`), or when the value returned by any of the following methods is implicitly cast to an integral type when `Option Strict` is set to `off`:

- [Conversion.Fix\(Double\)](#)
- [Conversion.Fix\(Object\)](#)
- [Conversion.Fix\(Single\)](#)
- [Conversion.Int\(Double\)](#)
- [Conversion.Int\(Object\)](#)
- [Conversion.Int\(Single\)](#)
- [Math.Ceiling\(Double\)](#)
- [Math.Floor\(Double\)](#)
- [Math.Round\(Double\)](#)
- [Math.Truncate\(Double\)](#)

This optimization allows code to run faster -- up to twice as fast for code that does a large number of conversions to integer types. The following example illustrates some simple method calls that are affected by this optimization:

```
Dim s As Single = 173.7619
Dim d As Double = s

Dim i1 As Integer = CInt(Fix(s))           ' Result: 173
Dim b1 As Byte = CByte(Int(d))            ' Result: 173
Dim s1 AS Short = CShort(Math.Truncate(s)) ' Result: 173
Dim i2 As Integer = CInt(Math.Ceiling(d))   ' Result: 174
Dim i3 As Integer = CInt(Math.Round(s))     ' Result: 174
```

Note that this truncates rather than rounds floating-point values.

Visual Basic 15.5

Non-trailing named arguments

In Visual Basic 15.3 and earlier versions, when a method call included arguments both by position and by name, positional arguments had to precede named arguments. Starting with Visual Basic 15.5, positional and named arguments can appear in any order as long as all arguments up to the last positional argument are in the correct position. This is particularly useful when named arguments are used to make code more readable.

For example, the following method call has two positional arguments between a named argument. The named argument makes it clear that the value 19 represents an age.

```
StudentInfo.Display("Mary", age:=19, #9/21/1998#)
```

`Private Protected` member access modifier

This new keyword combination defines a member that is accessible by all members in its containing class as well as by types derived from the containing class, but only if they are also found in the containing assembly. Because structures cannot be inherited, `Private Protected` can only be applied to the members of a class.

Leading hex/binary/octal separator

Visual Basic 2017 added support for the underscore character (`_`) as a digit separator. Starting with Visual Basic 15.5, you can use the underscore character as a leading separator between the prefix and hexadecimal, binary, or octal digits. The following example uses a leading digit separator to define 3,271,948,384 as a hexadecimal number:

```
Dim number As Integer = &H_C305_F860
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

Visual Basic 15.3

Named tuple inference

When you assign the value of tuple elements from variables, Visual Basic infers the name of tuple elements from the corresponding variable names; you do not have to explicitly name a tuple element. The following example uses inference to create a tuple with three named elements, `state`, `stateName`, and `capital`.

```
Const state As String = "MI"
Const stateName As String = "Michigan"
Const capital As String = "Lansing"
Dim stateInfo = ( state, stateName, capital )
Console.WriteLine($"{stateInfo.stateName}: 2-letter code: {stateInfo.State}, Capital {stateInfo.capital}")
' The example displays the following output:
'   Michigan: 2-letter code: MI, Capital Lansing
```

Additional compiler switches

The Visual Basic command-line compiler now supports the `-refout` and `-refonly` compiler options to control the

output of reference assemblies. **-refout** defines the output directory of the reference assembly, and **-refonly** specifies that only a reference assembly is to be output by compilation.

Visual Basic 2017

Tuples

Tuples are a lightweight data structure that most commonly is used to return multiple values from a single method call. Ordinarily, to return multiple values from a method, you have to do one of the following:

- Define a custom type (a `Class` or a `Structure`). This is a heavyweight solution.
- Define one or more `ByRef` parameters, in addition to returning a value from the method.

Visual Basic's support for tuples lets you quickly define a tuple, optionally assign semantic names to its values, and quickly retrieve its values. The following example wraps a call to the `TryParse` method and returns a tuple.

```
Imports System.Globalization

Public Module NumericLibrary
    Public Function ParseInteger(value As String) As (Success As Boolean, Number As Int32)
        Dim number As Integer
        Return (Int32.TryParse(value, NumberStyles.Any, CultureInfo.InvariantCulture, number), number)
    End Function
End Module
```

You can then call the method and handle the returned tuple with code like the following.

```
Dim numericString As String = "123,456"
Dim result = ParseInteger(numericString)
Console.WriteLine($"{If(result.Success, $"Success: {result.Number:N0}", "Failure")}")
Console.ReadLine()
'   Output: Success: 123,456
```

Binary literals and digit separators

You can define a binary literal by using the prefix `&B` or `&b`. In addition, you can use the underscore character, `_`, as a digit separator to enhance readability. The following example uses both features to assign a `Byte` value and to display it as a decimal, hexadecimal, and binary number.

```
Dim value As Byte = &B0110_1110
Console.WriteLine($"{NameOf(value)} = {value} (hex: 0x{value:X2}) " +
    $"(binary: {Convert.ToString(value, 2)})")
' The example displays the following output:
'   value = 110 (hex: 0x6E) (binary: 1101110)
```

For more information, see the "Literal assignments" section of the [Byte](#), [Integer](#), [Long](#), [Short](#), [SByte](#), [UInteger](#), [ULong](#), and [UShort](#) data types.

Support for C# reference return values

Starting with C# 7.0, C# supports reference return values. That is, when the calling method receives a value returned by reference, it can change the value of the reference. Visual Basic does not allow you to author methods with reference return values, but it does allow you to consume and modify the reference return values.

For example, the following `Sentence` class written in C# includes a `FindNext` method that finds the next word in a sentence that begins with a specified substring. The string is returned as a reference return value, and a `Boolean` variable passed by reference to the method indicates whether the search was successful. This means that the caller

can not only read the returned value; he or she can also modify it, and that modification is reflected in the `Sentence` class.

```
using System;

public class Sentence
{
    private string[] words;
    private int currentSearchPointer;

    public Sentence(string sentence)
    {
        words = sentence.Split(' ');
        currentSearchPointer = -1;
    }

    public ref string FindNext(string startWithString, ref bool found)
    {
        for (int count = currentSearchPointer + 1; count < words.Length; count++)
        {
            if (words[count].StartsWith(startWithString))
            {
                currentSearchPointer = count;
                found = true;
                return ref words[currentSearchPointer];
            }
        }
        currentSearchPointer = -1;
        found = false;
        return ref words[0];
    }

    public string GetSentence()
    {
        string stringToReturn = null;
        foreach (var word in words)
            stringToReturn += $"{word} ";

        return stringToReturn.Trim();
    }
}
```

In its simplest form, you can modify the word found in the sentence by using code like the following. Note that you are not assigning a value to the method, but rather to the expression that the method returns, which is the reference return value.

```
Dim sentence As New Sentence("A time to see the world is now.")
Dim found = False
sentence.FindNext("A", found) = "A good"
Console.WriteLine(sentence.GetSentence())
' The example displays the following output:
'     A good time to see the world is now.
```

A problem with this code, though, is that if a match is not found, the method returns the first word. Since the example does not examine the value of the `Boolean` argument to determine whether a match is found, it modifies the first word if there is no match. The following example corrects this by replacing the first word with itself if there is no match.

```

Dim sentence As New Sentence("A time to see the world is now.")
Dim found = False
sentence.FindNext("A", found) = IIf(found, "A good", sentence.FindNext("B", found))
Console.WriteLine(sentence.GetSentence())
' The example displays the following output:
'     A good time to see the world is now.

```

A better solution is to use a helper method to which the reference return value is passed by reference. The helper method can then modify the argument passed to it by reference. The following example does that.

```

Module Example
Public Sub Main()
    Dim sentence As New Sentence("A time to see the world is now.")
    Dim found = False
    Dim returns = RefHelper(sentence.FindNext("A", found), "A good", found)
    Console.WriteLine(sentence.GetSentence())
End Sub

Private Function RefHelper(ByRef stringFound As String, replacement As String, success As Boolean) _
    As (originalString As String, found As Boolean)
    Dim originalString = stringFound
    If found Then stringFound = replacement
    Return (originalString, found)
End Function
End Module
' The example displays the following output:
'     A good time to see the world is now.

```

For more information, see [Reference Return Values](#).

Visual Basic 14

[Nameof](#)

You can get the unqualified string name of a type or member for use in an error message without hard coding a string. This allows your code to remain correct when refactoring. This feature is also useful for hooking up model-view-controller MVC links and firing property changed events.

[String interpolation](#)

You can use string interpolation expressions to construct strings. An interpolated string expression looks like a template string that contains expressions. An interpolated string is easier to understand with respect to arguments than [Composite Formatting](#).

[Null-conditional member access and indexing](#)

You can test for null in a very light syntactic way before performing a member access (`?.`) or index (`?[]`) operation. These operators help you write less code to handle null checks, especially for descending into data structures. If the left operand or object reference is null, the operations returns null.

[Multi-line string literals](#)

String literals can contain newline sequences. You no longer need the old work around of using

```
<xml><![CDATA[...text with newlines...]]></xml>.Value
```

Comments

You can put comments after implicit line continuations, inside initializer expressions, and among LINQ expression terms.

Smarter fully-qualified name resolution

Given code such as `Threading.Thread.Sleep(1000)`, Visual Basic used to look up the namespace "Threading", discover it was ambiguous between `System.Threading` and `System.Windows.Threading`, and then report an error. Visual Basic now considers both possible namespaces together. If you show the completion list, the Visual Studio editor lists members from both types in the completion list.

Year-first date literals

You can have date literals in yyyy-mm-dd format, `#2015-03-17 16:10 PM#`.

Readonly interface properties

You can implement readonly interface properties using a readwrite property. The interface guarantees minimum functionality, and it does not stop an implementing class from allowing the property to be set.

[TypeOf <expr> IsNot <type>](#)

For more readability of your code, you can now use `TypeOf` with `IsNot`.

[#Disable Warning <ID> and #Enable Warning <ID>](#)

You can disable and enable specific warnings for regions within a source file.

XML doc comment improvements

When writing doc comments, you get smart editor and build support for validating parameter names, proper handling of `crefs` (generics, operators, etc.), colorizing, and refactoring.

[Partial module and interface definitions](#)

In addition to classes and structs, you can declare partial modules and interfaces.

[#Region directives inside method bodies](#)

You can put `#Region...#End Region` delimiters anywhere in a file, inside functions, and even spanning across function bodies.

[Overrides definitions are implicitly overloads](#)

If you add the `overrides` modifier to a definition, the compiler implicitly adds `Overloads` so that you can type less code in common cases.

CObj allowed in attributes arguments

The compiler used to give an error that `CObj(...)` was not a constant when used in attribute constructions.

Declaring and consuming ambiguous methods from different interfaces

Previously the following code yielded errors that prevented you from declaring `IMock` or from calling `GetDetails` (if these had been declared in C#):

```
Interface ICustomer
    Sub GetDetails(x As Integer)
End Interface

Interface ITime
    Sub GetDetails(x As String)
End Interface

Interface IMock : Inherits ICustomer, ITime
    Overloads Sub GetDetails(x As Char)
End Interface

Interface IMock2 : Inherits ICustomer, ITime
End Interface
```

Now the compiler will use normal overload resolution rules to choose the most appropriate `GetDetails` to call, and you can declare interface relationships in Visual Basic like those shown in the sample.

See also

- [What's New in Visual Studio 2017](#)

Get started with Visual Basic

5/23/2019 • 2 minutes to read • [Edit Online](#)

This section of the documentation helps you get started with Visual Basic application development.

Get started with Visual Basic and .NET Core

[Build a Visual Basic Hello World application with .NET Core in Visual Studio 2017](#)

[Build a class library with Visual Basic and .NET Core in Visual Studio 2017](#)

Additional information

- [What's new for Visual Basic](#)
Lists new features in each of the versions of Visual Basic .NET.
- [Visual Basic Breaking Changes in Visual Studio](#)
Lists changes in this release that might prevent an application from compiling or change its run-time behavior.
- [Additional Resources for Visual Basic Programmers](#)
Provides a list of Web sites and newsgroups that can help you find answers to common problems.

See also

- [Get Visual Basic](#)
Provides download links for Visual Studio versions that include Visual Basic support, including free versions.
- [Visual Basic Fundamentals for Absolute Beginners](#)
Microsoft Virtual Academy course that teaches the fundamentals of Visual Basic programming.
- [Object-Oriented Programming](#)
Provides links to pages that introduce object-oriented programming and describe how to create your own objects and how to use objects to simplify your coding.
- [Samples](#)
Provides links to sample code in Visual Basic.
- [Walkthroughs](#)
Provides a list of Help pages that demonstrate aspects of the Visual Basic language.
- [Talk to Us](#)
Covers how to receive support and give feedback.
- [Visual Studio](#)
Provides links into the Visual Studio documentation.
- [C#](#)
Provides links into the documentation on application development with Visual C#.
- [Visual C++](#)
Provides links into the Visual C++ documentation.

- [Office and SharePoint Development](#)

Provides information about using Microsoft Office and Visual Studio as part of a business application.

Visual Basic Breaking Changes in Visual Studio

4/2/2019 • 2 minutes to read • [Edit Online](#)

For the latest documentation on Visual Studio 2017, see [Visual Studio 2017 Documentation](#).

No changes in Visual Basic in Visual Studio 2015 will prevent an application that was created in Visual Basic in Visual Studio 2013 from compiling or change the run-time behavior of such an application.

See also

- [Arrays](#)
- [LINQ](#)
- [Lambda Expressions](#)
- [For Each...Next Statement](#)
- [Getting Started](#)
- [When is a non-breaking language fix breaking?](#)

Additional resources

8/7/2019 • 2 minutes to read • [Edit Online](#)

The following web sites provide guidance and can help you find answers to common problems.

Microsoft resources

On the web

URL	DESCRIPTION
Visual Basic .NET Language Design	Official repository on GitHub for Visual Basic .NET language design.
Microsoft Visual Basic Team Blog	Provides access to the Visual Basic team blog.

Code samples

URL	DESCRIPTION
Visual Basic documentation samples	Contains the samples used throughout the Visual Basic and .NET documentation.

Forums

URL	DESCRIPTION
Visual Basic Forums	Discusses general Visual Basic issues.

Videos and webcasts

URL	DESCRIPTION
Channel9	Provides continuous community through videos, Wikis, and forums.

Support

URL	DESCRIPTION
Microsoft Support	Provides access to Knowledge Base (KB) articles, downloads and updates, support webcasts, and other services.
Visual Studio Questions	Enables you to file bugs or provide suggestions to Microsoft about .NET and Visual Studio. You can also report a bug by selecting Help > Send Feedback > Report a Problem in Visual Studio.

Third-party resources

URL	DESCRIPTION
VBForums	Provides a forum to discuss Visual Basic, .NET, and more.
vbCity	A community site for people to learn and ask questions about Visual Basic and .NET.
Stack Overflow	Stack Overflow is a question and answer site for developers.

See also

- [Get started with Visual Basic](#)
- [Talk to Us](#)

Developing Applications with Visual Basic

3/5/2019 • 2 minutes to read • [Edit Online](#)

This section covers conceptual documentation for the Visual Basic language.

In This Section

[Programming in Visual Basic](#)

Covers a variety of programming subjects.

[Development with My](#)

Discusses a new feature called `My`, which provides access to information and default object instances that are related to an application and its run-time environment.

[Accessing Data in Visual Basic Applications](#)

Contains Help on accessing data in Visual Basic.

[Creating and Using Components in Visual Basic](#)

Defines the term *component* and discusses how and when to create components.

[Windows Forms Application Basics](#)

Provides information about creating Windows Forms applications by using Visual Studio.

[Customizing Projects and Extending My](#)

Describes how you can customize project templates to provide additional `My` objects.

Related Sections

[Visual Basic Programming Guide](#)

Walks through the essential elements of programming with Visual Basic.

[Visual Basic Language Reference](#)

Contains reference documentation for the Visual Basic language.

Programming in Visual Basic

4/2/2019 • 2 minutes to read • [Edit Online](#)

This section discusses programming tasks that you may want to learn more about as you create your Visual Basic application.

In this section

[Accessing Computer Resources](#)

Contains documentation on how to use the `My.Computer` object to access information about the computer on which an application runs and how to control the computer.

[Logging Information from the Application](#)

Contains documentation on how to log information from your application using the `My.Application.Log` and `My.Log` objects, and how to extend the application's logging capabilities.

[Accessing User Data](#)

Contains documentation on tasks that you can accomplish using the `My.User` object.

[Accessing Application Forms](#)

Contains documentation on accessing an application's forms by using the `My.Forms` and `My.Application` objects.

[Accessing Application Web Services](#)

Contains documentation on how to access the Web services referenced by the application using the `My.WebServices` object.

[Accessing Application Settings](#)

Contains documentation on accessing an application's settings using the `My.Settings` object.

[Processing Drives, Directories, and Files](#)

Contains documentation on how to access the file system using the `My.Computer.FileSystem` object.

See also

- [Visual Basic Language Features](#)
- [Programming Concepts](#)
- [Collections](#)
- [Developing Applications with Visual Basic](#)

Accessing computer resources (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

The `My.Computer` object is one of the three central objects in `My`, providing access to information and commonly used functionality. `My.Computer` provides methods, properties, and events for accessing the computer on which the application is running. Its objects include:

- [Audio](#)
- Clipboard ([ClipboardProxy](#))
- [Clock](#)
- [FileSystem](#)
- [Info](#)
- [Keyboard](#)
- [Mouse](#)
- [Network](#)
- [Ports](#)
- Registry ([RegistryProxy](#))

In this section

[Playing Sounds](#)

Lists tasks associated with `My.Computer.Audio`, such as playing a sound in the background.

[Storing Data to and Reading from the Clipboard](#)

Lists tasks associated with `My.Computer.Clipboard`, such as reading data from or writing data to the Clipboard.

[Getting Information about the Computer](#)

Lists tasks associated with `My.Computer.Info`, such as determining a computer's full name or IP addresses.

[Accessing the Keyboard](#)

Lists tasks associated with `My.Computer.Keyboard`, such as determining whether CAPS LOCK is on.

[Accessing the Mouse](#)

Lists tasks associated with `My.Computer.Mouse`, such as determining whether a mouse is present.

[Performing Network Operations](#)

Lists tasks associated with `My.Computer.Network`, such as uploading or downloading files.

[Accessing the Computer's Ports](#)

Lists tasks associated with `My.Computer.Ports`, such as showing available serial ports or sending strings to serial ports.

[Reading from and Writing to the Registry](#)

Lists tasks associated with `My.Computer.Registry`, such as reading data from or writing data to registry keys.

Logging Information from the Application (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

This section contains topics that cover how to log information from your application using the `My.Application.Log` or `My.Log` object, and how to extend the application's logging capabilities.

The `Log` object provides methods for writing information to the application's log listeners, and the `Log` object's advanced `TraceSource` property provides detailed configuration information. The `Log` object is configured by the application's configuration file.

The `My.Log` object is available only for ASP.NET applications. For client applications, use `My.Application.Log`. For more information, see [Log](#).

Tasks

TO	SEE
Write event information to the application's logs.	How to: Write Log Messages
Write exception information to the application's logs.	How to: Log Exceptions
Write trace information to the application's logs when the application starts and shuts down.	How to: Log Messages When the Application Starts or Shuts Down
Configure <code>My.Application.Log</code> to write information to a text file.	How to: Write Event Information to a Text File
Configure <code>My.Application.Log</code> to write information to an event log.	How to: Write to an Application Event Log
Change where <code>My.Application.Log</code> writes information.	Walkthrough: Changing Where My.Application.Log Writes Information
Determine where <code>My.Application.Log</code> writes information.	Walkthrough: Determining Where My.Application.Log Writes Information
Create a custom log listener for <code>My.Application.Log</code> .	Walkthrough: Creating Custom Log Listeners
Filter the output of the <code>My.Application.Log</code> logs.	Walkthrough: Filtering My.Application.Log Output

See also

- [Microsoft.VisualBasic.Logging.Log](#)
- [Working with Application Logs](#)
- [Troubleshooting: Log Listeners](#)

Accessing User Data (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

This section contains topics dealing with the `My.User` object and tasks that you can accomplish with it.

The `My.User` object provides access to information about the logged-on user by returning an object that implements the [IPrincipal](#) interface.

Tasks

TO	SEE
Get the user's login name	Name
Get the user's domain name, if the application uses Windows authentication	CurrentPrincipal
Determine the user's role	IsInRole

See also

- [User](#)

Accessing Application Forms (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

The `My.Forms` object provides an easy way to access an instance of each Windows Form declared in the application's project. You can also use properties of the `My.Application` object to access the application's splash screen and main form, and get a list of the application's open forms.

Tasks

The following table lists examples showing how to access an application's forms.

TO	SEE
Access one form from another form in an application.	My.Forms Object
Display the titles of all the application's open forms.	OpenForms
Update the splash screen with status information as the application starts.	SplashScreen

See also

- [OpenForms](#)
- [SplashScreen](#)
- [My.Forms Object](#)

Accessing Application Web Services (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

The `My.WebServices` object provides an instance of each Web service referenced by the current project. Each instance is instantiated on demand. You can access these Web services through the properties of the `My.WebServices` object. The name of the property is the same as the name of the Web service that the property accesses. Any class that inherits from `SoapHttpClientProtocol` is a Web service.

Tasks

The following table lists possible ways to access Web services referenced by an application.

TO	SEE
Call a Web service	My.WebServices Object
Call a Web service asynchronously and handle an event when it completes	How to: Call a Web Service Asynchronously

See also

- [My.WebServices Object](#)

How to: Call a Web Service Asynchronously (Visual Basic)

3/5/2019 • 2 minutes to read • [Edit Online](#)

This example attaches a handler to a Web service's asynchronous handler event, so that it can retrieve the result of an asynchronous method call. This example used the DemoTemperatureService Web service at <http://www.xmethods.net>.

When you reference a Web service in your project in the Visual Studio Integrated Development Environment (IDE), it is added to the `My.WebServices` object, and the IDE generates a client proxy class to access a specified Web service.

The proxy class allows you to call the Web service methods synchronously, where your application waits for the function to complete. In addition, the proxy creates additional members to help call the method asynchronously. For each Web service function, `NameOfWebServiceFunction`, the proxy creates a `NameOfWebServiceFunction Async` subroutine, a `NameOfWebServiceFunction Completed` event, and a `NameOfWebServiceFunction CompletedEventArgs` class. This example demonstrates how to use the asynchronous members to access the `getTemp` function of the DemoTemperatureService Web service.

NOTE

This code does not work in Web applications, because ASP.NET does not support the `My.WebServices` object.

To call a Web service asynchronously

1. Reference the DemoTemperatureService Web service at <http://www.xmethods.net>. The address is

```
http://www.xmethods.net/sd/2001/DemoTemperatureService.wsdl
```

2. Add an event handler for the `getTempCompleted` event:

```
Private Sub getTempCompletedHandler(ByVal sender As Object,  
    ByVal e As net.xmethods.www.getTempCompletedEventArgs)  
  
    MsgBox("Temperature: " & e.Result)  
End Sub
```

NOTE

You cannot use the `Handles` statement to associate an event handler with the `My.WebServices` object's events.

3. Add a field to track if the event handler has been added to the `getTempCompleted` event:

```
Private handlerAttached As Boolean = False
```

4. Add a method to add the event handler to the `getTempCompleted` event, if necessary, and to call the `getTempAsync` method:

```
Sub CallGetTempAsync(ByVal zipCode As Integer)
    If Not handlerAttached Then
        AddHandler My.WebServices.
            TemperatureService.getTempCompleted,
            AddressOf Me.TS_getTempCompleted
        handlerAttached = True
    End If
    My.WebServices.TemperatureService.getTempAsync(zipCode)
End Sub
```

To call the `getTemp` Web method asynchronously, call the `CallGetTempAsync` method. When the Web method finishes, its return value is passed to the `getTempCompletedHandler` event handler.

See also

- [Accessing Application Web Services](#)
- [My.WebServices Object](#)

Accessing application settings (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

This section contains topics describing the `My.Settings` object and the tasks it enables you to accomplish.

My.Settings

The properties of the `My.Settings` object provide access to your application's settings. To add or remove settings, use the **Settings** pane of the **Project Designer**.

The methods of the `My.Settings` object allow you to save the current user settings or revert the user settings to the last saved values.

Tasks

The following table lists examples showing how to access an application's forms.

TO	SEE
Update the value of a user setting	How to: Change User Settings in Visual Basic
Display application and user settings in a property grid	How to: Create Property Grids for User Settings in Visual Basic
Save updated user setting values	How to: Persist User Settings in Visual Basic
Determine the values of user settings	How to: Read Application Settings in Visual Basic

See also

- [Managing Application Settings \(.NET\)](#)
- [My.Settings Object](#)

2 minutes to read

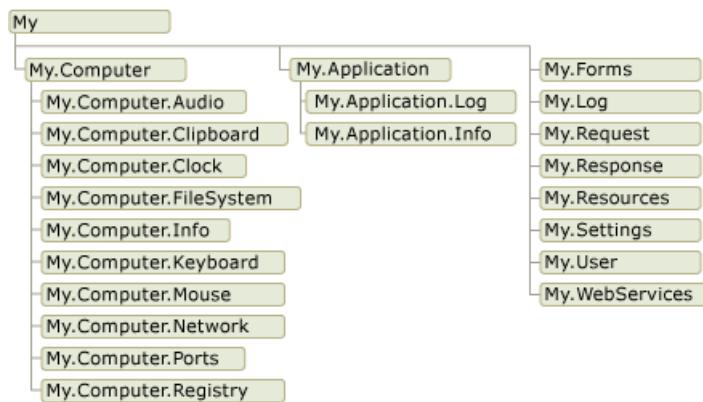
Development with My (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic provides new features for rapid application development that improve productivity and ease of use while delivering power. One of these features, called `My`, provides access to information and default object instances that are related to the application and its run-time environment. This information is organized in a format that is discoverable through IntelliSense and logically delineated according to use.

Top-level members of `My` are exposed as objects. Each object behaves similarly to a namespace or a class with `Shared` members, and it exposes a set of related members.

This table shows the top-level `My` objects and their relationship to each other.



In This Section

[Performing Tasks with My.Application, My.Computer, and My.User](#)

Describes the three central `My` objects, `My.Application`, `My.Computer`, and `My.User`, which provide access to information and functionality.

[Default Object Instances Provided by My.Forms and My.WebServices](#)

Describes the `My.Forms` and `My.WebServices` objects, which provide access to forms, data sources, and XML Web services used by your application.

[Rapid Application Development with My.Resources and My.Settings](#)

Describes the `My.Resources` and `My.Settings` objects, which provide access to an application's resources and settings.

[Overview of the Visual Basic Application Model](#)

Describes the Visual Basic Application Startup/Shutdown model.

[How My Depends on Project Type](#)

Gives details on which `My` features are available in different project types.

See also

- [ApplicationBase](#)
- [Computer](#)
- [User](#)
- [My.Forms Object](#)
- [My.WebServices Object](#)

- How My Depends on Project Type

Performing Tasks with My.Application, My.Computer, and My.User (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

The three central `My` objects that provide access to information and commonly used functionality are `My.Application` (`ApplicationBase`), `My.Computer` (`Computer`), and `My.User` (`User`). You can use these objects to access information that is related to the current application, the computer that the application is installed on, or the current user of the application, respectively.

My.Application, My.Computer, and My.User

The following examples demonstrate how information can be retrieved using `My`.

```
' Displays a message box that shows the full command line for the
' application.
Dim args As String = ""
For Each arg As String In My.Application.CommandLineArgs
    args &= arg & " "
Next
MsgBox(args)
```

```
' Gets a list of subfolders in a folder
My.Computer.FileSystem.GetDirectories(
    My.Computer.FileSystem.SpecialDirectories.MyDocuments, True, "*Logs*")
```

In addition to retrieving information, the members exposed through these three objects also allow you to execute methods related to that object. For instance, you can access a variety of methods to manipulate files or update the registry through `My.Computer`.

File I/O is significantly easier and faster with `My`, which includes a variety of methods and properties for manipulating files, directories, and drives. The `TextFieldParser` object allows you to read from large structured files that have delimited or fixed-width fields. This example opens the `TextFieldParser` `reader` and uses it to read from `C:\TestFolder1\test1.txt`.

```
Dim reader =
    My.Computer.FileSystem.OpenTextFieldParser("C:\TestFolder1\test1.txt")
reader.TextFieldType = Microsoft.VisualBasic.FileIO.FieldType.Delimited
reader.Delimiters = New String() {","}
Dim currentRow As String()
While Not reader.EndOfData
    Try
        currentRow = reader.ReadFields()
        Dim currentField As String
        For Each currentField In currentRow
            MsgBox(currentField)
        Next
    Catch ex As Microsoft.VisualBasic.FileIO.MalformedLineException
        MsgBox("Line " & ex.Message &
            "is not valid and will be skipped.")
    End Try
End While
```

`My.Application` allows you to change the culture for your application. The following example demonstrates how this method can be called.

```
' Changes the current culture for the application to Jamaican English.  
My.Application.ChangeCulture("en-JM")
```

See also

- [ApplicationBase](#)
- [Computer](#)
- [User](#)
- [How My Depends on Project Type](#)

Default Object Instances Provided by My.Forms and My.WebServices (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

The [My.Forms](#) and [My.WebServices](#) objects provide access to forms, data sources, and XML Web services used by your application. They do this by providing collections of *default instances* of each of these objects.

Default Instances

A default instance is an instance of the class that is provided by the runtime and does not need to be declared and instantiated using the `Dim` and `New` statements. The following example demonstrates how you might have declared and instantiated an instance of a [Form](#) class called `Form1`, and how you are now able to get a default instance of this [Form](#) class through `My.Forms`.

```
' The old method of declaration and instantiation
Dim myForm As New Form1
myForm.show()
```

```
' With My.Forms, you can directly call methods on the default
' instance()
My.Forms.Form1.Show()
```

The `My.Forms` object returns a collection of default instances for every [Form](#) class that exists in your project. Similarly, `My.WebServices` provides a default instance of the proxy class for every Web service that you have created a reference to in your application.

See also

- [My.Forms Object](#)
- [My.WebServices Object](#)
- [How My Depends on Project Type](#)

Rapid Application Development with My.Resources and My.Settings (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

The `My.Resources` object provides access to the application's resources and allows you to dynamically retrieve resources for your application.

Retrieving Resources

A number of resources such as audio files, icons, images, and strings can be retrieved through the `My.Resources` object. For example, you can access the application's culture-specific resource files. The following example sets the icon of the form to the icon named `Form1Icon` stored in the application's resource file.

```
Sub SetFormIcon()
    Me.Icon = My.Resources.Form1Icon
End Sub
```

The `My.Resources` object exposes only global resources. It does not provide access to resource files associated with forms. You must access the form resources from the form.

Similarly, the `My.Settings` object provides access to the application's settings and allows you to dynamically store and retrieve property settings and other information for your application. For more information, see [My.Resources Object](#) and [My.Settings Object](#).

See also

- [My.Resources Object](#)
- [My.Settings Object](#)
- [Accessing Application Settings](#)

Overview of the Visual Basic Application Model

4/28/2019 • 3 minutes to read • [Edit Online](#)

Visual Basic provides a well-defined model for controlling the behavior of Windows Forms applications: the Visual Basic Application model. This model includes events for handling the application's startup and shutdown, as well as events for catching unhandled exceptions. It also provides support for developing single-instance applications. The application model is extensible, so developers that need more control can customize its overridable methods.

Uses for the Application Model

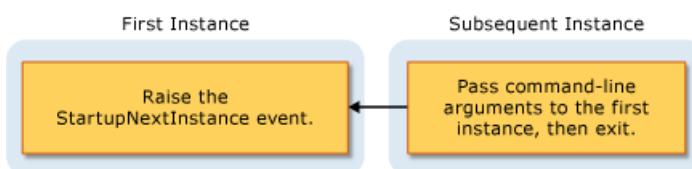
A typical application needs to perform tasks when it starts up and shuts down. For example, when it starts up, the application can display a splash screen, make database connections, load a saved state, and so on. When the application shuts down, it can close database connections, save the current state, and so on. In addition, the application can execute specific code when the application shuts down unexpectedly, such as during an unhandled exception.

The Visual Basic Application model makes it easy to create a *single-instance* application. A single-instance application differs from a normal application in that only one instance of the application can be running at a time. An attempt to launch another instance of a single-instance application results in the original instance being notified—by means of the `StartupNextInstance` event—that another launch attempt was made. The notification includes the subsequent instance's command-line arguments. The subsequent instance of the application is then closed before any initialization can occur.

A single-instance application starts and checks whether it is the first instance or a subsequent instance of the application:

- If it is the first instance, it starts as usual.
- Each subsequent attempt to start the application, while the first instance runs, results in very different behavior. The subsequent attempt notifies the first instance about the command-line arguments, and then immediately exits. The first instance handles the `StartupNextInstance` event to determine what the subsequent instance's command-line arguments were, and continues to run.

This diagram shows how a subsequent instance signals the first instance:



By handling the `StartupNextInstance` event, you can control how your single-instance application behaves. For example, Microsoft Outlook typically runs as a single-instance application; when Outlook is running and you attempt to start Outlook again, focus shifts to the original instance but another instance does not open.

Events in the Application Model

The following events are found in the application model:

- **Application startup.** The application raises the `Startup` event when it starts. By handling this event, you can add code that initializes the application before the main form is loaded. The `Startup` event also provides for canceling execution of the application during that phase of the startup process, if desired.

You can configure the application to show a splash screen while the application startup code runs. By default, the application model suppresses the splash screen when either the `/nosplash` or `-nosplash` command-line argument is used.

- **Single-instance applications.** The [StartupNextInstance](#) event is raised when a subsequent instance of a single-instance application starts. The event passes the command-line arguments of the subsequent instance.
- **Unhandled exceptions.** If the application encounters an unhandled exception, it raises the [UnhandledException](#) event. Your handler for that event can examine the exception and determine whether to continue execution.

The `UnhandledException` event is not raised in some circumstances. For more information, see [UnhandledException](#).

- **Network-connectivity changes.** If the computer's network availability changes, the application raises the [NetworkAvailabilityChanged](#) event.

The `NetworkAvailabilityChanged` event is not raised in some circumstances. For more information, see [NetworkAvailabilityChanged](#).

- **Application shut down.** The application provides the [Shutdown](#) event to signal when it is about to shut down. In that event handler, you can make sure that the operations your application needs to perform—closing and saving, for example—are completed. You can configure your application to shut down when the main form closes, or to shut down only when all forms close.

Availability

By default, the Visual Basic Application model is available for Windows Forms projects. If you configure the application to use a different startup object, or start the application code with a custom `Sub Main`, then that object or class may need to provide an implementation of the [WindowsFormsApplicationBase](#) class to use the application model. For information about changing the startup object, see [Application Page, Project Designer \(Visual Basic\)](#).

See also

- [WindowsFormsApplicationBase](#)
- [Startup](#)
- [StartupNextInstance](#)
- [UnhandledException](#)
- [Shutdown](#)
- [NetworkAvailabilityChanged](#)
- [WindowsFormsApplicationBase](#)
- [Extending the Visual Basic Application Model](#)

How My Depends on Project Type (Visual Basic)

4/24/2019 • 2 minutes to read • [Edit Online](#)

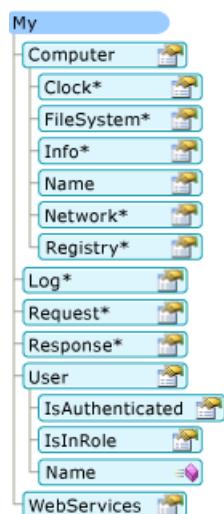
`My` exposes only those objects required by a particular project type. For example, the `My.Forms` object is available in a Windows Forms application but not available in a console application. This topic describes which `My` objects are available in different project types.

My in Windows Applications and Web Sites

`My` exposes only objects that are useful in the current project type; it suppresses objects that are not applicable. For example, the following image shows the `My` object model in a Windows Forms project.



In a Web site project, `My` exposes objects that are relevant to a Web developer (such as the `My.Request` and `My.Response` objects) while suppressing objects that are not relevant (such as the `My.Forms` object). The following image shows the `My` object model in a Web site project:



Project Details

The following table shows which `My` objects are enabled by default for eight project types: Windows application, class Library, console application, Windows control library, Web control library, Windows service, empty, and Web site.

There are three versions of the `My.Application` object, two versions of the `My.Computer` object, and two versions of `My.User` object; details about these versions are given in the footnotes after the table.

MY OBJECT	WINDOWS APPLICATION	CLASS LIBRARY	CONSOLE APPLICATION	WINDOWS CONTROL LIBRARY	WEB CONTROL LIBRARY	WINDOWS SERVICE	EMPTY	WEB SITE
<code>My.Application</code> ¹	<code>Yes</code> ²	<code>Yes</code> ²	<code>Yes</code> ³	<code>Yes</code> ²	No	<code>Yes</code> ³	No	No
<code>My.Computer</code>	<code>Yes</code> ⁴	<code>Yes</code> ⁴	<code>Yes</code> ⁴	<code>Yes</code> ⁴	<code>Yes</code> ⁵	<code>Yes</code> ⁴	No	<code>Yes</code> ⁵
<code>My.Forms</code>	<code>Yes</code>	No	No	<code>Yes</code>	No	No	No	No
<code>My.Log</code>	No	No	No	No	No	No	No	<code>Yes</code>
<code>My.Request</code>	No	No	No	No	No	No	No	<code>Yes</code>
<code>My.Resources</code>	<code>Yes</code>	<code>Yes</code>	<code>Yes</code>	<code>Yes</code>	<code>Yes</code>	<code>Yes</code>	No	No
<code>My.Response</code>	No	No	No	No	No	No	No	<code>Yes</code>
<code>My.Settings</code>	<code>Yes</code>	<code>Yes</code>	<code>Yes</code>	<code>Yes</code>	<code>Yes</code>	<code>Yes</code>	No	No
<code>My.User</code>	<code>Yes</code> ⁶	<code>Yes</code> ⁶	<code>Yes</code> ⁶	<code>Yes</code> ⁶	<code>Yes</code> ⁷	<code>Yes</code> ⁶	No	<code>Yes</code> ⁷
<code>My.WebServices</code>	<code>Yes</code>	<code>Yes</code>	<code>Yes</code>	<code>Yes</code>	<code>Yes</code>	<code>Yes</code>	No	No

¹ Windows Forms version of `My.Application`. Derives from the console version (see Note 3); adds support for interacting with the application's windows and provides the Visual Basic Application model.

² Library version of `My.Application`. Provides the basic functionality needed by an application: provides members for writing to the application log and accessing application information.

³ Console version of `My.Application`. Derives from the library version (see Note 2), and adds additional members for accessing the application's command-line arguments and ClickOnce deployment information.

⁴ Windows version of `My.Computer`. Derives from the Server version (see Note 5), and provides access to useful objects on a client machine, such as the keyboard, screen, and mouse.

⁵ Server version of `My.Computer`. Provides basic information about the computer, such as the name, access to the clock, and so on.

⁶ Windows version of `My.User`. This object is associated with the thread's current identity.

⁷ Web version of `My.User`. This object is associated with the user identity of the application's current HTTP request.

See also

- [ApplicationBase](#)
- [Computer](#)

- [Log](#)
- [User](#)
- [Customizing Which Objects are Available in My](#)
- [Conditional Compilation](#)
- [/define \(Visual Basic\)](#)
- [My.Forms Object](#)
- [My.Request Object](#)
- [My.Response Object](#)
- [My.WebServices Object](#)

Accessing data in Visual Basic applications

10/17/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic includes several new features to assist in developing applications that access data. Data-bound forms for Windows applications are created by dragging items from the [Data Sources Window](#) onto the form. You bind controls to data by dragging items from the **Data Sources Window** onto existing controls.

Related sections

[Accessing Data in Visual Studio](#)

Provides links to pages that discuss incorporating data access functionality into your applications.

[Visual Studio data tools for .NET](#)

Provides links to pages on creating applications that work with data, using Visual Studio.

[LINQ](#)

Provides links to topics that describe how to use LINQ with Visual Basic.

[LINQ to SQL](#)

Provides information about LINQ to SQL. Includes programming examples.

[LINQ to SQL Tools in Visual Studio](#)

Provides links to topics about how to create a [LINQ to SQL](#) object model in applications.

[Work with datasets in n-tier applications](#)

Provides links to topics about how to create multitiered data applications.

[Add new connections](#)

Provides links to pages on connecting your application to data with design-time tools and ADO.NET connection objects, using Visual Studio.

[Dataset Tools in Visual Studio](#)

Provides links to pages describing how to load data into datasets and how to execute SQL statements and stored procedures.

[Bind controls to data in Visual Studio](#)

Provides links to pages that explain how to display data on Windows Forms through data-bound controls.

[Edit Data in Datasets](#)

Provides links to pages describing how to manipulate the data in the data tables of a dataset.

[Validate data in datasets](#)

Provides links to pages describing how to add validation to a dataset during column and row changes.

[Save data back to the database](#)

Provides links to pages explaining how to send updated data from an application to the database.

[ADO.NET](#)

Describes the ADO.NET classes, which expose data-access services to the .NET Framework programmer.

[Data in Office Solutions](#)

Contains links to pages that explain how data works in Office solutions, including information about schema-oriented programming, data caching, and server-side data access.

Creating and Using Components in Visual Basic

5/14/2019 • 2 minutes to read • [Edit Online](#)

A *component* is a class that implements the [System.ComponentModel.IComponent](#) interface or that derives directly or indirectly from a class that implements [IComponent](#). A .NET Framework component is an object that is reusable, can interact with other objects, and provides control over external resources and design-time support.

An important feature of components is that they are designable, which means that a class that is a component can be used in the Visual Studio Integrated Development Environment. A component can be added to the Toolbox, dragged and dropped onto a form, and manipulated on a design surface. Notice that base design-time support for components is built into the .NET Framework; a component developer does not have to do any additional work to take advantage of the base design-time functionality.

A *control* is similar to a component, as both are designable. However, a control provides a user interface, while a component does not. A control must derive from one of the base control classes: [Control](#) or [Control](#).

When to Create a Component

If your class will be used on a design surface (such as the Windows Forms or Web Forms Designer) but has no user interface, it should be a component and implement [IComponent](#), or derive from a class that directly or indirectly implements [IComponent](#).

The [Component](#) and [MarshalByValueComponent](#) classes are base implementations of the [IComponent](#) interface. The main difference between these classes is that the [Component](#) class is marshaled by reference, while [IComponent](#) is marshaled by value. The following list provides broad guidelines for implementers.

- If your component needs to be marshaled by reference, derive from [Component](#).
- If your component needs to be marshaled by value, derive from [MarshalByValueComponent](#).
- If your component cannot derive from one of the base implementations due to single inheritance, implement [IComponent](#).

Component Classes

The [System.ComponentModel](#) namespace provides classes that are used to implement the run-time and design-time behavior of components and controls. This namespace includes the base classes and interfaces for implementing attributes and type converters, binding to data sources, and licensing components.

The core component classes are:

- [Component](#). A base implementation for the [IComponent](#) interface. This class enables object sharing between applications.
- [MarshalByValueComponent](#). A base implementation for the [IComponent](#) interface.
- [Container](#). The base implementation for the [.IContainer](#) interface. This class encapsulates zero or more components.

Some of the classes used for component licensing are:

- [License](#). The abstract base class for all licenses. A license is granted to a specific instance of a component.
- [LicenseManager](#). Provides properties and methods to add a license to a component and to manage a

[LicenseProvider](#).

- [LicenseProvider](#). The abstract base class for implementing a license provider.
- [LicenseProviderAttribute](#). Specifies the [LicenseProvider](#) class to use with a class.

Classes commonly used for describing and persisting components.

- [TypeDescriptor](#). Provides information about the characteristics for a component, such as its attributes, properties, and events.
- [EventDescriptor](#). Provides information about an event.
- [PropertyDescriptor](#). Provides information about a property.

Related Sections

[Troubleshooting Control and Component Authoring](#)

Explains how to fix common problems.

See also

- [How to: Access Design-Time Support in Windows Forms](#)

Windows Forms Application Basics (Visual Basic)

10/17/2019 • 6 minutes to read • [Edit Online](#)

An important part of Visual Basic is the ability to create Windows Forms applications that run locally on users' computers. You can use Visual Studio to create the application and user interface using Windows Forms. A Windows Forms application is built on classes from the [System.Windows.Forms](#) namespace.

Designing Windows Forms Applications

You can create Windows Forms and Windows service applications with Visual Studio. For more information, see the following topics:

- [Getting Started with Windows Forms](#). Provides information on how to create and program Windows Forms.
- [Windows Forms Controls](#). Collection of topics detailing the use of Windows Forms controls.
- [Windows Service Applications](#). Lists topics that explain how to create Windows services.

Building Rich, Interactive User Interfaces

Windows Forms is the smart-client component of the .NET Framework, a set of managed libraries that enable common application tasks such as reading and writing to the file system. Using a development environment like Visual Studio, you can create Windows Forms applications that display information, request input from users, and communicate with remote computers over a network.

In Windows Forms, a form is a visual surface on which you display information to the user. You commonly build Windows Forms applications by placing controls on forms and developing responses to user actions, such as mouse clicks or key presses. A *control* is a discrete user interface (UI) element that displays data or accepts data input.

Events

When a user does something to your form or one of its controls, it generates an event. Your application reacts to these events by using code, and processes the events when they occur. For more information, see [Creating Event Handlers in Windows Forms](#).

Controls

Windows Forms contains a variety of controls that you can place on forms: controls that display text boxes, buttons, drop-down boxes, radio buttons, and even Web pages. For a list of all the controls you can use on a form, see [Controls to Use on Windows Forms](#). If an existing control does not meet your needs, Windows Forms also supports creating your own custom controls using the [UserControl](#) class.

Windows Forms has rich UI controls that emulate features in high-end applications like Microsoft Office. Using the [ToolStrip](#) and [MenuStrip](#) control, you can create toolbars and menus that contain text and images, display submenus, and host other controls such as text boxes and combo boxes.

With the Visual Studio drag-and-drop forms designer, you can easily create Windows Forms applications: just select the controls with your cursor and place them where you want on the form. The designer provides tools such as grid lines and "snap lines" to take the hassle out of aligning controls. And whether you use Visual Studio or compile at the command line, you can use the [FlowLayoutPanel](#), [TableLayoutPanel](#) and [SplitContainer](#) controls to create advanced form layouts with minimal time and effort.

Custom UI Elements

Finally, if you must create your own custom UI elements, the [System.Drawing](#) namespace contains all of the classes you need to render lines, circles, and other shapes directly on a form.

For step-by-step information about using these features, see the following Help topics.

TO	SEE
Create a new Windows Forms application with Visual Studio	Tutorial 1: Create a picture viewer
Use controls on forms	How to: Add Controls to Windows Forms
Create graphics with System.Drawing	Getting Started with Graphics Programming
Create custom controls	How to: Inherit from the UserControl Class

Displaying and Manipulating Data

Many applications must display data from a database, XML file, XML Web service, or other data source. Windows Forms provides a flexible control called the [DataGridView](#) control for rendering such tabular data in a traditional row and column format, so that every piece of data occupies its own cell. Using [DataGridView](#) you can customize the appearance of individual cells, lock arbitrary rows and columns in place, and display complex controls inside cells, among other features.

Connecting to data sources over a network is a simple task with Windows Forms smart clients. The [BindingSource](#) component, new with Windows Forms in Visual Studio 2005 and the .NET Framework 2.0, represents a connection to a data source, and exposes methods for binding data to controls, navigating to the previous and next records, editing records, and saving changes back to the original source. The [BindingNavigator](#) control provides a simple interface over the [BindingSource](#) component for users to navigate between records.

Data-Bound Controls

You can create data-bound controls easily using the Data Sources window, which displays data sources such as databases, Web services, and objects in your project. You can create data-bound controls by dragging items from this window onto forms in your project. You can also data-bind existing controls to data by dragging objects from the Data Sources window onto existing controls.

Settings

Another type of data binding you can manage in Windows Forms is settings. Most smart-client applications must retain some information about their run-time state, such as the last-known size of forms, and retain user-preference data, such as default locations for saved files. The application-settings feature addresses these requirements by providing an easy way to store both types of settings on the client computer. Once defined using either Visual Studio or a code editor, these settings are persisted as XML and automatically read back into memory at run time.

For step-by-step information about using these features, see the following Help topics.

TO	SEE
Use the BindingSource component	How to: Bind Windows Forms Controls with the BindingSource Component Using the Designer
Work with ADO.NET data sources	How to: Sort and Filter ADO.NET Data with the Windows Forms BindingSource Component

TO	SEE
Use the Data Sources window	Walkthrough: Displaying Data on a Windows Form

Deploying Applications to Client Computers

Once you have written your application, you must send it to your users so that they can install and run it on their own client computers. Using the ClickOnce technology, you can deploy your applications from within Visual Studio by using just a few clicks and provide users with a URL pointing to your application on the Web. ClickOnce manages all of the elements and dependencies in your application and ensures that the application is properly installed on the client computer.

ClickOnce applications can be configured to run only when the user is connected to the network, or to run both online and offline. When you specify that an application should support offline operation, ClickOnce adds a link to your application in the user's **Start** menu, so that the user can open it without using the URL.

When you update your application, you publish a new deployment manifest and a new copy of your application to your Web server. ClickOnce detects that there is an update available and upgrades the user's installation; no custom programming is required to update old assemblies.

For a full introduction to ClickOnce, see [ClickOnce Security and Deployment](#). For step-by-step information about using these features, see the following Help topics:

TO	SEE
Deploy an application with ClickOnce	How to: Publish a ClickOnce Application using the Publish Wizard Walkthrough: Manually Deploying a ClickOnce Application
Update a ClickOnce deployment	How to: Manage Updates for a ClickOnce Application
Manage security with ClickOnce	How to: Enable ClickOnce Security Settings

Other Controls and Features

There are many other features in Windows Forms that make implementing common tasks fast and easy, such as support for creating dialog boxes, printing, adding Help and documentation, and localizing your application to multiple languages. In addition, Windows Forms relies on the robust security system of the .NET Framework, enabling you to release more secure applications to your customers.

For step-by-step information about using these features, see the following Help topics:

TO	SEE
Print the contents of a form	How to: Print Graphics in Windows Forms How to: Print a Multi-Page Text File in Windows Forms
Learn more about Windows Forms security	Security in Windows Forms Overview

See also

- [WindowsFormsApplicationBase](#)

- [Windows Forms Overview](#)
- [My.Forms Object](#)

Customizing Projects and Extending My with Visual Basic

4/3/2019 • 2 minutes to read • [Edit Online](#)

You can customize project templates to provide additional `My` objects. This makes it easy for other developers to find and use your objects.

In This Section

[Extending the My Namespace in Visual Basic](#)

Describes how to add custom members and values to the `My` namespace in Visual Basic.

[Packaging and Deploying Custom My Extensions](#)

Describes how to publish custom `My` namespace extensions by using Visual Studio templates.

[Extending the Visual Basic Application Model](#)

Describes how to specify your own extensions to the application model by overriding members of the `WindowsFormsApplicationBase` class.

[Customizing Which Objects are Available in My](#)

Describes how to control which `My` objects are enabled by setting your project's `_MYTYPE` conditional-compilation constant.

Related Sections

[Development with My](#)

Describes which `My` objects are available in different project types by default.

[Overview of the Visual Basic Application Model](#)

Describes Visual Basic's model for controlling the behavior of Windows Forms applications.

[How My Depends on Project Type](#)

Describes which `My` objects are available in different project types by default.

[Conditional Compilation](#)

Discusses how the compiler uses conditional-compilation to select particular sections of code to compile and exclude other sections.

[ApplicationBase](#)

Describes the `My` object that provides properties, methods, and events related to the current application.

See also

- [Developing Applications with Visual Basic](#)

Extending the My Namespace in Visual Basic

4/28/2019 • 8 minutes to read • [Edit Online](#)

The `My` namespace in Visual Basic exposes properties and methods that enable you to easily take advantage of the power of the .NET Framework. The `My` namespace simplifies common programming problems, often reducing a difficult task to a single line of code. Additionally, the `My` namespace is fully extensible so that you can customize the behavior of `My` and add new services to its hierarchy to adapt to specific application needs. This topic discusses both how to customize existing members of the `My` namespace and how to add your own custom classes to the `My` namespace.

Topic Contents

- [Customizing Existing My Namespace Members](#)
- [Adding Members to My Objects](#)
- [Adding Custom Objects to the My Namespace](#)
- [Adding Members to the My Namespace](#)
- [Adding Events to Custom My Objects](#)
- [Design Guidelines](#)
- [Designing Class Libraries for My](#)
- [Packaging and Deploying Extensions](#)

Customizing Existing My Namespace Members

The `My` namespace in Visual Basic exposes frequently used information about your application, your computer, and more. For a complete list of the objects in the `My` namespace, see [My Reference](#). You may have to customize existing members of the `My` namespace so that they better match the needs of your application. Any property of an object in the `My` namespace that is not read-only can be set to a custom value.

For example, assume that you frequently use the `My.User` object to access the current security context for the user running your application. However, your company uses a custom user object to expose additional information and capabilities for users within the company. In this scenario, you can replace the default value of the `My.User.CurrentPrincipal` property with an instance of your own custom principal object, as shown in the following example.

```
My.User.CurrentPrincipal = CustomPrincipal
```

Setting the `CurrentPrincipal` property on the `My.User` object changes the identity under which the application runs. The `My.User` object, in turn, returns information about the newly specified user.

Adding Members to My Objects

The types returned from `My.Application` and `My.Computer` are defined as `Partial` classes. Therefore, you can extend the `My.Application` and `My.Computer` objects by creating a `Partial` class named `MyApplication` or `MyComputer`. The class cannot be a `Private` class. If you specify the class as part of the `My` namespace, you can add properties and methods that will be included with the `My.Application` or `My.Computer` objects.

For example, the following example adds a property named `DnsServerIPAddresses` to the `My.Computer` object.

```
Imports System.Net.NetworkInformation

Namespace My

    Partial Class MyComputer
        Friend ReadOnly Property DnsServerIPAddresses() As IPAddressCollection
            Get
                Dim dnsAddressList As IPAddressCollection = Nothing

                For Each adapter In System.Net.NetworkInformation.
                    NetworkInterface.GetAllNetworkInterfaces()

                        Dim adapterProperties = adapter.GetIPProperties()
                        Dim dnsServers As IPAddressCollection = adapterProperties.DnsAddresses
                        If dnsAddressList Is Nothing Then
                            dnsAddressList = dnsServers
                        Else
                            dnsAddressList.Union(dnsServers)
                        End If
                    Next adapter

                Return dnsAddressList
            End Get
        End Property
    End Class

End Namespace
```

Adding Custom Objects to the My Namespace

Although the `My` namespace provides solutions for many common programming tasks, you may encounter tasks that the `My` namespace does not address. For example, your application might access custom directory services for user data, or your application might use assemblies that are not installed by default with Visual Basic. You can extend the `My` namespace to include custom solutions to common tasks that are specific to your environment. The `My` namespace can easily be extended to add new members to meet growing application needs. Additionally, you can deploy your `My` namespace extensions to other developers as a Visual Basic template.

Adding Members to the My Namespace

Because `My` is a namespace like any other namespace, you can add top-level properties to it by just adding a module and specifying a `Namespace` of `My`. Annotate the module with the `HideModuleName` attribute as shown in the following example. The `HideModuleName` attribute ensures that IntelliSense will not display the module name when it displays the members of the `My` namespace.

```
Namespace My
    <HideModuleName()>
    Module MyCustomModule

    End Module
End Namespace
```

To add members to the `My` namespace, add properties as needed to the module. For each property added to the `My` namespace, add a private field of type `ThreadSafeObjectProvider<of T>`, where the type is the type returned by your custom property. This field is used to create thread-safe object instances to be returned by the property by calling the `GetInstance` method. As a result, each thread that is accessing the extended property receives its own instance of the returned type. The following example adds a property named `SampleExtension` that is of type `SampleExtension` to the `My` namespace:

```
Namespace My
<HideModuleName()>
Module MyCustomExtensions
    Private _extension As New ThreadSafeObjectProvider(Of SampleExtension)
    Friend ReadOnly Property SampleExtension() As SampleExtension
        Get
            Return _extension.GetInstance()
        End Get
    End Property
End Module
End Namespace
```

Adding Events to Custom My Objects

You can use the `My.Application` object to expose events for your custom `My` objects by extending the `MyApplication` partial class in the `My` namespace. For Windows-based projects, you can double-click the **My Project** node in for your project in **Solution Explorer**. In the Visual Basic **Project Designer**, click the `Application` tab and then click the `View Application Events` button. A new file that is named `ApplicationEvents.vb` will be created. It contains the following code for extending the `MyApplication` class.

```
Namespace My
    Partial Friend Class MyApplication
    End Class
End Namespace
```

You can add event handlers for your custom `My` objects by adding custom event handlers to the `MyApplication` class. Custom events enable you to add code that will execute when an event handler is added, removed, or the event is raised. Note that the `AddHandler` code for a custom event runs only if code is added by a user to handle the event. For example, consider that the `SampleExtension` object from the previous section has a `Load` event that you want to add a custom event handler for. The following code example shows a custom event handler named `SampleExtensionLoad` that will be invoked when the `My.SampleExtension.Load` event occurs. When code is added to handle the new `My.SampleExtensionLoad` event, the `AddHandler` part of this custom event code is executed. The `MyApplication_SampleExtensionLoad` method is included in the code example to show an example of an event handler that handles the `My.SampleExtensionLoad` event. Note that the `SampleExtensionLoad` event will be available when you select the **My Application Events** option in the left drop-down list above the Code Editor when you are editing the `ApplicationEvents.vb` file.

```
Namespace My
```

```
Partial Friend Class MyApplication

    ' Custom event handler for Load event.
    Private _sampleExtensionHandlers As EventHandler

    Public Custom Event SampleExtensionLoad As EventHandler
        AddHandler(ByVal value As EventHandler)
            ' Warning: This code is not thread-safe. Do not call
            ' this code from multiple concurrent threads.
            If _sampleExtensionHandlers Is Nothing Then
                AddHandler My.SampleExtension.Load, AddressOf OnSampleExtensionLoad
            End If
            _sampleExtensionHandlers =
                System.Delegate.Combine(_sampleExtensionHandlers, value)
        End AddHandler
        RemoveHandler(ByVal value As EventHandler)
            _sampleExtensionHandlers =
                System.Delegate.Remove(_sampleExtensionHandlers, value)
        End RemoveHandler
        RaiseEvent(ByVal sender As Object, ByVal e As EventArgs)
            If _sampleExtensionHandlers IsNot Nothing Then
                _sampleExtensionHandlers.Invoke(sender, e)
            End If
        End RaiseEvent
    End Event

    ' Method called by custom event handler to raise user-defined
    ' event handlers.
    <Global.System.ComponentModel.EditorBrowsable(
        Global.System.ComponentModel.EditorBrowsableState.Advanced)>
    Protected Overridable Sub OnSampleExtensionLoad(
        ByVal sender As Object, ByVal e As EventArgs)
        RaiseEvent SampleExtensionLoad(sender, e)
    End Sub

    ' Event handler to call My.SampleExtensionLoad event.
    Private Sub MyApplication_SampleExtensionLoad(
        ByVal sender As Object, ByVal e As System.EventArgs
    ) Handles Me.SampleExtensionLoad

    End Sub
End Class
End Namespace
```

Design Guidelines

When you develop extensions to the `My` namespace, use the following guidelines to help minimize the maintenance costs of your extension components.

- **Include only the extension logic.** The logic included in the `My` namespace extension should include only the code that is needed to expose the required functionality in the `My` namespace. Because your extension will reside in user projects as source code, updating the extension component incurs a high maintenance cost and should be avoided if possible.
- **Minimize project assumptions.** When you create your extensions of the `My` namespace, do not assume a set of references, project-level imports, or specific compiler settings (for example, `Option Strict` off). Instead, minimize dependencies and fully qualify all type references by using the `Global` keyword. Also, ensure that the extension compiles with `Option Strict` on to minimize errors in the extension.
- **Isolate the extension code.** Placing the code in a single file makes your extension easily deployable as a

Visual Studio item template. For more information, see "Packaging and Deploying Extensions" later in this topic. Placing all the `My` namespace extension code in a single file or a separate folder in a project will also help users locate the `My` namespace extension.

Designing Class Libraries for My

As is the case with most object models, some design patterns work well in the `My` namespace and others do not. When designing an extension to the `My` namespace, consider the following principles:

- **Stateless methods.** Methods in the `My` namespace should provide a complete solution to a specific task. Ensure that the parameter values that are passed to the method provide all the input required to complete the particular task. Avoid creating methods that rely on prior state, such as open connections to resources.
- **Global instances.** The only state that is maintained in the `My` namespace is global to the project. For example, `My.Application.Info` encapsulates state that is shared throughout the application.
- **Simple parameter types.** Keep things simple by avoiding complex parameter types. Instead, create methods that either take no parameter input or that take simple input types such as strings, primitive types, and so on.
- **Factory methods.** Some types are necessarily difficult to instantiate. Providing factory methods as extensions to the `My` namespace enables you to more easily discover and consume types that fall into this category. An example of a factory method that works well is `My.Computer.FileSystem.OpenTextFileReader`. There are several stream types available in the .NET Framework. By specifying text files specifically, the `OpenTextFileReader` helps the user understand which stream to use.

These guidelines do not preclude general design principles for class libraries. Rather, they are recommendations that are optimized for developers who are using Visual Basic and the `My` namespace. For general design principles for creating class libraries, see [Framework Design Guidelines](#).

Packaging and Deploying Extensions

You can include `My` namespace extensions in a Visual Studio project template, or you can package your extensions and deploy them as a Visual Studio item template. When you package your `My` namespace extensions as a Visual Studio item template, you can take advantage of additional capabilities provided by Visual Basic. These capabilities enable you to include an extension when a project references a particular assembly, or enable users to explicitly add your `My` namespace extension by using the **My Extensions** page of the Visual Basic Project Designer.

For details about how to deploy `My` namespace extensions, see [Packaging and Deploying Custom My Extensions](#).

See also

- [Packaging and Deploying Custom My Extensions](#)
- [Extending the Visual Basic Application Model](#)
- [Customizing Which Objects are Available in My](#)
- [My Extensions Page, Project Designer](#)
- [Application Page, Project Designer \(Visual Basic\)](#)
- [Partial](#)

Package and deploy custom My extensions (Visual Basic)

8/19/2018 • 4 minutes to read • [Edit Online](#)

Visual Basic provides an easy way for you to deploy your custom `My` namespace extensions by using Visual Studio templates. If you are creating a project template for which your `My` extensions are an integral part of the new project type, you can just include your custom `My` extension code with the project when you export the template. For more information about exporting project templates, see [How to: Create Project Templates](#).

If your custom `My` extension is in a single code file, you can export the file as an item template that users can add to any type of Visual Basic project. You can then customize the item template to enable additional capabilities and behavior for your custom `My` extension in a Visual Basic project. Those capabilities include the following:

- Allowing users to manage your custom `My` extension from the **My Extensions** page of the Visual Basic Project Designer.
- Automatically adding your custom `My` extension when a reference to a specified assembly is added to a project.
- Hiding the `My` extension item template in the **Add Item** dialog box so that it is not included in the list of project items.

This topic discusses how to package a custom `My` extension as a hidden item template that can be managed from the **My Extensions** page of the Visual Basic Project Designer. The custom `My` extension can also be added automatically when a reference to a specified assembly is added to a project.

Create a My namespace extension

The first step in creating a deployment package for a custom `My` extension is to create the extension as a single code file. For details and guidance about how to create a custom `My` extension, see [Extending the My Namespace in Visual Basic](#).

Export a My namespace extension as an item template

After you have a code file that includes your `My` namespace extension, you can export the code file as a Visual Studio item template. For instructions on how to export a file as a Visual Studio item template, see [How to: Create Item Templates](#).

NOTE

If your `My` namespace extension has a dependency on a particular assembly, you can customize your item template to automatically install your `My` namespace extension when a reference to that assembly is added. As a result, you will want to exclude that assembly reference when you export the code file as a Visual Studio item template.

Customize the item template

You can enable your item template to be managed from the **My Extensions** page of the Visual Basic Project Designer. You can also enable the item template to be added automatically when a reference to a specified assembly is added to a project. To enable these customizations, you will add a new file, called the `CustomData` file,

to your template, and then add a new element to the XML in your .vstemplate file.

Add the CustomData file

The CustomData file is a text file that has a file name extension of .CustomData (the file name can be set to any value meaningful to your template) and that contains XML. The XML in the CustomData file instructs Visual Basic to include your `My` extension when users use the **My Extensions** page of the Visual Basic Project Designer. You can optionally add the `<AssemblyFullName>` attribute to your CustomData file XML. This instructs Visual Basic to automatically install your custom `My` extension when a reference to a particular assembly is added to the project. You can use any text editor or XML editor to create the CustomData file, and then add it to your item template's compressed folder (.zip file).

For example, the following XML shows the contents of a CustomData file that will add the template item to the My Extensions folder of a Visual Basic project when a reference to the Microsoft.VisualBasic.PowerPacks.Vs.dll assembly is added to the project.

```
<VBMyExtensionTemplate  
ID="Microsoft.VisualBasic.Samples.MyExtensions.MyPrinterInfo"  
Version="1.0.0.0"  
AssemblyFullName="Microsoft.VisualBasic.PowerPacks-vs"  
/>
```

The CustomData file contains a `<VBMyExtensionTemplate>` element that has attributes as listed in the following table.

ATTRIBUTE	DESCRIPTION
<code>ID</code>	Required. A unique identifier for the extension. If the extension that has this ID has already been added to the project, the user will not be prompted to add it again.
<code>Version</code>	Required. A version number for the item template.
<code>AssemblyFullName</code>	Optional. An assembly name. When a reference to this assembly is added to the project, the user will be prompted to add the <code>My</code> extension from this item template.

Add the `<CustomDataSignature>` element to the .vstemplate file

To identify your Visual Studio item template as a `My` namespace extension, you must also modify the .vstemplate file for your item template. You must add a `<CustomDataSignature>` element to the `<TemplateData>` element. The `<CustomDataSignature>` element must contain the text `Microsoft.VisualBasic.MyExtension`, as shown in the following example.

```
<CustomDataSignature>Microsoft.VisualBasic.MyExtension</CustomDataSignature>
```

You cannot modify files in a compressed folder (.zip file) directly. You must copy the .vstemplate file from the compressed folder, modify it, and then replace the .vstemplate file in the compressed folder with your updated copy.

The following example shows the contents of a .vstemplate file that has the `<CustomDataSignature>` element added.

```
<VSTemplate Version="2.0.0" xmlns="http://schemas.microsoft.com/developer/vstemplate/2005" Type="Item">
<TemplateData>
  <DefaultName>MyCustomExtensionModule.vb</DefaultName>
  <Name>MyPrinterInfo</Name>
  <Description>Custom My Extensions Item Template</Description>
  <ProjectType>VisualBasic</ProjectType>
  <SortOrder>10</SortOrder>
  <Icon>_TemplateIcon.ico</Icon>
  <CustomDataSignature>Microsoft.VisualBasic.MyExtension</CustomDataSignature>
</TemplateData>
<TemplateContent>
  <References />
  <ProjectItem SubType="Code"
    TargetFileName="$fileinputname$.vb"
    ReplaceParameters="true"
    >MyCustomExtensionModule.vb</ProjectItem>
</TemplateContent>
</VSTemplate>
```

Install the template

To install the template, you can copy the compressed folder (.zip file) to the Visual Basic item templates folder. By default, user item templates are located in %USERPROFILE%\Documents\Visual Studio <Version>\Templates\ItemTemplates\Visual Basic. Alternatively, you can publish the template as a Visual Studio Installer (.vsi) file.

See also

- [Extending the My Namespace in Visual Basic](#)
- [Extending the Visual Basic Application Model](#)
- [Customizing Which Objects are Available in My](#)
- [My Extensions Page, Project Designer](#)

Extending the Visual Basic Application Model

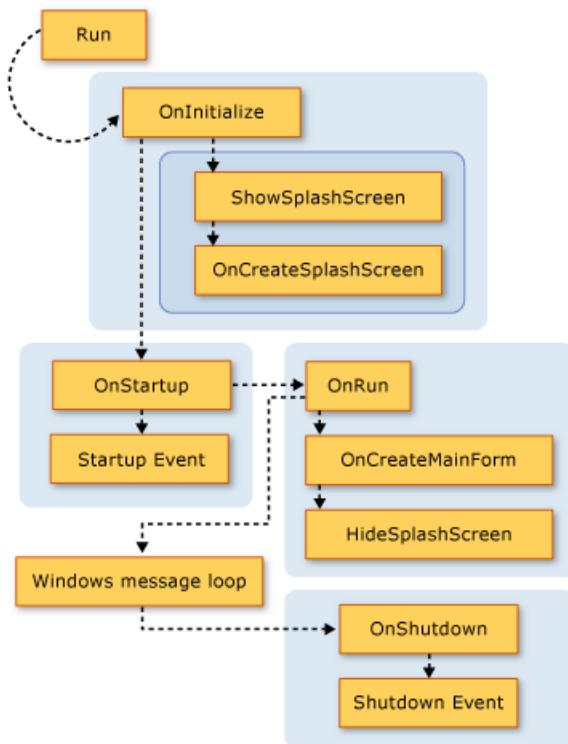
10/18/2019 • 4 minutes to read • [Edit Online](#)

You can add functionality to the application model by overriding the `Overridable` members of the `WindowsFormsApplicationBase` class. This technique allows you to customize the behavior of the application model and add calls to your own methods as the application starts up and shuts down.

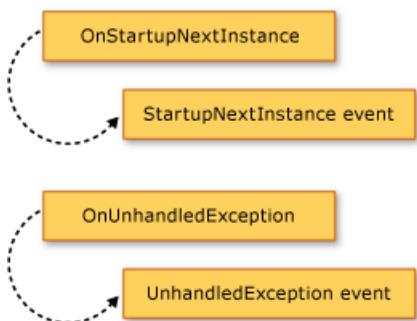
Visual Overview of the Application Model

This section visually presents the sequence of function calls in the Visual Basic Application Model. The next section describes the purpose of each function in detail.

The following graphic shows the application model call sequence in a normal Visual Basic Windows Forms application. The sequence starts when the `Sub Main` procedure calls the `Run` method.



The Visual Basic Application Model also provides the `StartupNextInstance` and `UnhandledException` events. The following graphics show the mechanism for raising these events.



Overriding the Base Methods

The `Run` method defines the order in which the `Application` methods run. By default, the `Sub Main` procedure for

a Windows Forms application calls the [Run](#) method.

If the application is a normal application (multiple-instance application), or the first instance of a single-instance application, the [Run](#) method executes the [Overridable](#) methods in the following order:

1. [OnInitialize](#). By default, this method sets the visual styles, text display styles, and current principal for the main application thread (if the application uses Windows authentication), and calls [ShowSplashScreen](#) if neither `/nosplash` nor `-nosplash` is used as a command-line argument.

The application startup sequence is canceled if this function returns `False`. This can be useful if there are circumstances in which the application should not run.

The [OnInitialize](#) method calls the following methods:

- a. [ShowSplashScreen](#). Determines if the application has a splash screen defined and if it does, displays the splash screen on a separate thread.

The [ShowSplashScreen](#) method contains the code that displays the splash screen for at least the number of milliseconds specified by the [MinimumSplashScreenDisplayTime](#) property. To use this functionality, you must add the splash screen to your application using the **Project Designer** (which sets the `My.Application.MinimumSplashScreenDisplayTime` property to two seconds), or set the `My.Application.MinimumSplashScreenDisplayTime` property in a method that overrides the [OnInitialize](#) or [OnCreateSplashScreen](#) method. For more information, see [MinimumSplashScreenDisplayTime](#).

- b. [OnCreateSplashScreen](#). Allows a designer to emit code that initializes the splash screen.

By default, this method does nothing. If you select a splash screen for your application in the Visual Basic **Project Designer**, the designer overrides the [OnCreateSplashScreen](#) method with a method that sets the [SplashScreen](#) property to a new instance of the splash-screen form.

2. [OnStartup](#). Provides an extensibility point for raising the [Startup](#) event. The application startup sequence stops if this function returns `False`.

By default, this method raises the [Startup](#) event. If the event handler sets the [Cancel](#) property of the event argument to `True`, the method returns `False` to cancel the application startup.

3. [OnRun](#). Provides the starting point for when the main application is ready to start running, after the initialization is done.

By default, before it enters the Windows Forms message loop, this method calls the [OnCreateMainForm](#) (to create the application's main form) and [HideSplashScreen](#) (to close the splash screen) methods:

- a. [OnCreateMainForm](#). Provides a way for a designer to emit code that initializes the main form.

By default, this method does nothing. However, when you select a main form for your application in the Visual Basic **Project Designer**, the designer overrides the [OnCreateMainForm](#) method with a method that sets the [MainForm](#) property to a new instance of the main form.

- b. [HideSplashScreen](#). If application has a splash screen defined and it is open, this method closes the splash screen.

By default, this method closes the splash screen.

4. [OnStartupNextInstance](#). Provides a way to customize how a single-instance application behaves when another instance of the application starts.

By default, this method raises the [StartupNextInstance](#) event.

5. [OnShutdown](#). Provides an extensibility point for raising the [Shutdown](#) event. This method does not run if an unhandled exception occurs in the main application.

By default, this method raises the [Shutdown](#) event.

6. [OnUnhandledException](#). Executed if an unhandled exception occurs in any of the above listed methods.

By default, this method raises the [UnhandledException](#) event as long as a debugger is not attached and the application is handling the [UnhandledException](#) event.

If the application is a single-instance application, and the application is already running, the subsequent instance of the application calls the [OnStartupNextInstance](#) method on the original instance of the application, and then exits.

The [OnStartupNextInstance\(StartupNextInstanceEventArgs\)](#) constructor calls the [UseCompatibleTextRendering](#) property to determine which text rendering engine to use for the application's forms. By default, the [UseCompatibleTextRendering](#) property returns `False`, indicating that the GDI text rendering engine be used, which is the default in Visual Basic 2005 and later versions. You can override the [UseCompatibleTextRendering](#) property to return `True`, which indicates that the GDI+ text rendering engine be used, which is the default in Visual Basic .NET 2002 and Visual Basic .NET 2003.

Configuring the Application

As a part of the Visual Basic Application model, the [WindowsFormsApplicationBase](#) class provides protected properties that configure the application. These properties should be set in the constructor of the implementing class.

In a default Windows Forms project, the **Project Designer** creates code to set the properties with the designer settings. The properties are used only when the application is starting; setting them after the application starts has no effect.

PROPERTY	DETERMINES	SETTING IN THE APPLICATION PANE OF THE PROJECT DESIGNER
IsSingleInstance	Whether the application runs as a single-instance or multiple-instance application.	Make single instance application check box
EnableVisualStyles	If the application will use visual styles that match Windows XP.	Enable XP visual styles check box
SaveMySettingsOnExit	If application automatically saves application's user-settings changes when the application exits.	Save My.Settings on Shutdown check box
ShutdownStyle	What causes the application to terminate, such as when the startup form closes or when the last form closes.	Shutdown mode list

See also

- [ApplicationBase](#)
- [Startup](#)
- [StartupNextInstance](#)
- [UnhandledException](#)
- [Shutdown](#)
- [NetworkAvailabilityChanged](#)

- [Overview of the Visual Basic Application Model](#)
- [Application Page, Project Designer \(Visual Basic\)](#)

Customizing Which Objects are Available in My (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

This topic describes how you can control which `My` objects are enabled by setting your project's `_MYTYPE` conditional-compilation constant. The Visual Studio Integrated Development Environment (IDE) keeps the `_MYTYPE` conditional-compilation constant for a project in sync with the project's type.

Predefined `_MYTYPE` Values

You must use the `/define` compiler option to set the `_MYTYPE` conditional-compilation constant. When specifying your own value for the `_MYTYPE` constant, you must enclose the string value in backslash/quotation mark (\") sequences. For example, you could use:

```
/define:_MYTYPE=\"WindowsForms\"
```

This table shows what the `_MYTYPE` conditional-compilation constant is set to for several project types.

PROJECT TYPE	_MYTYPE VALUE
Class Library	"Windows"
Console Application	"Console"
Web	"Web"
Web Control Library	"WebControl"
Windows Application	"WindowsForms"
Windows Application, when starting with custom <code>Sub Main</code>	"WindowsFormsWithCustomSubMain"
Windows Control Library	"Windows"
Windows Service	"Console"
Empty	"Empty"

NOTE

All conditional-compilation string comparisons are case-sensitive, regardless of how the `Option Compare` statement is set.

Dependent `_MY` Compilation Constants

The `_MYTYPE` conditional-compilation constant, in turn, controls the values of several other `_MY` compilation constants:

<code>_MYTYPE</code>	<code>_MYAPPLICATIONTYPE</code>	<code>_MYCOMPUTERTYPE</code>	<code>_MYFORMS</code>	<code>_MYUSERTYPE</code>	<code>_MYWEBSERVICES</code>
"Console"	"Console"	"Windows"	Undefined	"Windows"	TRUE
"Custom"	Undefined	Undefined	Undefined	Undefined	Undefined
"Empty"	Undefined	Undefined	Undefined	Undefined	Undefined
"Web"	Undefined	"Web"	FALSE	"Web"	FALSE
"WebControl"	Undefined	"Web"	FALSE	"Web"	TRUE
"Windows" or ""	"Windows"	"Windows"	Undefined	"Windows"	TRUE
"WindowsForms"	"WindowsForms"	"Windows"	TRUE	"Windows"	TRUE
"WindowsForms WithCustomSub Main"	"Console"	"Windows"	TRUE	"Windows"	TRUE

By default, undefined conditional-compilation constants resolve to `FALSE`. You can specify values for the undefined constants when compiling your project to override the default behavior.

NOTE

When `_MYTYPE` is set to "Custom", the project contains the `My` namespace, but it contains no objects. However, setting `_MYTYPE` to "Empty" prevents the compiler from adding the `My` namespace and its objects.

This table describes the effects of the predefined values of the `_MY` compilation constants.

CONSTANT	MEANING
<code>_MYAPPLICATIONTYPE</code>	<p>Enables <code>My.Application</code>, if the constant is "Console," "Windows," or "WindowsForms":</p> <ul style="list-style-type: none"> - The "Console" version derives from <code>ConsoleApplicationBase</code>, and has fewer members than the "Windows" version. - The "Windows" version derives from <code>ApplicationBase</code>, and has fewer members than the "WindowsForms" version. - The "WindowsForms" version of <code>My.Application</code> derives from <code>WindowsFormsApplicationBase</code>. If the <code>TARGET</code> constant is defined to be "winexe", then the class includes a <code>Sub Main</code> method.
<code>_MYCOMPUTERTYPE</code>	<p>Enables <code>My.Computer</code>, if the constant is "Web" or "Windows":</p> <ul style="list-style-type: none"> - The "Web" version derives from <code>ServerComputer</code>, and has fewer members than the "Windows" version. - The "Windows" version of <code>My.Computer</code> derives from <code>Computer</code>.
<code>_MYFORMS</code>	Enables <code>My.Forms</code> , if the constant is <code>TRUE</code> .

CONSTANT	MEANING
_MYUSERTYPE	Enables <code>My.User</code> , if the constant is "Web" or "Windows": - The "Web" version of <code>My.User</code> is associated with the user identity of the current HTTP request. - The "Windows" version of <code>My.User</code> is associated with the thread's current principal.
_MYWEBSERVICES	Enables <code>My.WebServices</code> , if the constant is <code>TRUE</code> .
_MYTYPE	Enables <code>My.Log</code> , <code>My.Request</code> , and <code>My.Response</code> , if the constant is "Web".

See also

- [ApplicationBase](#)
- [Computer](#)
- [Log](#)
- [User](#)
- [How My Depends on Project Type](#)
- [Conditional Compilation](#)
- [/define \(Visual Basic\)](#)
- [My.Forms Object](#)
- [My.Request Object](#)
- [My.Response Object](#)
- [My.WebServices Object](#)

Programming Concepts (Visual Basic)

3/8/2019 • 2 minutes to read • [Edit Online](#)

This section explains programming concepts in the Visual Basic language.

In this section

TITLE	DESCRIPTION
Asynchronous Programming with Async and Await (Visual Basic)	Describes how to write asynchronous solutions by using <code>Async</code> and <code>Await</code> keywords. Includes a walkthrough.
Attributes overview (Visual Basic)	Discusses how to provide additional information about programming elements such as types, fields, methods, and properties by using attributes.
Caller Information (Visual Basic)	Describes how to obtain information about the caller of a method. This information includes the file path and the line number of the source code and the member name of the caller.
Collections (Visual Basic)	Describes some of the types of collections provided by the .NET Framework. Demonstrates how to use simple collections and collections of key/value pairs.
Covariance and Contravariance (Visual Basic)	Shows how to enable implicit conversion of generic type parameters in interfaces and delegates.
Expression Trees (Visual Basic)	Explains how you can use expression trees to enable dynamic modification of executable code.
Iterators (Visual Basic)	Describes iterators, which are used to step through collections and return elements one at a time.
Language-Integrated Query (LINQ) (Visual Basic)	Discusses the powerful query capabilities in the language syntax of Visual Basic, and the model for querying relational databases, XML documents, datasets, and in-memory collections.
Object-Oriented Programming (Visual Basic)	Describes common object-oriented concepts, including encapsulation, inheritance, and polymorphism.
Reflection (Visual Basic)	Explains how to use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties.
Serialization (Visual Basic)	Describes key concepts in binary, XML, and SOAP serialization.

Related sections

Assemblies in .NET	Describes how to create and use assemblies.
Performance Tips	Discusses several basic rules that may help you increase the performance of your application.

Asynchronous programming with Async and Await (Visual Basic)

10/18/2019 • 16 minutes to read • [Edit Online](#)

You can avoid performance bottlenecks and enhance the overall responsiveness of your application by using asynchronous programming. However, traditional techniques for writing asynchronous applications can be complicated, making them difficult to write, debug, and maintain.

Visual Studio 2012 introduced a simplified approach, async programming, that leverages asynchronous support in the .NET Framework 4.5 and higher as well as in the Windows Runtime. The compiler does the difficult work that the developer used to do, and your application retains a logical structure that resembles synchronous code. As a result, you get all the advantages of asynchronous programming with a fraction of the effort.

This topic provides an overview of when and how to use async programming and includes links to support topics that contain details and examples.

Async improves responsiveness

Asynchrony is essential for activities that are potentially blocking, such as when your application accesses the web. Access to a web resource sometimes is slow or delayed. If such an activity is blocked within a synchronous process, the entire application must wait. In an asynchronous process, the application can continue with other work that doesn't depend on the web resource until the potentially blocking task finishes.

The following table shows typical areas where asynchronous programming improves responsiveness. The listed APIs from the .NET Framework 4.5 and the Windows Runtime contain methods that support async programming.

APPLICATION AREA	SUPPORTING APIs THAT CONTAIN ASYNC METHODS
Web access	HttpClient , SyndicationClient
Working with files	StorageFile , StreamWriter , StreamReader , XmlReader
Working with images	MediaCapture , BitmapEncoder , BitmapDecoder
WCF programming	Synchronous and Asynchronous Operations

Asynchrony proves especially valuable for applications that access the UI thread because all UI-related activity usually shares one thread. If any process is blocked in a synchronous application, all are blocked. Your application stops responding, and you might conclude that it has failed when instead it's just waiting.

When you use asynchronous methods, the application continues to respond to the UI. You can resize or minimize a window, for example, or you can close the application if you don't want to wait for it to finish.

The async-based approach adds the equivalent of an automatic transmission to the list of options that you can choose from when designing asynchronous operations. That is, you get all the benefits of traditional asynchronous programming but with much less effort from the developer.

Async methods are easier to write

The `Async` and `Await` keywords in Visual Basic are the heart of async programming. By using those two keywords, you can use resources in the .NET Framework or the Windows Runtime to create an asynchronous method almost as easily as you create a synchronous method. Asynchronous methods that you define by using `Async` and `Await` are referred to as `async` methods.

The following example shows an `async` method. Almost everything in the code should look completely familiar to you. The comments call out the features that you add to create the asynchrony.

You can find a complete Windows Presentation Foundation (WPF) example file at the end of this topic, and you can download the sample from [Async Sample: Example from "Asynchronous Programming with Async and Await"](#).

```
' Three things to note about writing an Async Function:  
' - The function has an Async modifier.  
' - Its return type is Task or Task(Of T). (See "Return Types" section.)  
' - As a matter of convention, its name ends in "Async".  
Async Function AccessTheWebAsync() As Task(Of Integer)  
    Using client As New HttpClient()  
        ' Call and await separately.  
        ' - AccessTheWebAsync can do other things while GetStringAsync is also running.  
        ' - getStringTask stores the task we get from the call to GetStringAsync.  
        ' - Task(Of String) means it is a task which returns a String when it is done.  
        Dim getStringTask As Task(Of String) =  
            client.GetStringAsync("https://docs.microsoft.com/dotnet")  
        ' You can do other work here that doesn't rely on the string from GetStringAsync.  
        DoIndependentWork()  
        ' The Await operator suspends AccessTheWebAsync.  
        ' - AccessTheWebAsync does not continue until getStringTask is complete.  
        ' - Meanwhile, control returns to the caller of AccessTheWebAsync.  
        ' - Control resumes here when getStringTask is complete.  
        ' - The Await operator then retrieves the String result from getStringTask.  
        Dim urlContents As String = Await getStringTask  
        ' The Return statement specifies an Integer result.  
        ' A method which awaits AccessTheWebAsync receives the Length value.  
        Return urlContents.Length  
  
    End Using  
  
End Function
```

If `AccessTheWebAsync` doesn't have any work that it can do between calling `GetStringAsync` and awaiting its completion, you can simplify your code by calling and awaiting in the following single statement.

```
Dim urlContents As String = Await client.GetStringAsync()
```

The following characteristics summarize what makes the previous example an `async` method:

- The method signature includes an `Async` modifier.
 - The name of an `async` method, by convention, ends with an "Async" suffix.
 - The return type is one of the following types:
 - `Task<TResult>` if your method has a return statement in which the operand has type `TResult`.
 - `Task` if your method has no return statement or has a return statement with no operand.
 - `Sub` if you're writing an `async` event handler.
- For more information, see "Return Types and Parameters" later in this topic.
- The method usually includes at least one `await` expression, which marks a point where the method can't continue until the awaited asynchronous operation is complete. In the meantime, the method is suspended,

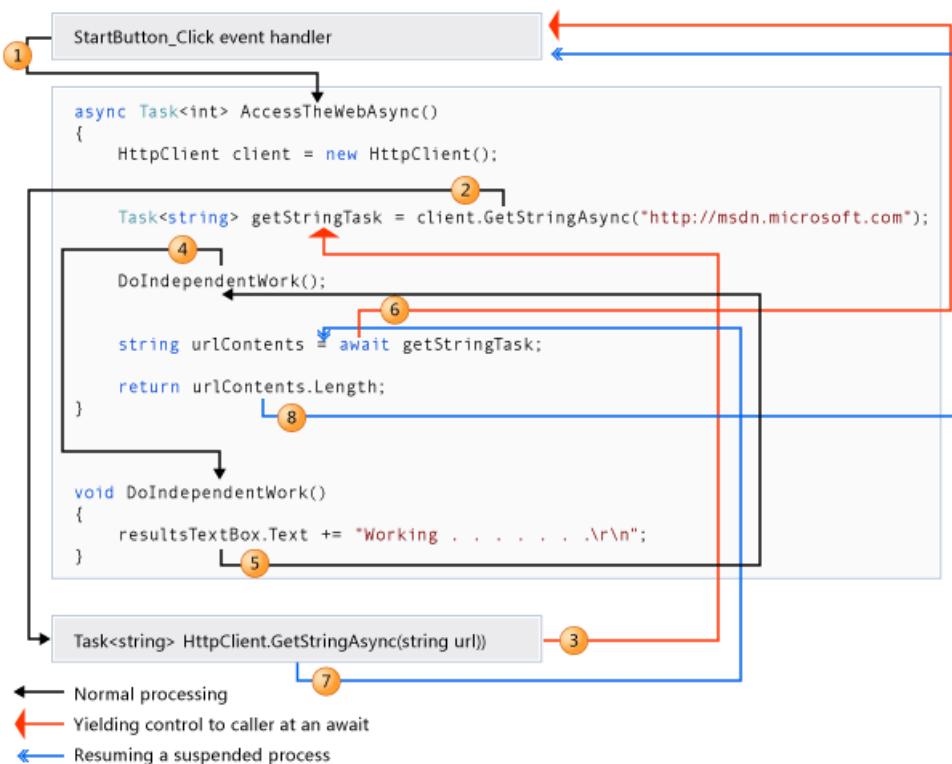
and control returns to the method's caller. The next section of this topic illustrates what happens at the suspension point.

In async methods, you use the provided keywords and types to indicate what you want to do, and the compiler does the rest, including keeping track of what must happen when control returns to an await point in a suspended method. Some routine processes, such as loops and exception handling, can be difficult to handle in traditional asynchronous code. In an async method, you write these elements much as you would in a synchronous solution, and the problem is solved.

For more information about asynchrony in previous versions of the .NET Framework, see [TPL and Traditional .NET Framework Asynchronous Programming](#).

What happens in an Async method

The most important thing to understand in asynchronous programming is how the control flow moves from method to method. The following diagram leads you through the process:



The numbers in the diagram correspond to the following steps:

1. An event handler calls and awaits the `AccessTheWebAsync` async method.
2. `AccessTheWebAsync` creates an `HttpClient` instance and calls the `GetStringAsync` asynchronous method to download the contents of a website as a string.
3. Something happens in `GetStringAsync` that suspends its progress. Perhaps it must wait for a website to download or some other blocking activity. To avoid blocking resources, `GetStringAsync` yields control to its caller, `AccessTheWebAsync`.

`GetStringAsync` returns a `Task<TResult>` where `TResult` is a string, and `AccessTheWebAsync` assigns the task to the `getStringTask` variable. The task represents the ongoing process for the call to `GetStringAsync`, with a commitment to produce an actual string value when the work is complete.

4. Because `getStringTask` hasn't been awaited yet, `AccessTheWebAsync` can continue with other work that doesn't depend on the final result from `GetStringAsync`. That work is represented by a call to the `DoIndependentWork` method.

synchronous method `DoIndependentWork`.

5. `DoIndependentWork` is a synchronous method that does its work and returns to its caller.
6. `AccessTheWebAsync` has run out of work that it can do without a result from `getStringTask`. `AccessTheWebAsync` next wants to calculate and return the length of the downloaded string, but the method can't calculate that value until the method has the string.

Therefore, `AccessTheWebAsync` uses an await operator to suspend its progress and to yield control to the method that called `AccessTheWebAsync`. `AccessTheWebAsync` returns a `Task<int>` (`Task(Of Integer)` in Visual Basic) to the caller. The task represents a promise to produce an integer result that's the length of the downloaded string.

NOTE

If `GetStringAsync` (and therefore `getStringTask`) is complete before `AccessTheWebAsync` awaits it, control remains in `AccessTheWebAsync`. The expense of suspending and then returning to `AccessTheWebAsync` would be wasted if the called asynchronous process (`getStringTask`) has already completed and `AccessTheWebSync` doesn't have to wait for the final result.

Inside the caller (the event handler in this example), the processing pattern continues. The caller might do other work that doesn't depend on the result from `AccessTheWebAsync` before awaiting that result, or the caller might await immediately. The event handler is waiting for `AccessTheWebAsync`, and `AccessTheWebAsync` is waiting for `GetStringAsync`.

7. `GetStringAsync` completes and produces a string result. The string result isn't returned by the call to `GetStringAsync` in the way that you might expect. (Remember that the method already returned a task in step 3.) Instead, the string result is stored in the task that represents the completion of the method, `getStringTask`. The await operator retrieves the result from `getStringTask`. The assignment statement assigns the retrieved result to `urlContents`.
8. When `AccessTheWebAsync` has the string result, the method can calculate the length of the string. Then the work of `AccessTheWebAsync` is also complete, and the waiting event handler can resume. In the full example at the end of the topic, you can confirm that the event handler retrieves and prints the value of the length result.

If you are new to asynchronous programming, take a minute to consider the difference between synchronous and asynchronous behavior. A synchronous method returns when its work is complete (step 5), but an async method returns a task value when its work is suspended (steps 3 and 6). When the async method eventually completes its work, the task is marked as completed and the result, if any, is stored in the task.

For more information about control flow, see [Control Flow in Async Programs \(Visual Basic\)](#).

API Async Methods

You might be wondering where to find methods such as `GetStringAsync` that support async programming. The .NET Framework 4.5 or higher contains many members that work with `Async` and `Await`. You can recognize these members by the "Async" suffix that's attached to the member name and a return type of `Task` or `Task<TResult>`. For example, the `System.IO.Stream` class contains methods such as `CopyToAsync`, `ReadAsync`, and `WriteAsync` alongside the synchronous methods `CopyTo`, `Read`, and `Write`.

The Windows Runtime also contains many methods that you can use with `Async` and `Await` in Windows apps. For more information and example methods, see [Call asynchronous APIs in C# or Visual Basic](#), [Asynchronous programming \(Windows Runtime apps\)](#), and [WhenAny: Bridging between the .NET Framework and the Windows Runtime](#).

Threads

Async methods are intended to be non-blocking operations. An `Await` expression in an async method doesn't block the current thread while the awaited task is running. Instead, the expression signs up the rest of the method as a continuation and returns control to the caller of the async method.

The `Async` and `Await` keywords don't cause additional threads to be created. Async methods don't require multi-threading because an async method doesn't run on its own thread. The method runs on the current synchronization context and uses time on the thread only when the method is active. You can use `Task.Run` to move CPU-bound work to a background thread, but a background thread doesn't help with a process that's just waiting for results to become available.

The async-based approach to asynchronous programming is preferable to existing approaches in almost every case. In particular, this approach is better than `BackgroundWorker` for I/O-bound operations because the code is simpler and you don't have to guard against race conditions. In combination with `Task.Run`, async programming is better than `BackgroundWorker` for CPU-bound operations because async programming separates the coordination details of running your code from the work that `Task.Run` transfers to the threadpool.

Async and Await

If you specify that a method is an async method by using an `Async` modifier, you enable the following two capabilities.

- The marked async method can use `Await` to designate suspension points. The await operator tells the compiler that the async method can't continue past that point until the awaited asynchronous process is complete. In the meantime, control returns to the caller of the async method.

The suspension of an async method at an `Await` expression doesn't constitute an exit from the method, and `Finally` blocks don't run.

- The marked async method can itself be awaited by methods that call it.

An async method typically contains one or more occurrences of an `Await` operator, but the absence of `Await` expressions doesn't cause a compiler error. If an async method doesn't use an `Await` operator to mark a suspension point, the method executes as a synchronous method does, despite the `Async` modifier. The compiler issues a warning for such methods.

`Async` and `Await` are contextual keywords. For more information and examples, see the following topics:

- [Async](#)
- [Await Operator](#)

Return types and parameters

In .NET Framework programming, an async method typically returns a `Task` or a `Task<TResult>`. Inside an async method, an `Await` operator is applied to a task that's returned from a call to another async method.

You specify `Task<TResult>` as the return type if the method contains a `Return` statement that specifies an operand of type `TResult`.

You use `Task` as the return type if the method has no return statement or has a return statement that doesn't return an operand.

The following example shows how you declare and call a method that returns a `Task<TResult>` or a `Task`:

```

' Signature specifies Task(Of Integer)
Async Function TaskOfTResult_MethodAsync() As Task(Of Integer)

    Dim hours As Integer
    ' ...
    ' Return statement specifies an integer result.
    Return hours
End Function

' Calls to TaskOfTResult_MethodAsync
Dim returnedTaskTResult As Task(Of Integer) = TaskOfTResult_MethodAsync()
Dim intResult As Integer = Await returnedTaskTResult
' or, in a single statement
Dim intResult As Integer = Await TaskOfTResult_MethodAsync()

' Signature specifies Task
Async Function Task_MethodAsync() As Task

    ' ...
    ' The method has no return statement.
End Function

' Calls to Task_MethodAsync
Task returnedTask = Task_MethodAsync()
Await returnedTask
' or, in a single statement
Await Task_MethodAsync()

```

Each returned task represents ongoing work. A task encapsulates information about the state of the asynchronous process and, eventually, either the final result from the process or the exception that the process raises if it doesn't succeed.

An async method can also be a `Sub` method. This return type is used primarily to define event handlers, where a return type is required. Async event handlers often serve as the starting point for async programs.

An async method that's a `Sub` procedure can't be awaited, and the caller can't catch any exceptions that the method throws.

An async method can't declare `ByRef` parameters, but the method can call methods that have such parameters.

For more information and examples, see [Async Return Types \(Visual Basic\)](#). For more information about how to catch exceptions in async methods, see [Try...Catch...Finally Statement](#).

Asynchronous APIs in Windows Runtime programming have one of the following return types, which are similar to tasks:

- `IAsyncOperation<TResult>`, which corresponds to `Task<TResult>`
- `IAsyncAction`, which corresponds to `Task`
- `IAsyncActionWithProgress<TProgress>`
- `IAsyncOperationWithProgress<TResult, TProgress>`

For more information and an example, see [Call asynchronous APIs in C# or Visual Basic](#).

Naming convention

By convention, you append "Async" to the names of methods that have an `Async` modifier.

You can ignore the convention where an event, base class, or interface contract suggests a different name. For example, you shouldn't rename common event handlers, such as `Button1_Click`.

Related topics and samples (Visual Studio)

TITLE	DESCRIPTION	SAMPLE
Walkthrough: Accessing the Web by Using Async and Await (Visual Basic)	Shows how to convert a synchronous WPF solution to an asynchronous WPF solution. The application downloads a series of websites.	Async Sample: Accessing the Web Walkthrough
How to: Extend the Async Walkthrough by Using Task.WhenAll (Visual Basic)	Adds <code>Task.WhenAll</code> to the previous walkthrough. The use of <code>WhenAll</code> starts all the downloads at the same time.	
How to: Make Multiple Web Requests in Parallel by Using Async and Await (Visual Basic)	Demonstrates how to start several tasks at the same time.	Async Sample: Make Multiple Web Requests in Parallel
Async Return Types (Visual Basic)	Illustrates the types that async methods can return and explains when each type is appropriate.	
Control Flow in Async Programs (Visual Basic)	Traces in detail the flow of control through a succession of await expressions in an asynchronous program.	Async Sample: Control Flow in Async Programs
Fine-Tuning Your Async Application (Visual Basic)	<p>Shows how to add the following functionality to your async solution:</p> <ul style="list-style-type: none"> - Cancel an Async Task or a List of Tasks (Visual Basic) - Cancel Async Tasks after a Period of Time (Visual Basic) - Cancel Remaining Async Tasks after One Is Complete (Visual Basic) - Start Multiple Async Tasks and Process Them As They Complete (Visual Basic) 	Async Sample: Fine Tuning Your Application
Handling Reentrancy in Async Apps (Visual Basic)	Shows how to handle cases in which an active asynchronous operation is restarted while it's running.	
WhenAny: Bridging between the .NET Framework and the Windows Runtime	Shows how to bridge between Task types in the .NET Framework and IAsyncOperations in the Windows Runtime so that you can use <code>WhenAny</code> with a Windows Runtime method.	Async Sample: Bridging between .NET and Windows Runtime (AsTask and WhenAny)
Async Cancellation: Bridging between the .NET Framework and the Windows Runtime	Shows how to bridge between Task types in the .NET Framework and IAsyncOperations in the Windows Runtime so that you can use <code>CancellationTokenSource</code> with a Windows Runtime method.	Async Sample: Bridging between .NET and Windows Runtime (AsTask & Cancellation)
Using Async for File Access (Visual Basic)	Lists and demonstrates the benefits of using async and await to access files.	

TITLE	DESCRIPTION	SAMPLE
Task-based Asynchronous Pattern (TAP)	Describes a new pattern for asynchrony in the .NET Framework. The pattern is based on the <code>Task</code> and <code>Task<TResult></code> types.	
Async Videos on Channel 9	Provides links to a variety of videos about async programming.	

Complete Example

The following code is the `MainWindow.xaml.vb` file from the Windows Presentation Foundation (WPF) application that this topic discusses. You can download the sample from [Async Sample: Example from "Asynchronous Programming with Async and Await"](#).

```

Imports System.Net.Http

' Example that demonstrates Asynchronous Programming with Async and Await.
' It uses HttpClient.GetStringAsync to download the contents of a website.
' Sample Output:
' Working . . . . .
'
' Length of the downloaded string: 39678.

Class MainWindow

    ' Mark the event handler with Async so you can use Await in it.
    Private Async Sub StartButton_Click(sender As Object, e As RoutedEventArgs)

        ' Call and await immediately.
        ' StartButton_Click suspends until AccessTheWebAsync is done.
        Dim contentLength As Integer = Await AccessTheWebAsync()

        ResultsTextBox.Text &= $"{vbCrLf}Length of the downloaded string: {contentLength}.{vbCrLf}"

    End Sub

    ' Three things to note about writing an Async Function:
    ' - The function has an Async modifier.
    ' - Its return type is Task or Task(Of T). (See "Return Types" section.)
    ' - As a matter of convention, its name ends in "Async".
    Async Function AccessTheWebAsync() As Task(Of Integer)

        Using client As New HttpClient()

            ' Call and await separately.
            ' - AccessTheWebAsync can do other things while GetStringAsync is also running.
            ' - getStringTask stores the task we get from the call to GetStringAsync.
            ' - Task(Of String) means it is a task which returns a String when it is done.
            Dim getStringTask As Task(Of String) =
                client.GetStringAsync("https://docs.microsoft.com/dotnet")

            ' You can do other work here that doesn't rely on the string from GetStringAsync.
            DoIndependentWork()

            ' The Await operator suspends AccessTheWebAsync.
            ' - AccessTheWebAsync does not continue until getStringTask is complete.
            ' - Meanwhile, control returns to the caller of AccessTheWebAsync.
            ' - Control resumes here when getStringTask is complete.
            ' - The Await operator then retrieves the String result from getStringTask.
            Dim urlContents As String = Await getStringTask

            ' The Return statement specifies an Integer result.
            ' A method which awaits AccessTheWebAsync receives the Length value.
            Return urlContents.Length

        End Using

    End Function

    Sub DoIndependentWork()
        ResultsTextBox.Text &= $"Working . . . . .{vbCrLf}"
    End Sub

End Class

```

See also

- [Await Operator](#)
- [Async](#)

Attributes overview (Visual Basic)

10/18/2019 • 4 minutes to read • [Edit Online](#)

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called *reflection*. For more information, see [Reflection \(Visual Basic\)](#).

Attributes have the following properties:

- Attributes add metadata to your program. *Metadata* is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required. For more information, see [Creating Custom Attributes \(Visual Basic\)](#).
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Your program can examine its own metadata or the metadata in other programs by using reflection. For more information, see [Accessing Attributes by Using Reflection \(Visual Basic\)](#).

Using Attributes

Attributes can be placed on most any declaration, though a specific attribute might restrict the types of declarations on which it is valid. In Visual Basic, an attribute is enclosed in angle brackets (< >). It must appear immediately before the element to which it is applied, on the same line.

In this example, the [SerializableAttribute](#) attribute is used to apply a specific characteristic to a class:

```
<System.Serializable()> Public Class SampleClass  
    ' Objects of this type can be serialized.  
End Class
```

A method with the attribute [DllImportAttribute](#) is declared like this:

```
Imports System.Runtime.InteropServices  
  
<System.Runtime.InteropServicesDllImport("user32.dll")>  
Sub SampleMethod()  
End Sub
```

More than one attribute can be placed on a declaration:

```
Imports System.Runtime.InteropServices
```

```
Sub MethodA(<[In](), Out()> ByVal x As Double)
End Sub
Sub MethodB(<Out(), [In]()> ByVal x As Double)
End Sub
```

Some attributes can be specified more than once for a given entity. An example of such a multiuse attribute is [ConditionalAttribute](#):

```
<Conditional("DEBUG"), Conditional("TEST1")>
Sub TraceMethod()
End Sub
```

NOTE

By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET Framework. However, you do not need to specify the attribute suffix when using attributes in code. For example, `[DllImport]` is equivalent to `[DllImportAttribute]`, but `DllImportAttribute` is the attribute's actual name in the .NET Framework.

Attribute Parameters

Many attributes have parameters, which can be positional, unnamed, or named. Any positional parameters must be specified in a certain order and cannot be omitted; named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, these three attributes are equivalent:

```
<DllImport("user32.dll")>
<DllImport("user32.dll", SetLastError:=False, ExactSpelling:=False)>
<DllImport("user32.dll", ExactSpelling:=False, SetLastError:=False)>
```

The first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted. Refer to the individual attribute's documentation for information on default parameter values.

Attribute Targets

The *target* of an attribute is the entity to which the attribute applies. For example, an attribute may apply to a class, a particular method, or an entire assembly. By default, an attribute applies to the element that it precedes. But you can also explicitly identify, for example, whether an attribute is applied to a method, or to its parameter, or to its return value.

To explicitly identify an attribute target, use the following syntax:

```
<target : attribute-list>
```

The list of possible `target` values is shown in the following table.

TARGET VALUE	APPLIES TO
<code>assembly</code>	Entire assembly
<code>module</code>	Current assembly module (which is different from a Visual Basic Module)

The following example shows how to apply attributes to assemblies and modules. For more information, see

Common Attributes (Visual Basic).

```
Imports System.Reflection  
<Assembly: AssemblyTitleAttribute("Production assembly 4"),  
Module: CLSCompliant(True)>
```

Common Uses for Attributes

The following list includes a few of the common uses of attributes in code:

- Marking methods using the `WebMethod` attribute in Web services to indicate that the method should be callable over the SOAP protocol. For more information, see [WebMethodAttribute](#).
- Describing how to marshal method parameters when interoperating with native code. For more information, see [MarshalAsAttribute](#).
- Describing the COM properties for classes, methods, and interfaces.
- Calling unmanaged code using the `DllImportAttribute` class.
- Describing your assembly in terms of title, version, description, or trademark.
- Describing which members of a class to serialize for persistence.
- Describing how to map between class members and XML nodes for XML serialization.
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtaining information about the caller to a method.

Related Sections

For more information, see:

- [Creating Custom Attributes \(Visual Basic\)](#)
- [Accessing Attributes by Using Reflection \(Visual Basic\)](#)
- [How to: Create a C/C++ Union by Using Attributes \(Visual Basic\)](#)
- [Common Attributes \(Visual Basic\)](#)
- [Caller Information \(Visual Basic\)](#)

See also

- [Visual Basic Programming Guide](#)
- [Reflection \(Visual Basic\)](#)
- [Attributes](#)

Caller Information (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

By using Caller Info attributes, you can obtain information about the caller to a method. You can obtain file path of the source code, the line number in the source code, and the member name of the caller. This information is helpful for tracing, debugging, and creating diagnostic tools.

To obtain this information, you use attributes that are applied to optional parameters, each of which has a default value. The following table lists the Caller Info attributes that are defined in the [System.Runtime.CompilerServices](#) namespace:

ATTRIBUTE	DESCRIPTION	TYPE
CallerFilePathAttribute	Full path of the source file that contains the caller. This is the file path at compile time.	<code>String</code>
CallerLineNumberAttribute	Line number in the source file at which the method is called.	<code>Integer</code>
CallerMemberNameAttribute	Method or property name of the caller. See Member Names later in this topic.	<code>String</code>

Example

The following example shows how to use Caller Info attributes. On each call to the `TraceMessage` method, the caller information is substituted as arguments to the optional parameters.

```
Private Sub DoProcessing()
    TraceMessage("Something happened.")
End Sub

Public Sub TraceMessage(message As String,
    <System.Runtime.CompilerServices.CallerMemberName> Optional memberName As String = Nothing,
    <System.Runtime.CompilerServices.CallerFilePath> Optional sourcefilePath As String = Nothing,
    <System.Runtime.CompilerServices.CallerLineNumber()> Optional sourceLineNumber As Integer = 0)

    System.Diagnostics.Trace.WriteLine("message: " & message)
    System.Diagnostics.Trace.WriteLine("member name: " & memberName)
    System.Diagnostics.Trace.WriteLine("source file path: " & sourcefilePath)
    System.Diagnostics.Trace.WriteLine("source line number: " & sourceLineNumber)
End Sub

' Sample output:
'   message: Something happened.
'   member name: DoProcessing
'   source file path: C:\Users\username\Documents\Visual Studio
'   source line number: 15
```

Remarks

You must specify an explicit default value for each optional parameter. You can't apply Caller Info attributes to parameters that aren't specified as optional.

The Caller Info attributes don't make a parameter optional. Instead, they affect the default value that's passed in when the argument is omitted.

Caller Info values are emitted as literals into the Intermediate Language (IL) at compile time. Unlike the results of the [StackTrace](#) property for exceptions, the results aren't affected by obfuscation.

You can explicitly supply the optional arguments to control the caller information or to hide caller information.

Member Names

You can use the `CallerMemberName` attribute to avoid specifying the member name as a `String` argument to the called method. By using this technique, you avoid the problem that **Rename Refactoring** doesn't change the `String` values. This benefit is especially useful for the following tasks:

- Using tracing and diagnostic routines.
- Implementing the [INotifyPropertyChanged](#) interface when binding data. This interface allows the property of an object to notify a bound control that the property has changed, so that the control can display the updated information. Without the `CallerMemberName` attribute, you must specify the property name as a literal.

The following chart shows the member names that are returned when you use the `CallerMemberName` attribute.

CALLS OCCURS WITHIN	MEMBER NAME RESULT
Method, property, or event	The name of the method, property, or event from which the call originated.
Constructor	The string ".ctor"
Static constructor	The string ".cctor"
Destructor	The string "Finalize"
User-defined operators or conversions	The generated name for the member, for example, "op_Addition".
Attribute constructor	The name of the member to which the attribute is applied. If the attribute is any element within a member (such as a parameter, a return value, or a generic type parameter), this result is the name of the member that's associated with that element.
No containing member (for example, assembly-level or attributes that are applied to types)	The default value of the optional parameter.

See also

- [Attributes \(Visual Basic\)](#)
- [Common Attributes \(Visual Basic\)](#)
- [Optional Parameters](#)
- [Programming Concepts \(Visual Basic\)](#)

Collections (Visual Basic)

4/26/2019 • 14 minutes to read • [Edit Online](#)

For many applications, you want to create and manage groups of related objects. There are two ways to group objects: by creating arrays of objects, and by creating collections of objects.

Arrays are most useful for creating and working with a fixed number of strongly-typed objects. For information about arrays, see [Arrays](#).

Collections provide a more flexible way to work with groups of objects. Unlike arrays, the group of objects you work with can grow and shrink dynamically as the needs of the application change. For some collections, you can assign a key to any object that you put into the collection so that you can quickly retrieve the object by using the key.

A collection is a class, so you must declare an instance of the class before you can add elements to that collection.

If your collection contains elements of only one data type, you can use one of the classes in the [System.Collections.Generic](#) namespace. A generic collection enforces type safety so that no other data type can be added to it. When you retrieve an element from a generic collection, you do not have to determine its data type or convert it.

NOTE

For the examples in this topic, include `Imports` statements for the `System.Collections.Generic` and `System.Linq` namespaces.

Using a Simple Collection

The examples in this section use the generic `List<T>` class, which enables you to work with a strongly typed list of objects.

The following example creates a list of strings and then iterates through the strings by using a [For Each...Next](#) statement.

```
' Create a list of strings.  
Dim salmons As New List(Of String)  
salmons.Add("chinook")  
salmons.Add("coho")  
salmons.Add("pink")  
salmons.Add("sockeye")  
  
' Iterate through the list.  
For Each salmon As String In salmons  
    Console.WriteLine(salmon & " ")  
Next  
'Output: chinook coho pink sockeye
```

If the contents of a collection are known in advance, you can use a *collection initializer* to initialize the collection. For more information, see [Collection Initializers](#).

The following example is the same as the previous example, except a collection initializer is used to add elements to the collection.

```

' Create a list of strings by using a
' collection initializer.
Dim salmons As New List(Of String) From
 {"chinook", "coho", "pink", "sockeye"}

For Each salmon As String In salmons
 Console.WriteLine(salmon & " ")
Next
'Output: chinook coho pink sockeye

```

You can use a [For...Next](#) statement instead of a [For Each](#) statement to iterate through a collection. You accomplish this by accessing the collection elements by the index position. The index of the elements starts at 0 and ends at the element count minus 1.

The following example iterates through the elements of a collection by using [For...Next](#) instead of [For Each](#).

```

Dim salmons As New List(Of String) From
 {"chinook", "coho", "pink", "sockeye"}

For index = 0 To salmons.Count - 1
 Console.WriteLine(salmons(index) & " ")
Next
'Output: chinook coho pink sockeye

```

The following example removes an element from the collection by specifying the object to remove.

```

' Create a list of strings by using a
' collection initializer.
Dim salmons As New List(Of String) From
 {"chinook", "coho", "pink", "sockeye"}

' Remove an element in the list by specifying
' the object.
salmons.Remove("coho")

For Each salmon As String In salmons
 Console.WriteLine(salmon & " ")
Next
'Output: chinook pink sockeye

```

The following example removes elements from a generic list. Instead of a [For Each](#) statement, a [For...Next](#) statement that iterates in descending order is used. This is because the [RemoveAt](#) method causes elements after a removed element to have a lower index value.

```

Dim numbers As New List(Of Integer) From
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

' Remove odd numbers.
For index As Integer = numbers.Count - 1 To 0 Step -1
    If numbers(index) Mod 2 = 1 Then
        ' Remove the element by specifying
        ' the zero-based index in the list.
        numbers.RemoveAt(index)
    End If
Next

' Iterate through the list.
' A lambda expression is placed in the ForEach method
' of the List(T) object.
numbers.ForEach(
    Sub(number) Console.WriteLine(number & " "))
' Output: 0 2 4 6 8

```

For the type of elements in the `List<T>`, you can also define your own class. In the following example, the `Galaxy` class that is used by the `List<T>` is defined in the code.

```

Private Sub IterateThroughList()
    Dim theGalaxies As New List(Of Galaxy) From
    {
        New Galaxy With {.Name = "Tadpole", .MegaLightYears = 400},
        New Galaxy With {.Name = "Pinwheel", .MegaLightYears = 25},
        New Galaxy With {.Name = "Milky Way", .MegaLightYears = 0},
        New Galaxy With {.Name = "Andromeda", .MegaLightYears = 3}
    }

    For Each theGalaxy In theGalaxies
        With theGalaxy
            Console.WriteLine(.Name & " " & .MegaLightYears)
        End With
    Next

    ' Output:
    ' Tadpole 400
    ' Pinwheel 25
    ' Milky Way 0
    ' Andromeda 3
End Sub

Public Class Galaxy
    Public Property Name As String
    Public Property MegaLightYears As Integer
End Class

```

Kinds of Collections

Many common collections are provided by the .NET Framework. Each type of collection is designed for a specific purpose.

Some of the common collection classes are described in this section:

- [System.Collections.Generic](#) classes
- [System.Collections.Concurrent](#) classes
- [System.Collections](#) classes

- Visual Basic `Collection` class

System.Collections.Generic Classes

You can create a generic collection by using one of the classes in the [System.Collections.Generic](#) namespace. A generic collection is useful when every item in the collection has the same data type. A generic collection enforces strong typing by allowing only the desired data type to be added.

The following table lists some of the frequently used classes of the [System.Collections.Generic](#) namespace:

CLASS	DESCRIPTION
Dictionary<TKey,TValue>	Represents a collection of key/value pairs that are organized based on the key.
List<T>	Represents a list of objects that can be accessed by index. Provides methods to search, sort, and modify lists.
Queue<T>	Represents a first in, first out (FIFO) collection of objects.
SortedList<TKey,TValue>	Represents a collection of key/value pairs that are sorted by key based on the associated IComparer<T> implementation.
Stack<T>	Represents a last in, first out (LIFO) collection of objects.

For additional information, see [Commonly Used Collection Types](#), [Selecting a Collection Class](#), and [System.Collections.Generic](#).

System.Collections.Concurrent Classes

In the .NET Framework 4 or newer, the collections in the [System.Collections.Concurrent](#) namespace provide efficient thread-safe operations for accessing collection items from multiple threads.

The classes in the [System.Collections.Concurrent](#) namespace should be used instead of the corresponding types in the [System.Collections.Generic](#) and [System.Collections](#) namespaces whenever multiple threads are accessing the collection concurrently. For more information, see [Thread-Safe Collections](#) and [System.Collections.Concurrent](#).

Some classes included in the [System.Collections.Concurrent](#) namespace are [BlockingCollection<T>](#), [ConcurrentDictionary<TKey,TValue>](#), [ConcurrentQueue<T>](#), and [ConcurrentStack<T>](#).

System.Collections Classes

The classes in the [System.Collections](#) namespace do not store elements as specifically typed objects, but as objects of type `Object`.

Whenever possible, you should use the generic collections in the [System.Collections.Generic](#) namespace or the [System.Collections.Concurrent](#) namespace instead of the legacy types in the `System.Collections` namespace.

The following table lists some of the frequently used classes in the `System.Collections` namespace:

CLASS	DESCRIPTION
ArrayList	Represents an array of objects whose size is dynamically increased as required.
Hashtable	Represents a collection of key/value pairs that are organized based on the hash code of the key.
Queue	Represents a first in, first out (FIFO) collection of objects.

CLASS	DESCRIPTION
Stack	Represents a last in, first out (LIFO) collection of objects.

The [System.Collections.Specialized](#) namespace provides specialized and strongly typed collection classes, such as string-only collections and linked-list and hybrid dictionaries.

Visual Basic Collection Class

You can use the Visual Basic [Collection](#) class to access a collection item by using either a numeric index or a [String](#) key. You can add items to a collection object either with or without specifying a key. If you add an item without a key, you must use its numeric index to access it.

The Visual Basic [Collection](#) class stores all its elements as type [Object](#), so you can add an item of any data type. There is no safeguard against inappropriate data types being added.

When you use the Visual Basic [Collection](#) class, the first item in a collection has an index of 1. This differs from the .NET Framework collection classes, for which the starting index is 0.

Whenever possible, you should use the generic collections in the [System.Collections.Generic](#) namespace or the [System.Collections.Concurrent](#) namespace instead of the Visual Basic [Collection](#) class.

For more information, see [Collection](#).

Implementing a Collection of Key/Value Pairs

The [Dictionary<TKey,TValue>](#) generic collection enables you to access to elements in a collection by using the key of each element. Each addition to the dictionary consists of a value and its associated key. Retrieving a value by using its key is fast because the [Dictionary](#) class is implemented as a hash table.

The following example creates a [Dictionary](#) collection and iterates through the dictionary by using a [For Each](#) statement.

```

Private Sub IterateThroughDictionary()
    Dim elements As Dictionary(Of String, Element) = BuildDictionary()

    For Each kvp As KeyValuePair(Of String, Element) In elements
        Dim theElement As Element = kvp.Value

        Console.WriteLine("key: " & kvp.Key)
        With theElement
            Console.WriteLine("values: " & .Symbol & " " &
                .Name & " " & .AtomicNumber)
        End With
    Next
End Sub

Private Function BuildDictionary() As Dictionary(Of String, Element)
    Dim elements As New Dictionary(Of String, Element)

    AddToDictionary(elements, "K", "Potassium", 19)
    AddToDictionary(elements, "Ca", "Calcium", 20)
    AddToDictionary(elements, "Sc", "Scandium", 21)
    AddToDictionary(elements, "Ti", "Titanium", 22)

    Return elements
End Function

Private Sub AddToDictionary(ByVal elements As Dictionary(Of String, Element),
    ByVal symbol As String, ByVal name As String, ByVal atomicNumber As Integer)
    Dim theElement As New Element

    theElement.Symbol = symbol
    theElement.Name = name
    theElement.AtomicNumber = atomicNumber

    elements.Add(Key:=theElement.Symbol, value:=theElement)
End Sub

Public Class Element
    Public Property Symbol As String
    Public Property Name As String
    Public Property AtomicNumber As Integer
End Class

```

To instead use a collection initializer to build the `Dictionary` collection, you can replace the `BuildDictionary` and `AddToDictionary` methods with the following method.

```

Private Function BuildDictionary2() As Dictionary(Of String, Element)
    Return New Dictionary(Of String, Element) From
    {
        {"K", New Element With
            {.Symbol = "K", .Name = "Potassium", .AtomicNumber = 19}},
        {"Ca", New Element With
            {.Symbol = "Ca", .Name = "Calcium", .AtomicNumber = 20}},
        {"Sc", New Element With
            {.Symbol = "Sc", .Name = "Scandium", .AtomicNumber = 21}},
        {"Ti", New Element With
            {.Symbol = "Ti", .Name = "Titanium", .AtomicNumber = 22}}
    }
End Function

```

The following example uses the `ContainsKey` method and the `Item[TKey]` property of `Dictionary` to quickly find an item by key. The `Item` property enables you to access an item in the `elements` collection by using the `elements(symbol)` code in Visual Basic.

```
Private Sub FindInDictionary(ByVal symbol As String)
    Dim elements As Dictionary(Of String, Element) = BuildDictionary()

    If elements.ContainsKey(symbol) = False Then
        Console.WriteLine(symbol & " not found")
    Else
        Dim theElement = elements(symbol)
        Console.WriteLine("found: " & theElement.Name)
    End If
End Sub
```

The following example instead uses the [TryGetValue](#) method quickly find an item by key.

```
Private Sub FindInDictionary2(ByVal symbol As String)
    Dim elements As Dictionary(Of String, Element) = BuildDictionary()

    Dim theElement As Element = Nothing
    If elements.TryGetValue(symbol, theElement) = False Then
        Console.WriteLine(symbol & " not found")
    Else
        Console.WriteLine("found: " & theElement.Name)
    End If
End Sub
```

Using LINQ to Access a Collection

LINQ (Language-Integrated Query) can be used to access collections. LINQ queries provide filtering, ordering, and grouping capabilities. For more information, see [Getting Started with LINQ in Visual Basic](#).

The following example runs a LINQ query against a generic `List`. The LINQ query returns a different collection that contains the results.

```

Private Sub ShowLINQ()
    Dim elements As List(Of Element) = BuildList()

    ' LINQ Query.
    Dim subset = From theElement In elements
                 Where theElement.AtomicNumber < 22
                 Order By theElement.Name

    For Each theElement In subset
        Console.WriteLine(theElement.Name & " " & theElement.AtomicNumber)
    Next

    ' Output:
    ' Calcium 20
    ' Potassium 19
    ' Scandium 21
End Sub

Private Function BuildList() As List(Of Element)
    Return New List(Of Element) From
    {
        {New Element With
            {.Symbol = "K", .Name = "Potassium", .AtomicNumber = 19}},
        {New Element With
            {.Symbol = "Ca", .Name = "Calcium", .AtomicNumber = 20}},
        {New Element With
            {.Symbol = "Sc", .Name = "Scandium", .AtomicNumber = 21}},
        {New Element With
            {.Symbol = "Ti", .Name = "Titanium", .AtomicNumber = 22}}
    }
End Function

Public Class Element
    Public Property Symbol As String
    Public Property Name As String
    Public Property AtomicNumber As Integer
End Class

```

Sorting a Collection

The following example illustrates a procedure for sorting a collection. The example sorts instances of the `Car` class that are stored in a `List<T>`. The `Car` class implements the `IComparable<T>` interface, which requires that the `CompareTo` method be implemented.

Each call to the `CompareTo` method makes a single comparison that is used for sorting. User-written code in the `CompareTo` method returns a value for each comparison of the current object with another object. The value returned is less than zero if the current object is less than the other object, greater than zero if the current object is greater than the other object, and zero if they are equal. This enables you to define in code the criteria for greater than, less than, and equal.

In the `ListCars` method, the `cars.Sort()` statement sorts the list. This call to the `Sort` method of the `List<T>` causes the `CompareTo` method to be called automatically for the `Car` objects in the `List`.

```

Public Sub ListCars()

    ' Create some new cars.
    Dim cars As New List(Of Car) From
    {
        New Car With {.Name = "car1", .Color = "blue", .Speed = 20},
        New Car With {.Name = "car2", .Color = "red", .Speed = 50},
        New Car With {.Name = "car3", .Color = "green", .Speed = 10},
        New Car With {.Name = "car4", .Color = "blue", .Speed = 50},
        New Car With {.Name = "car5", .Color = "black", .Speed = 30}
    }

```

```

        New Car With {.Name = "car5", .Color = "blue", .Speed = 30},
        New Car With {.Name = "car6", .Color = "red", .Speed = 60},
        New Car With {.Name = "car7", .Color = "green", .Speed = 50}
    }

    ' Sort the cars by color alphabetically, and then by speed
    ' in descending order.
    cars.Sort()

    ' View all of the cars.
    For Each thisCar As Car In cars
        Console.WriteLine(thisCar.Color.PadRight(5) & " ")
        Console.WriteLine(thisCar.Speed.ToString & " ")
        Console.WriteLine(thisCar.Name)
        Console.WriteLine()
    Next

    ' Output:
    ' blue 50 car4
    ' blue 30 car5
    ' blue 20 car1
    ' green 50 car7
    ' green 10 car3
    ' red 60 car6
    ' red 50 car2
End Sub

Public Class Car
    Implements IComparable(Of Car)

    Public Property Name As String
    Public Property Speed As Integer
    Public Property Color As String

    Public Function CompareTo(ByVal other As Car) As Integer _
        Implements System.IComparable(Of Car).CompareTo
        ' A call to this method makes a single comparison that is
        ' used for sorting.

        ' Determine the relative order of the objects being compared.
        ' Sort by color alphabetically, and then by speed in
        ' descending order.

        ' Compare the colors.
        Dim compare As Integer
        compare = String.Compare(Me.Color, other.Color, True)

        ' If the colors are the same, compare the speeds.
        If compare = 0 Then
            compare = Me.Speed.CompareTo(other.Speed)

            ' Use descending order for speed.
            compare = -compare
        End If

        Return compare
    End Function
End Class

```

Defining a Custom Collection

You can define a collection by implementing the [IEnumerable<T>](#) or [IEnumerable](#) interface. For additional information, see [Enumerating a Collection](#).

Although you can define a custom collection, it is usually better to instead use the collections that are included in the .NET Framework, which are described in [Kinds of Collections](#) earlier in this topic.

The following example defines a custom collection class named `AllColors`. This class implements the `IEnumerable` interface, which requires that the `GetEnumerator` method be implemented.

The `GetEnumerator` method returns an instance of the `ColorEnumerator` class. `ColorEnumerator` implements the `IEnumerator` interface, which requires that the `Current` property, `MoveNext` method, and `Reset` method be implemented.

```

Public Sub ListColors()
    Dim colors As New AllColors()

    For Each theColor As Color In colors
        Console.WriteLine(theColor.Name & " ")
    Next
    Console.WriteLine()
    ' Output: red blue green
End Sub

' Collection class.
Public Class AllColors
    Implements System.Collections.IEnumerable

    Private _colors() As Color =
    {
        New Color With {.Name = "red"},
        New Color With {.Name = "blue"},
        New Color With {.Name = "green"}
    }

    Public Function GetEnumerator() As System.Collections.IEnumerator _
        Implements System.Collections.IEnumerable.GetEnumerator

        Return New ColorEnumerator(_colors)

        ' Instead of creating a custom enumerator, you could
        ' use the GetEnumerator of the array.
        'Return _colors.GetEnumerator
    End Function

    ' Custom enumerator.
    Private Class ColorEnumerator
        Implements System.Collections.IEnumerator

        Private _colors() As Color
        Private _position As Integer = -1

        Public Sub New(ByVal colors() As Color)
            _colors = colors
        End Sub

        Public ReadOnly Property Current() As Object _
            Implements System.Collections.IEnumerator.Current
            Get
                Return _colors(_position)
            End Get
        End Property

        Public Function MoveNext() As Boolean _
            Implements System.Collections.IEnumerator.MoveNext
            _position += 1
            Return (_position < _colors.Length)
        End Function

        Public Sub Reset() Implements System.Collections.IEnumerator.Reset
            _position = -1
        End Sub
    End Class
End Class

' Element class.
Public Class Color
    Public Property Name As String
End Class

```

Iterators

An *iterator* is used to perform a custom iteration over a collection. An iterator can be a method or a `get` accessor. An iterator uses a `Yield` statement to return each element of the collection one at a time.

You call an iterator by using a `For Each...Next` statement. Each iteration of the `For Each` loop calls the iterator. When a `Yield` statement is reached in the iterator, an expression is returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator is called.

For more information, see [Iterators \(Visual Basic\)](#).

The following example uses an iterator method. The iterator method has a `Yield` statement that is inside a `For...Next` loop. In the `ListEvenNumbers` method, each iteration of the `For Each` statement body creates a call to the iterator method, which proceeds to the next `Yield` statement.

```
Public Sub ListEvenNumbers()
    For Each number As Integer In EvenSequence(5, 18)
        Console.WriteLine(number & " ")
    Next
    Console.WriteLine()
    ' Output: 6 8 10 12 14 16 18
End Sub

Private Iterator Function EvenSequence(
    ByVal firstNumber As Integer, ByVal lastNumber As Integer) _
    As IEnumerable(Of Integer)

    ' Yield even numbers in the range.
    For number = firstNumber To lastNumber
        If number Mod 2 = 0 Then
            Yield number
        End If
    Next
End Function
```

See also

- [Collection Initializers](#)
- [Programming Concepts \(Visual Basic\)](#)
- [Option Strict Statement](#)
- [LINQ to Objects \(Visual Basic\)](#)
- [Parallel LINQ \(PLINQ\)](#)
- [Collections and Data Structures](#)
- [Selecting a Collection Class](#)
- [Comparisons and Sorts Within Collections](#)
- [When to Use Generic Collections](#)

Covariance and Contravariance (Visual Basic)

10/17/2019 • 3 minutes to read • [Edit Online](#)

In Visual Basic, covariance and contravariance enable implicit reference conversion for array types, delegate types, and generic type arguments. Covariance preserves assignment compatibility and contravariance reverses it.

The following code demonstrates the difference between assignment compatibility, covariance, and contravariance.

```
' Assignment compatibility.  
Dim str As String = "test"  
' An object of a more derived type is assigned to an object of a less derived type.  
Dim obj As Object = str  
  
' Covariance.  
Dim strings As IEnumerable(Of String) = New List(Of String)()  
' An object that is instantiated with a more derived type argument  
' is assigned to an object instantiated with a less derived type argument.  
' Assignment compatibility is preserved.  
Dim objects As IEnumerable(Of Object) = strings  
  
' Contravariance.  
' Assume that there is the following method in the class:  
' Shared Sub SetObject(ByVal o As Object)  
' End Sub  
Dim actObject As Action(Of Object) = AddressOf SetObject  
  
' An object that is instantiated with a less derived type argument  
' is assigned to an object instantiated with a more derived type argument.  
' Assignment compatibility is reversed.  
Dim actString As Action(Of String) = actObject
```

Covariance for arrays enables implicit conversion of an array of a more derived type to an array of a less derived type. But this operation is not type safe, as shown in the following code example.

```
Dim array() As Object = New String(10) {}  
' The following statement produces a run-time exception.  
' array(0) = 10
```

Covariance and contravariance support for method groups allows for matching method signatures with delegate types. This enables you to assign to delegates not only methods that have matching signatures, but also methods that return more derived types (covariance) or that accept parameters that have less derived types (contravariance) than that specified by the delegate type. For more information, see [Variance in Delegates \(Visual Basic\)](#) and [Using Variance in Delegates \(Visual Basic\)](#).

The following code example shows covariance and contravariance support for method groups.

```

Shared Function GetObject() As Object
    Return Nothing
End Function

Shared Sub SetObject(ByVal obj As Object)
End Sub

Shared Function GetString() As String
    Return ""
End Function

Shared Sub SetString(ByVal str As String)

End Sub

Shared Sub Test()
    ' Covariance. A delegate specifies a return type as object,
    ' but you can assign a method that returns a string.
    Dim del As Func(Of Object) = AddressOf GetString

    ' Contravariance. A delegate specifies a parameter type as string,
    ' but you can assign a method that takes an object.
    Dim del2 As Action(Of String) = AddressOf SetObject
End Sub

```

In .NET Framework 4 or later, Visual Basic supports covariance and contravariance in generic interfaces and delegates and allows for implicit conversion of generic type parameters. For more information, see [Variance in Generic Interfaces \(Visual Basic\)](#) and [Variance in Delegates \(Visual Basic\)](#).

The following code example shows implicit reference conversion for generic interfaces.

```

Dim strings As IEnumerable(Of String) = New List(Of String)
Dim objects As IEnumerable(Of Object) = strings

```

A generic interface or delegate is called *variant* if its generic parameters are declared covariant or contravariant. Visual Basic enables you to create your own variant interfaces and delegates. For more information, see [Creating Variant Generic Interfaces \(Visual Basic\)](#) and [Variance in Delegates \(Visual Basic\)](#).

Related Topics

TITLE	DESCRIPTION
Variance in Generic Interfaces (Visual Basic)	Discusses covariance and contravariance in generic interfaces and provides a list of variant generic interfaces in the .NET Framework.
Creating Variant Generic Interfaces (Visual Basic)	Shows how to create custom variant interfaces.
Using Variance in Interfaces for Generic Collections (Visual Basic)	Shows how covariance and contravariance support in the <code>IEnumerable<T></code> and <code>IComparable<T></code> interfaces can help you reuse code.
Variance in Delegates (Visual Basic)	Discusses covariance and contravariance in generic and non-generic delegates and provides a list of variant generic delegates in the .NET Framework.

TITLE	DESCRIPTION
Using Variance in Delegates (Visual Basic)	Shows how to use covariance and contravariance support in non-generic delegates to match method signatures with delegate types.
Using Variance for Func and Action Generic Delegates (Visual Basic)	Shows how covariance and contravariance support in the <code>Func</code> and <code>Action</code> delegates can help you reuse code.

Expression Trees (Visual Basic)

4/2/2019 • 4 minutes to read • [Edit Online](#)

Expression trees represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`.

You can compile and run code represented by expression trees. This enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. For more information about expression trees in LINQ, see [How to: Use Expression Trees to Build Dynamic Queries \(Visual Basic\)](#).

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and the .NET Framework and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (MSIL). For more information about the DLR, see [Dynamic Language Runtime Overview](#).

You can have the C# or Visual Basic compiler create an expression tree for you based on an anonymous lambda expression, or you can create expression trees manually by using the [System.Linq.Expressions](#) namespace.

Creating Expression Trees from Lambda Expressions

When a lambda expression is assigned to a variable of type [Expression<TDelegate>](#), the compiler emits code to build an expression tree that represents the lambda expression.

The Visual Basic compiler can generate expression trees only from expression lambdas (or single-line lambdas). It cannot parse statement lambdas (or multi-line lambdas). For more information about lambda expressions in Visual Basic, see [Lambda Expressions](#).

The following code examples demonstrate how to have the Visual Basic compiler create an expression tree that represents the lambda expression `Function(num) num < 5`.

```
Dim lambda As Expression(Of Func(Of Integer, Boolean)) =
    Function(num) num < 5
```

Creating Expression Trees by Using the API

To create expression trees by using the API, use the [Expression](#) class. This class contains static factory methods that create expression tree nodes of specific types, for example, [ParameterExpression](#), which represents a variable or parameter, or [MethodCallExpression](#), which represents a method call. [ParameterExpression](#), [MethodCallExpression](#), and the other expression-specific types are also defined in the [System.Linq.Expressions](#) namespace. These types derive from the abstract type [Expression](#).

The following code example demonstrates how to create an expression tree that represents the lambda expression `Function(num) num < 5` by using the API.

```

' Import the following namespace to your project: System.Linq.Expressions

' Manually build the expression tree for the lambda expression num => num < 5.
Dim numParam As ParameterExpression = Expression.Parameter(GetType(Integer), "num")
Dim five As ConstantExpression = Expression.Constant(5, GetType(Integer))
Dim numLessThanFive As BinaryExpression = Expression.LessThan(numParam, five)
Dim lambda1 As Expression(Of Func(Of Integer, Boolean)) =
    Expression.Lambda(Of Func(Of Integer, Boolean))(
        numLessThanFive,
        New ParameterExpression() {numParam})

```

In .NET Framework 4 or later, the expression trees API also supports assignments and control flow expressions such as loops, conditional blocks, and `try-catch` blocks. By using the API, you can create expression trees that are more complex than those that can be created from lambda expressions by the Visual Basic compiler. The following example demonstrates how to create an expression tree that calculates the factorial of a number.

```

' Creating a parameter expression.
Dim value As ParameterExpression =
    Expression.Parameter(GetType(Integer), "value")

' Creating an expression to hold a local variable.
Dim result As ParameterExpression =
    Expression.Parameter(GetType(Integer), "result")

' Creating a label to jump to from a loop.
Dim label As LabelTarget = Expression.Label(GetType(Integer))

' Creating a method body.
Dim block As BlockExpression = Expression.Block(
    New ParameterExpression() {result},
    Expression.Assign(result, Expression.Constant(1)),
    Expression.Loop(
        Expression.IfThenElse(
            Expression.GreaterThan(value, Expression.Constant(1)),
            Expression.MultiplyAssign(result,
                Expression.PostDecrementAssign(value)),
            Expression.Break(label, result)
        ),
        label
    )
)

' Compile an expression tree and return a delegate.
Dim factorial As Integer =
    Expression.Lambda(Of Func(Of Integer, Integer))(block, value).Compile()(5)

Console.WriteLine(factorial)
' Prints 120.

```

For more information, see [Generating Dynamic Methods with Expression Trees in Visual Studio 2010](#), which also applies to later versions of Visual Studio.

Parsing Expression Trees

The following code example demonstrates how the expression tree that represents the lambda expression `Function(num) num < 5` can be decomposed into its parts.

```

' Import the following namespace to your project: System.Linq.Expressions

' Create an expression tree.
Dim exprTree As Expression(Of Func(Of Integer, Boolean)) = Function(num) num < 5

' Decompose the expression tree.
Dim param As ParameterExpression = exprTree.Parameters(0)
Dim operation As BinaryExpression = exprTree.Body
Dim left As ParameterExpression = operation.Left
Dim right As ConstantExpression = operation.Right

Console.WriteLine(String.Format("Decomposed expression: {0} => {1} {2} {3}",
    param.Name, left.Name, operation.NodeType, right.Value))

' This code produces the following output:
'

' Decomposed expression: num => num LessThan 5

```

Immutability of Expression Trees

Expression trees should be immutable. This means that if you want to modify an expression tree, you must construct a new expression tree by copying the existing one and replacing nodes in it. You can use an expression tree visitor to traverse the existing expression tree. For more information, see [How to: Modify Expression Trees \(Visual Basic\)](#).

Compiling Expression Trees

The [Expression<TDelegate>](#) type provides the [Compile](#) method that compiles the code represented by an expression tree into an executable delegate.

The following code example demonstrates how to compile an expression tree and run the resulting code.

```

' Creating an expression tree.
Dim expr As Expression(Of Func(Of Integer, Boolean)) =
    Function(num) num < 5

' Compiling the expression tree into a delegate.
Dim result As Func(Of Integer, Boolean) = expr.Compile()

' Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4))

' Prints True.

' You can also use simplified syntax
' to compile and run an expression tree.
' The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4))

' Also prints True.

```

For more information, see [How to: Execute Expression Trees \(Visual Basic\)](#).

See also

- [System.Linq.Expressions](#)
- [How to: Execute Expression Trees \(Visual Basic\)](#)
- [How to: Modify Expression Trees \(Visual Basic\)](#)
- [Lambda Expressions](#)

- [Dynamic Language Runtime Overview](#)
- [Programming Concepts \(Visual Basic\)](#)

Iterators (Visual Basic)

10/18/2019 • 10 minutes to read • [Edit Online](#)

An *iterator* can be used to step through collections such as lists and arrays.

An iterator method or `get` accessor performs a custom iteration over a collection. An iterator method uses the `Yield` statement to return each element one at a time. When a `Yield` statement is reached, the current location in code is remembered. Execution is restarted from that location the next time the iterator function is called.

You consume an iterator from client code by using a `For Each...Next` statement, or by using a LINQ query.

In the following example, the first iteration of the `For Each` loop causes execution to proceed in the `SomeNumbers` iterator method until the first `Yield` statement is reached. This iteration returns a value of 3, and the current location in the iterator method is retained. On the next iteration of the loop, execution in the iterator method continues from where it left off, again stopping when it reaches a `Yield` statement. This iteration returns a value of 5, and the current location in the iterator method is again retained. The loop completes when the end of the iterator method is reached.

```
Sub Main()
    For Each number As Integer In SomeNumbers()
        Console.WriteLine(number & " ")
    Next
    ' Output: 3 5 8
    Console.ReadKey()
End Sub

Private Iterator Function SomeNumbers() As System.Collections.IEnumerable
    Yield 3
    Yield 5
    Yield 8
End Function
```

The return type of an iterator method or `get` accessor can be `IEnumerable`, `IEnumerable<T>`, `IEnumerator`, or `IEnumerator<T>`.

You can use an `Exit Function` or `Return` statement to end the iteration.

A Visual Basic iterator function or `get` accessor declaration includes an `Iterator` modifier.

Iterators were introduced in Visual Basic in Visual Studio 2012.

In this topic

- [Simple Iterator](#)
- [Creating a Collection Class](#)
- [Try Blocks](#)
- [Anonymous Methods](#)
- [Using Iterators with a Generic List](#)
- [Syntax Information](#)
- [Technical Implementation](#)

- Use of Iterators

NOTE

For all examples in the topic except the Simple Iterator example, include **Imports** statements for the `System.Collections` and `System.Collections.Generic` namespaces.

Simple Iterator

The following example has a single `Yield` statement that is inside a `For...Next` loop. In `Main`, each iteration of the `For Each` statement body creates a call to the iterator function, which proceeds to the next `Yield` statement.

```
Sub Main()
    For Each number As Integer In EvenSequence(5, 18)
        Console.WriteLine(number & " ")
    Next
    ' Output: 6 8 10 12 14 16 18
    Console.ReadKey()
End Sub

Private Iterator Function EvenSequence(
    ByVal firstNumber As Integer, ByVal lastNumber As Integer) _
As System.Collections.Generic.IEnumerable(Of Integer)

    ' Yield even numbers in the range.
    For number As Integer = firstNumber To lastNumber
        If number Mod 2 = 0 Then
            Yield number
        End If
    Next
End Function
```

Creating a Collection Class

In the following example, the `DaysOfTheWeek` class implements the **IEnumerable** interface, which requires a **GetEnumerator** method. The compiler implicitly calls the `GetEnumerator` method, which returns an **IEnumerator**.

The `GetEnumerator` method returns each string one at a time by using the `yield` statement, and an `Iterator` modifier is in the function declaration.

```

Sub Main()
    Dim days As New DaysOfTheWeek()
    For Each day As String In days
        Console.WriteLine(day & " ")
    Next
    ' Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey()
End Sub

Private Class DaysOfTheWeek
    Implements IEnumerable

    Public days =
        New String() {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"}

    Public Iterator Function GetEnumerator() As IEnumerator _
        Implements IEnumerable.GetEnumerator

        ' Yield each day of the week.
        For i As Integer = 0 To days.Length - 1
            Yield days(i)
        Next
    End Function
End Class

```

The following example creates a `Zoo` class that contains a collection of animals.

The `For Each` statement that refers to the class instance (`theZoo`) implicitly calls the `GetEnumerator` method. The `For Each` statements that refer to the `Birds` and `Mammals` properties use the `AnimalsForType` named iterator method.

```

Sub Main()
    Dim theZoo As New Zoo()

    theZoo.AddMammal("Whale")
    theZoo.AddMammal("Rhinoceros")
    theZoo.AddBird("Penguin")
    theZoo.AddBird("Warbler")

    For Each name As String In theZoo
        Console.WriteLine(name & " ")
    Next
    Console.WriteLine()
    ' Output: Whale Rhinoceros Penguin Warbler

    For Each name As String In theZoo.Birds
        Console.WriteLine(name & " ")
    Next
    Console.WriteLine()
    ' Output: Penguin Warbler

    For Each name As String In theZoo.Mammals
        Console.WriteLine(name & " ")
    Next
    Console.WriteLine()
    ' Output: Whale Rhinoceros

    Console.ReadKey()
End Sub

Public Class Zoo
    Implements IEnumerable

    ' Private members.
    Private animals As New List(Of Animal)

```

```

' Public methods.
Public Sub AddMammal(ByVal name As String)
    animals.Add(New Animal With {.Name = name, .Type = Animal.TypeEnum.Mammal})
End Sub

Public Sub AddBird(ByVal name As String)
    animals.Add(New Animal With {.Name = name, .Type = Animal.TypeEnum.Bird})
End Sub

Public Iterator Function GetEnumerator() As IEnumerable _
    Implements IEnumerable.GetEnumerator

    For Each theAnimal As Animal In animals
        Yield theAnimal.Name
    Next
End Function

' Public members.
Public ReadOnly Property Mammals As IEnumerable
    Get
        Return AnimalsForType(Animal.TypeEnum.Mammal)
    End Get
End Property

Public ReadOnly Property Birds As IEnumerable
    Get
        Return AnimalsForType(Animal.TypeEnum.Bird)
    End Get
End Property

' Private methods.
Private Iterator Function AnimalsForType( _
    ByVal type As Animal.TypeEnum) As IEnumerable
    For Each theAnimal As Animal In animals
        If (theAnimal.Type = type) Then
            Yield theAnimal.Name
        End If
    Next
End Function

' Private class.
Private Class Animal
    Public Enum TypeEnum
        Bird
        Mammal
    End Enum

    Public Property Name As String
    Public Property Type As TypeEnum
    End Class
End Class

```

Try Blocks

Visual Basic allows a `Yield` statement in the `Try` block of a [Try...Catch...Finally Statement](#). A `Try` block that has a `Yield` statement can have `Catch` blocks, and can have a `Finally` block.

The following example includes `Try`, `Catch`, and `Finally` blocks in an iterator function. The `Finally` block in the iterator function executes before the `For Each` iteration finishes.

```

Sub Main()
    For Each number As Integer In Test()
        Console.WriteLine(number)
    Next
    Console.WriteLine("For Each is done.")

    ' Output:
    ' 3
    ' 4
    ' Something happened. Yields are done.
    ' Finally is called.
    ' For Each is done.
    Console.ReadKey()
End Sub

Private Iterator Function Test() As IEnumerable(Of Integer)
    Try
        Yield 3
        Yield 4
        Throw New Exception("Something happened. Yields are done.")
        Yield 5
        Yield 6
    Catch ex As Exception
        Console.WriteLine(ex.Message)
    Finally
        Console.WriteLine("Finally is called.")
    End Try
End Function

```

A `Yield` statement cannot be inside a `Catch` block or a `Finally` block.

If the `For Each` body (instead of the iterator method) throws an exception, a `Catch` block in the iterator function is not executed, but a `Finally` block in the iterator function is executed. A `catch` block inside an iterator function catches only exceptions that occur inside the iterator function.

Anonymous Methods

In Visual Basic, an anonymous function can be an iterator function. The following example illustrates this.

```

Dim iterateSequence = Iterator Function() _
    As IEnumerable(Of Integer)
    Yield 1
    Yield 2
End Function

For Each number As Integer In iterateSequence()
    Console.Write(number & " ")
Next
' Output: 1 2
Console.ReadKey()

```

The following example has a non-iterator method that validates the arguments. The method returns the result of an anonymous iterator that describes the collection elements.

```

Sub Main()
    For Each number As Integer In GetSequence(5, 10)
        Console.WriteLine(number & " ")
    Next
    ' Output: 5 6 7 8 9 10
    Console.ReadKey()
End Sub

Public Function GetSequence(ByVal low As Integer, ByVal high As Integer) _
As IEnumerable
    ' Validate the arguments.
    If low < 1 Then
        Throw New ArgumentException("low is too low")
    End If
    If high > 140 Then
        Throw New ArgumentException("high is too high")
    End If

    ' Return an anonymous iterator function.
    Dim iterateSequence = Iterator Function() As IEnumerable
        For index = low To high
            Yield index
        Next
    End Function

    Return iterateSequence()
End Function

```

If validation is instead inside the iterator function, the validation cannot be performed until the start of the first iteration of the `For Each` body.

Using Iterators with a Generic List

In the following example, the `Stack(of T)` generic class implements the `IEnumerable<T>` generic interface. The `Push` method assigns values to an array of type `T`. The `GetEnumerator` method returns the array values by using the `Yield` statement.

In addition to the generic `GetEnumerator` method, the non-generic `GetEnumerator` method must also be implemented. This is because `IEnumerable<T>` inherits from `IEnumerable`. The non-generic implementation defers to the generic implementation.

The example uses named iterators to support various ways of iterating through the same collection of data. These named iterators are the `TopToBottom` and `BottomToTop` properties, and the `TopN` method.

The `BottomToTop` property declaration includes the `Iterator` keyword.

```

Sub Main()
    Dim theStack As New Stack(Of Integer)

    ' Add items to the stack.
    For number As Integer = 0 To 9
        theStack.Push(number)
    Next

    ' Retrieve items from the stack.
    ' For Each is allowed because theStack implements
    ' IEnumerable(Of Integer).
    For Each number As Integer In theStack
        Console.WriteLine("{0} ", number)
    Next
    Console.WriteLine()
    ' Output: 9 8 7 6 5 4 3 2 1 0

    ' For Each is allowed, because theStack.TopToBottom

```

```

    ' Each is allowed, because the stack implements IEnumerable(Of Integer).
For Each number As Integer In theStack.TopToBottom
    Console.WriteLine("{0} ", number)
Next
Console.WriteLine()
' Output: 9 8 7 6 5 4 3 2 1 0

For Each number As Integer In theStack.BottomToTop
    Console.WriteLine("{0} ", number)
Next
Console.WriteLine()
' Output: 0 1 2 3 4 5 6 7 8 9

For Each number As Integer In theStack.TopN(7)
    Console.WriteLine("{0} ", number)
Next
Console.WriteLine()
' Output: 9 8 7 6 5 4 3

Console.ReadKey()
End Sub

Public Class Stack(Of T)
    Implements IEnumerable(Of T)

    Private values As T() = New T(99) {}
    Private top As Integer = 0

    Public Sub Push(ByVal t As T)
        values(top) = t
        top = top + 1
    End Sub

    Public Function Pop() As T
        top = top - 1
        Return values(top)
    End Function

    ' This function implements the GetEnumerator method. It allows
    ' an instance of the class to be used in a For Each statement.
    Public Iterator Function GetEnumerator() As IEnumerator(Of T) _
        Implements IEnumerable(Of T).GetEnumerator

        For index As Integer = top - 1 To 0 Step -1
            Yield values(index)
        Next
    End Function

    Public Iterator Function GetEnumerator1() As IEnumerator _
        Implements IEnumerable.GetEnumerator

        Yield GetEnumerator()
    End Function

    Public ReadOnly Property TopToBottom() As IEnumerable(Of T)
        Get
            Return Me
        End Get
    End Property

    Public ReadOnly Iterator Property BottomToTop As IEnumerable(Of T)
        Get
            For index As Integer = 0 To top - 1
                Yield values(index)
            Next
        End Get
    End Property

    Public Iterator Function TopN(ByVal n As Integer) As IEnumerable

```

```

    Public Iterator Function TopN(ByVal itemsFromTop As Integer) _
        As IEnumerable(Of T)

        ' Return less than itemsFromTop if necessary.
        Dim startIndex As Integer =
            If(itemsFromTop >= top, 0, top - itemsFromTop)

        For index As Integer = top - 1 To startIndex Step -1
            Yield values(index)
        Next
    End Function
End Class

```

Syntax Information

An iterator can occur as a method or `get` accessor. An iterator cannot occur in an event, instance constructor, static constructor, or static destructor.

An implicit conversion must exist from the expression type in the `Yield` statement to the return type of the iterator.

In Visual Basic, an iterator method cannot have any `ByRef` parameters.

In Visual Basic, "Yield" is not a reserved word and has special meaning only when it is used in an `Iterator` method or `get` accessor.

Technical Implementation

Although you write an iterator as a method, the compiler translates it into a nested class that is, in effect, a state machine. This class keeps track of the position of the iterator as long the `For Each...Next` loop in the client code continues.

To see what the compiler does, you can use the `Ildasm.exe` tool to view the Microsoft intermediate language code that is generated for an iterator method.

When you create an iterator for a `class` or `struct`, you do not have to implement the whole `IEnumerator` interface. When the compiler detects the iterator, it automatically generates the `Current`, `MoveNext`, and `Dispose` methods of the `IEnumerator` or `IEnumerator<T>` interface.

On each successive iteration of the `For Each...Next` loop (or the direct call to `IEnumerator.MoveNext`), the next iterator code body resumes after the previous `Yield` statement. It then continues to the next `Yield` statement until the end of the iterator body is reached, or until an `Exit Function` or `Return` statement is encountered.

Iterators do not support the `IEnumerator.Reset` method. To re-iterate from the start, you must obtain a new iterator.

For additional information, see the [Visual Basic Language Specification](#).

Use of Iterators

Iterators enable you to maintain the simplicity of a `For Each` loop when you need to use complex code to populate a list sequence. This can be useful when you want to do the following:

- Modify the list sequence after the first `For Each` loop iteration.
- Avoid fully loading a large list before the first iteration of a `For Each` loop. An example is a paged fetch to load a batch of table rows. Another example is the `EnumerateFiles` method, which implements iterators within the .NET Framework.

- Encapsulate building the list in the iterator. In the iterator method, you can build the list and then yield each result in a loop.

See also

- [System.Collections.Generic](#)
- [IEnumerable<T>](#)
- [For Each...Next Statement](#)
- [Yield Statement](#)
- [Iterator](#)

Language-Integrated Query (LINQ) (Visual Basic)

5/4/2018 • 2 minutes to read • [Edit Online](#)

LINQ is a set of features that extends powerful query capabilities to the language syntax of Visual Basic. LINQ introduces standard, easily-learned patterns for querying and updating data, and the technology can be extended to support potentially any kind of data store. The .NET Framework includes LINQ provider assemblies that enable the use of LINQ with .NET Framework collections, SQL Server databases, ADO.NET Datasets, and XML documents.

In This Section

[Introduction to LINQ \(Visual Basic\)](#)

Provides a general introduction to the kinds of applications that you can write and the kinds of problems that you can solve with LINQ queries.

[Getting Started with LINQ in Visual Basic](#)

Describes the basic facts you should know in order to understand the Visual Basic documentation and samples.

[Visual Studio IDE and Tools Support for LINQ \(Visual Basic\)](#)

Describes Visual Studio's Object Relational Designer, debugger support for queries, and other IDE features related to LINQ.

[Standard Query Operators Overview \(Visual Basic\)](#)

Provides an introduction to the standard query operators. It also provides links to topics that have more information about each type of query operation.

[LINQ to Objects \(Visual Basic\)](#)

Includes links to topics that explain how to use LINQ to Objects to access in-memory data structures,

[LINQ to XML \(Visual Basic\)](#)

Includes links to topics that explain how to use LINQ to XML, which provides the in-memory document modification capabilities of the Document Object Model (DOM), and supports LINQ query expressions.

[LINQ to ADO.NET \(Portal Page\)](#)

Provides an entry point for documentation about LINQ to DataSet, LINQ to SQL, and LINQ to Entities. LINQ to DataSet enables you to build richer query capabilities into [DataSet](#) by using the same query functionality that is available for other data sources. LINQ to SQL provides a run-time infrastructure for managing relational data as objects. LINQ to Entities enables developers to write queries against the Entity Framework conceptual model by using C#.

[Enabling a Data Source for LINQ Querying](#)

Provides an introduction to custom LINQ providers, LINQ expression trees, and other ways to extend LINQ.

Object-oriented programming (Visual Basic)

11/16/2018 • 10 minutes to read • [Edit Online](#)

Visual Basic provides full support for object-oriented programming including encapsulation, inheritance, and polymorphism.

Encapsulation means that a group of related properties, methods, and other members are treated as a single unit or object.

Inheritance describes the ability to create new classes based on an existing class.

Polymorphism means that you can have multiple classes that can be used interchangeably, even though each class implements the same properties or methods in different ways.

This section describes the following concepts:

- [Classes and objects](#)
 - [Class members](#)
 - [Properties and fields](#)
 - [Methods](#)
 - [Constructors](#)
 - [Destructors](#)
 - [Events](#)
 - [Nested classes](#)
 - [Access modifiers and access levels](#)
 - [Instantiating classes](#)
 - [Shared classes and members](#)
 - [Anonymous types](#)
- [Inheritance](#)
 - [Overriding members](#)
- [Interfaces](#)
- [Generics](#)
- [Delegates](#)

Classes and objects

The terms *class* and *object* are sometimes used interchangeably, but in fact, classes describe the *type* of objects, while objects are usable *instances* of classes. So, the act of creating an object is called *instantiation*. Using the blueprint analogy, a class is a blueprint, and an object is a building made from that blueprint.

To define a class:

```
Class SampleClass  
End Class
```

Visual Basic also provides a light version of classes called *structures* that are useful when you need to create large array of objects and do not want to consume too much memory for that.

To define a structure:

```
Structure SampleStructure  
End Structure
```

For more information, see:

- [Class Statement](#)
- [Structure Statement](#)

Class members

Each class can have different *class members* that include properties that describe class data, methods that define class behavior, and events that provide communication between different classes and objects.

Properties and fields

Fields and properties represent information that an object contains. Fields are like variables because they can be read or set directly.

To define a field:

```
Class SampleClass  
    Public SampleField As String  
End Class
```

Properties have get and set procedures, which provide more control on how values are set or returned.

Visual Basic allows you either to create a private field for storing the property value or use so-called auto-implemented properties that create this field automatically behind the scenes and provide the basic logic for the property procedures.

To define an auto-implemented property:

```
Class SampleClass  
    Public Property SampleProperty as String  
End Class
```

If you need to perform some additional operations for reading and writing the property value, define a field for storing the property value and provide the basic logic for storing and retrieving it:

```
Class SampleClass  
    Private m_Sample As String  
    Public Property Sample() As String  
        Get  
            ' Return the value stored in the field.  
            Return m_Sample  
        End Get  
        Set(ByVal Value As String)  
            ' Store the value in the field.  
            m_Sample = Value  
        End Set  
    End Property  
End Class
```

Most properties have methods or procedures to both set and get the property value. However, you can create read-only or write-only properties to restrict them from being modified or read. In Visual Basic you can use `ReadOnly` and `WriteOnly` keywords. However, auto-implemented properties cannot be read-only or write-only.

For more information, see:

- [Property Statement](#)
- [Get Statement](#)
- [Set Statement](#)
- [ReadOnly](#)
- [WriteOnly](#)

Methods

A *method* is an action that an object can perform.

NOTE

In Visual Basic, there are two ways to create a method: the `Sub` statement is used if the method does not return a value; the `Function` statement is used if a method returns a value.

To define a method of a class:

```
Class SampleClass
    Public Function SampleFunc(ByVal SampleParam As String)
        ' Add code here
    End Function
End Class
```

A class can have several implementations, or *overloads*, of the same method that differ in the number of parameters or parameter types.

To overload a method:

```
Overloads Sub Display(ByVal theChar As Char)
    ' Add code that displays Char data.
End Sub
Overloads Sub Display(ByVal theInteger As Integer)
    ' Add code that displays Integer data.
End Sub
```

In most cases you declare a method within a class definition. However, Visual Basic also supports *extension methods* that allow you to add methods to an existing class outside the actual definition of the class.

For more information, see:

- [Function Statement](#)
- [Sub Statement](#)
- [Overloads](#)
- [Extension Methods](#)

Constructors

Constructors are class methods that are executed automatically when an object of a given type is created.

Constructors usually initialize the data members of the new object. A constructor can run only once when a class is created. Furthermore, the code in the constructor always runs before any other code in a class. However, you can create multiple constructor overloads in the same way as for any other method.

To define a constructor for a class:

```
Class SampleClass
    Sub New(ByVal s As String)
        // Add code here.
    End Sub
End Class
```

For more information, see: [Object Lifetime: How Objects Are Created and Destroyed](#).

Destructors

Destructors are used to destruct instances of classes. In the .NET Framework, the garbage collector automatically manages the allocation and release of memory for the managed objects in your application. However, you may still need destructors to clean up any unmanaged resources that your application creates. There can be only one destructor for a class.

For more information about destructors and garbage collection in the .NET Framework, see [Garbage Collection](#).

Events

Events enable a class or object to notify other classes or objects when something of interest occurs. The class that sends (or raises) the event is called the *publisher* and the classes that receive (or handle) the event are called *subscribers*. For more information about events, how they are raised and handled, see [Events](#).

- To declare events, use the [Event Statement](#).
- To raise events, use the [RaiseEvent Statement](#).
- To specify event handlers using a declarative way, use the [WithEvents](#) statement and the [Handles](#) clause.
- To be able to dynamically add, remove, and change the event handler associated with an event, use the [AddHandler Statement](#) and [RemoveHandler Statement](#) together with the [AddressOf Operator](#).

Nested classes

A class defined within another class is called *nested*. By default, the nested class is private.

```
Class Container
    Class Nested
        ' Add code here.
    End Class
End Class
```

To create an instance of the nested class, use the name of the container class followed by the dot and then followed by the name of the nested class:

```
Dim nestedInstance As Container.Nested = New Container.Nested()
```

Access modifiers and access levels

All classes and class members can specify what access level they provide to other classes by using *access modifiers*.

The following access modifiers are available:

VISUAL BASIC MODIFIER	DEFINITION
Public	The type or member can be accessed by any other code in the same assembly or another assembly that references it.

VISUAL BASIC MODIFIER	DEFINITION
Private	The type or member can only be accessed by code in the same class.
Protected	The type or member can only be accessed by code in the same class or in a derived class.
Friend	The type or member can be accessed by any code in the same assembly, but not from another assembly.
Protected Friend	The type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.

For more information, see [Access levels in Visual Basic](#).

Instantiating classes

To create an object, you need to instantiate a class, or create a class instance.

```
Dim sampleObject as New SampleClass()
```

After instantiating a class, you can assign values to the instance's properties and fields and invoke class methods.

```
' Set a property value.
sampleObject.SampleProperty = "Sample String"
' Call a method.
sampleObject.SampleMethod()
```

To assign values to properties during the class instantiation process, use object initializers:

```
Dim sampleObject = New SampleClass With
    {.FirstProperty = "A", .SecondProperty = "B"}
```

For more information, see:

- [New Operator](#)
- [Object Initializers: Named and Anonymous Types](#)

Shared classes and members

A shared member of the class is a property, procedure, or field that is shared by all instances of a class.

To define a shared member:

```
Class SampleClass
    Public Shared SampleString As String = "Sample String"
End Class
```

To access the shared member, use the name of the class without creating an object of this class:

```
MsgBox(SampleClass.SampleString)
```

Shared modules in Visual Basic have shared members only and cannot be instantiated. Shared members also cannot access non-shared properties, fields or methods

For more information, see:

- [Shared](#)
- [Module Statement](#)

Anonymous types

Anonymous types enable you to create objects without writing a class definition for the data type. Instead, the compiler generates a class for you. The class has no usable name and contains the properties you specify in declaring the object.

To create an instance of an anonymous type:

```
' sampleObject is an instance of a simple anonymous type.  
Dim sampleObject =  
    New With {Key .FirstProperty = "A", .SecondProperty = "B"}
```

For more information, see: [Anonymous Types](#).

Inheritance

Inheritance enables you to create a new class that reuses, extends, and modifies the behavior that is defined in another class. The class whose members are inherited is called the *base class*, and the class that inherits those members is called the *derived class*. However, all classes in Visual Basic implicitly inherit from the [Object](#) class that supports .NET class hierarchy and provides low-level services to all classes.

NOTE

Visual Basic doesn't support multiple inheritance. That is, you can specify only one base class for a derived class.

To inherit from a base class:

```
Class DerivedClass  
    Inherits BaseClass  
End Class
```

By default all classes can be inherited. However, you can specify whether a class must not be used as a base class, or create a class that can be used as a base class only.

To specify that a class cannot be used as a base class:

```
NotInheritable Class SampleClass  
End Class
```

To specify that a class can be used as a base class only and cannot be instantiated:

```
MustInherit Class BaseClass  
End Class
```

For more information, see:

- [Inherits Statement](#)
- [NotInheritable](#)
- [MustInherit](#)

Overriding members

By default, a derived class inherits all members from its base class. If you want to change the behavior of the inherited member, you need to override it. That is, you can define a new implementation of the method, property or event in the derived class.

The following modifiers are used to control how properties and methods are overridden:

VISUAL BASIC MODIFIER	DEFINITION
Overridable	Allows a class member to be overridden in a derived class.
Overrides	Overrides a virtual (overridable) member defined in the base class.
NotOverridable	Prevents a member from being overridden in an inheriting class.
MustOverride	Requires that a class member to be overridden in the derived class.
Shadows	Hides a member inherited from a base class

Interfaces

Interfaces, like classes, define a set of properties, methods, and events. But unlike classes, interfaces do not provide implementation. They are implemented by classes, and defined as separate entities from classes. An interface represents a contract, in that a class that implements an interface must implement every aspect of that interface exactly as it is defined.

To define an interface:

```
Public Interface ISampleInterface
    Sub DoSomething()
End Interface
```

To implement an interface in a class:

```
Class SampleClass
    Implements ISampleInterface
    Sub DoSomething
        ' Method implementation.
    End Sub
End Class
```

For more information, see:

- [Interfaces](#)
- [Interface Statement](#)
- [Implements Statement](#)

Generics

Classes, structures, interfaces and methods in .NET can include *type parameters* that define types of objects that they can store or use. The most common example of generics is a collection, where you can specify the type of objects to be stored in a collection.

To define a generic class:

```
Class SampleGeneric(Of T)
    Public Field As T
End Class
```

To create an instance of a generic class:

```
Dim sampleObject As New SampleGeneric(Of String)
sampleObject.Field = "Sample string"
```

For more information, see:

- [Generics](#)
- [Generic Types in Visual Basic](#)

Delegates

A *delegate* is a type that defines a method signature, and can provide a reference to any method with a compatible signature. You can invoke (or call) the method through the delegate. Delegates are used to pass methods as arguments to other methods.

NOTE

Event handlers are nothing more than methods that are invoked through delegates. For more information about using delegates in event handling, see [Events](#).

To create a delegate:

```
Delegate Sub SampleDelegate(ByVal str As String)
```

To create a reference to a method that matches the signature specified by the delegate:

```
Class SampleClass
    ' Method that matches the SampleDelegate signature.
    Sub SampleSub(ByVal str As String)
        ' Add code here.
    End Sub
    ' Method that instantiates the delegate.
    Sub SampleDelegateSub()
        Dim sd As SampleDelegate = AddressOf SampleSub
        sd("Sample string")
    End Sub
End Class
```

For more information, see:

- [Delegates](#)
- [Delegate Statement](#)
- [AddressOf Operator](#)

See also

- [Visual Basic Programming Guide](#)

Reflection (Visual Basic)

9/13/2019 • 2 minutes to read • [Edit Online](#)

Reflection provides objects (of type [Type](#)) that describe assemblies, modules and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you are using attributes in your code, reflection enables you to access them. For more information, see [Attributes](#).

Here's a simple example of reflection using the static method `GetType` - inherited by all types from the `Object` base class - to obtain the type of a variable:

```
' Using GetType to obtain type information:  
Dim i As Integer = 42  
Dim type As System.Type = i.GetType()  
System.Console.WriteLine(type)
```

The output is:

```
System.Int32
```

The following example uses reflection to obtain the full name of the loaded assembly.

```
' Using Reflection to get information from an Assembly:  
Dim info As System.Reflection.Assembly = GetType(System.Int32).Assembly  
System.Console.WriteLine(info)
```

The output is:

```
mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

Reflection Overview

Reflection is useful in the following situations:

- When you have to access attributes in your program's metadata. For more information, see [Retrieving Information Stored in Attributes](#).
- For examining and instantiating types in an assembly.
- For building new types at runtime. Use classes in [System.Reflection.Emit](#).
- For performing late binding, accessing methods on types created at run time. See the topic [Dynamically Loading and Using Types](#).

Related Sections

For more information:

- [Reflection](#)
- [Viewing Type Information](#)
- [Reflection and Generic Types](#)

- [System.Reflection.Emit](#)
- [Retrieving Information Stored in Attributes](#)

See also

- [Visual Basic Programming Guide](#)
- [Assemblies in .NET](#)

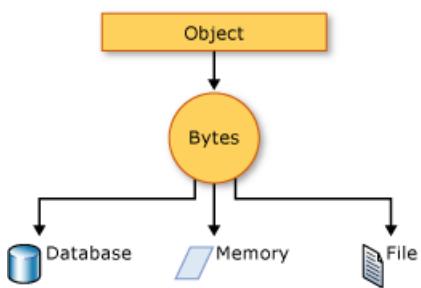
Serialization (Visual Basic)

3/25/2019 • 3 minutes to read • [Edit Online](#)

Serialization is the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

How Serialization Works

This illustration shows the overall process of serialization.



The object is serialized to a stream, which carries not just the data, but information about the object's type, such as its version, culture, and assembly name. From that stream, it can be stored in a database, a file, or memory.

Uses for Serialization

Serialization allows the developer to save the state of an object and recreate it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions like sending the object to a remote application by means of a Web Service, passing an object from one domain to another, passing an object through a firewall as an XML string, or maintaining security or user-specific information across applications.

Making an Object Serializable

To serialize an object, you need the object to be serialized, a stream to contain the serialized object, and a **Formatter**. `System.Runtime.Serialization` contains the classes necessary for serializing and deserializing objects.

Apply the `SerializableAttribute` attribute to a type to indicate that instances of this type can be serialized. A `SerializationException` exception is thrown if you attempt to serialize but the type does not have the `SerializableAttribute` attribute.

If you do not want a field within your class to be serializable, apply the `NonSerializedAttribute` attribute. If a field of a serializable type contains a pointer, a handle, or some other data structure that is specific to a particular environment, and the field cannot be meaningfully reconstituted in a different environment, then you may want to make it nonserializable.

If a serialized class contains references to objects of other classes that are marked `SerializableAttribute`, those objects will also be serialized.

Binary and XML Serialization

Either binary or XML serialization can be used. In binary serialization, all members, even those that are read-only, are serialized, and performance is enhanced. XML serialization provides more readable code, as well as greater flexibility of object sharing and usage for interoperability purposes.

Binary Serialization

Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-based network streams.

XML Serialization

XML serialization serializes the public fields and properties of an object, or the parameters and return values of methods, into an XML stream that conforms to a specific XML Schema definition language (XSD) document. XML serialization results in strongly typed classes with public properties and fields that are converted to XML.

[System.Xml.Serialization](#) contains the classes necessary for serializing and deserializing XML.

You can apply attributes to classes and class members in order to control the way the [XmlSerializer](#) serializes or deserializes an instance of the class.

Basic and Custom Serialization

Serialization can be performed in two ways, basic and custom. Basic serialization uses the .NET Framework to automatically serialize the object.

Basic Serialization

The only requirement in basic serialization is that the object has the [SerializableAttribute](#) attribute applied. The [NonSerializedAttribute](#) can be used to keep specific fields from being serialized.

When you use basic serialization, the versioning of objects may create problems, in which case custom serialization may be preferable. Basic serialization is the easiest way to perform serialization, but it does not provide much control over the process.

Custom Serialization

In custom serialization, you can specify exactly which objects will be serialized and how it will be done. The class must be marked [SerializableAttribute](#) and implement the [ISerializable](#) interface.

If you want your object to be deserialized in a custom manner as well, you must use a custom constructor.

Designer Serialization

Designer serialization is a special form of serialization that involves the kind of object persistence usually associated with development tools. Designer serialization is the process of converting an object graph into a source file that can later be used to recover the object graph. A source file can contain code, markup, or even SQL table information.

Related Topics and Examples

[Walkthrough: Persisting an Object in Visual Studio \(Visual Basic\)](#)

Demonstrates how serialization can be used to persist an object's data between instances, allowing you to store values and retrieve them the next time the object is instantiated.

[How to: Read Object Data from an XML File \(Visual Basic\)](#)

Shows how to read object data that was previously written to an XML file using the [XmlSerializer](#) class.

[How to: Write Object Data to an XML File \(Visual Basic\)](#)

Shows how to write the object from a class to an XML file using the [XmlSerializer](#) class.

Program Structure and Code Conventions (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

This section introduces the typical Visual Basic program structure, provides a simple Visual Basic program, "Hello, World", and discusses Visual Basic code conventions. Code conventions are suggestions that focus not on a program's logic but on its physical structure and appearance. Following them makes your code easier to read, understand, and maintain. Code conventions can include, among others:

- Standardized formats for labeling and commenting code.
- Guidelines for spacing, formatting, and indenting code.
- Naming conventions for objects, variables, and procedures.

The following topics present a set of programming guidelines for Visual Basic programs, along with examples of good usage.

In This Section

[Structure of a Visual Basic Program](#)

Provides an overview of the elements that make up a Visual Basic program.

[Main Procedure in Visual Basic](#)

Discusses the procedure that serves as the starting point and overall control for your application.

[References and the Imports Statement](#)

Discusses how to reference objects in other assemblies.

[Namespaces in Visual Basic](#)

Describes how namespaces organize objects within assemblies.

[Visual Basic Naming Conventions](#)

Includes general guidelines for naming procedures, constants, variables, arguments, and objects.

[Visual Basic Coding Conventions](#)

Reviews the guidelines used in developing the samples in this documentation.

[Conditional Compilation](#)

Describes how to compile particular blocks of code selectively while directing the compiler to ignore others.

[How to: Break and Combine Statements in Code](#)

Shows how to divide long statements into multiple lines and combine short statements on one line.

[How to: Collapse and Hide Sections of Code](#)

Shows how to collapse and hide sections of code in the Visual Basic code editor.

[How to: Label Statements](#)

Shows how to mark a line of code to identify it for use with statements such as `On Error Goto`.

[Special Characters in Code](#)

Shows how and where to use non-numeric and non-alphabetic characters.

[Comments in Code](#)

Discusses how to add descriptive comments to your code.

[Keywords as Element Names in Code](#)

Describes how to use brackets ([]) to delimit variable names that are also Visual Basic keywords.

[Me, My, MyBase, and MyClass](#)

Describes various ways to refer to elements of a Visual Basic program.

[Visual Basic Limitations](#)

Discusses the removal of known coding limits within Visual Basic.

Related Sections

[Typographic and Code Conventions](#)

Provides standard coding conventions for Visual Basic.

[Writing Code](#)

Describes features that make it easier for you to write and manage your code.

Structure of a Visual Basic Program

4/28/2019 • 3 minutes to read • [Edit Online](#)

A Visual Basic program is built up from standard building blocks. A *solution* comprises one or more projects. A *project* in turn can contain one or more assemblies. Each *assembly* is compiled from one or more source files. A *source file* provides the definition and implementation of classes, structures, modules, and interfaces, which ultimately contain all your code.

For more information about these building blocks of a Visual Basic program, see [Solutions and Projects](#) and [Assemblies in .NET](#).

File-Level Programming Elements

When you start a project or file and open the code editor, you see some code already in place and in the correct order. Any code that you write should follow the following sequence:

1. `Option` statements
2. `Imports` statements
3. `Namespace` statements and namespace-level elements

If you enter statements in a different order, compilation errors can result.

A program can also contain conditional compilation statements. You can intersperse these in the source file among the statements of the preceding sequence.

Option Statements

`Option` statements establish ground rules for subsequent code, helping prevent syntax and logic errors. The [Option Explicit Statement](#) ensures that all variables are declared and spelled correctly, which reduces debugging time. The [Option Strict Statement](#) helps to minimize logic errors and data loss that can occur when you work between variables of different data types. The [Option Compare Statement](#) specifies the way strings are compared to each other, based on either their `Binary` or `Text` values.

Imports Statements

You can include an [Imports Statement \(.NET Namespace and Type\)](#) to import names defined outside your project. An `Imports` statement allows your code to refer to classes and other types defined within the imported namespace, without having to qualify them. You can use as many `Imports` statements as appropriate. For more information, see [References and the Imports Statement](#).

Namespace Statements

Namespaces help you organize and classify your programming elements for ease of grouping and accessing. You use the [Namespace Statement](#) to classify the following statements within a particular namespace. For more information, see [Namespaces in Visual Basic](#).

Conditional Compilation Statements

Conditional compilation statements can appear almost anywhere in your source file. They cause parts of your code to be included or excluded at compile time depending on certain conditions. You can also use them for debugging your application, because conditional code runs in debugging mode only. For more information, see [Conditional Compilation](#).

Namespace-Level Programming Elements

Classes, structures, and modules contain all the code in your source file. They are *namespace-level* elements, which can appear within a namespace or at the source file level. They hold the declarations of all other programming elements. Interfaces, which define element signatures but provide no implementation, also appear at module level. For more information on the module-level elements, see the following:

- [Class Statement](#)
- [Structure Statement](#)
- [Module Statement](#)
- [Interface Statement](#)

Data elements at namespace level are enumerations and delegates.

Module-Level Programming Elements

Procedures, operators, properties, and events are the only programming elements that can hold executable code (statements that perform actions at run time). They are the *module-level* elements of your program. For more information on the procedure-level elements, see the following:

- [Function Statement](#)
- [Sub Statement](#)
- [Declare Statement](#)
- [Operator Statement](#)
- [Property Statement](#)
- [Event Statement](#)

Data elements at module level are variables, constants, enumerations, and delegates.

Procedure-Level Programming Elements

Most of the contents of *procedure-level* elements are executable statements, which constitute the run-time code of your program. All executable code must be in some procedure (`Function`, `Sub`, `Operator`, `Get`, `Set`, `AddHandler`, `RemoveHandler`, `RaiseEvent`). For more information, see [Statements](#).

Data elements at procedure level are limited to local variables and constants.

The Main Procedure

The `Main` procedure is the first code to run when your application has been loaded. `Main` serves as the starting point and overall control for your application. There are four varieties of `Main`:

- `Sub Main()`
- `Sub Main(ByVal cmdArgs() As String)`
- `Function Main() As Integer`
- `Function Main(ByVal cmdArgs() As String) As Integer`

The most common variety of this procedure is `Sub Main()`. For more information, see [Main Procedure in Visual Basic](#).

See also

- [Main Procedure in Visual Basic](#)
- [Visual Basic Naming Conventions](#)
- [Visual Basic Limitations](#)

Main Procedure in Visual Basic

8/22/2019 • 3 minutes to read • [Edit Online](#)

Every Visual Basic application must contain a procedure called `Main`. This procedure serves as the starting point and overall control for your application. The .NET Framework calls your `Main` procedure when it has loaded your application and is ready to pass control to it. Unless you are creating a Windows Forms application, you must write the `Main` procedure for applications that run on their own.

`Main` contains the code that runs first. In `Main`, you can determine which form is to be loaded first when the program starts, find out if a copy of your application is already running on the system, establish a set of variables for your application, or open a database that the application requires.

Requirements for the Main Procedure

A file that runs on its own (usually with extension .exe) must contain a `Main` procedure. A library (for example with extension .dll) does not run on its own and does not require a `Main` procedure. The requirements for the different types of projects you can create are as follows:

- Console applications run on their own, and you must supply at least one `Main` procedure.
- Windows Forms applications run on their own. However, the Visual Basic compiler automatically generates a `Main` procedure in such an application, and you do not need to write one.
- Class libraries do not require a `Main` procedure. These include Windows Control Libraries and Web Control Libraries. Web applications are deployed as class libraries.

Declaring the Main Procedure

There are four ways to declare the `Main` procedure. It can take arguments or not, and it can return a value or not.

NOTE

If you declare `Main` in a class, you must use the `Shared` keyword. In a module, `Main` does not need to be `Shared`.

- The simplest way is to declare a `Sub` procedure that does not take arguments or return a value.

```
Module mainModule
    Sub Main()
        MsgBox("The Main procedure is starting the application.")
        ' Insert call to appropriate starting place in your code.
        MsgBox("The application is terminating.")
    End Sub
End Module
```

- `Main` can also return an `Integer` value, which the operating system uses as the exit code for your program. Other programs can test this code by examining the Windows ERRORLEVEL value. To return an exit code, you must declare `Main` as a `Function` procedure instead of a `Sub` procedure.

```

Module mainModule
    Function Main() As Integer
        MsgBox("The Main procedure is starting the application.")
        Dim returnValue As Integer = 0
        ' Insert call to appropriate starting place in your code.
        ' On return, assign appropriate value to returnValue.
        ' 0 usually means successful completion.
        MsgBox("The application is terminating with error level " &
            CStr(returnValue) & ".")
        Return returnValue
    End Function
End Module

```

- `Main` can also take a `String` array as an argument. Each string in the array contains one of the command-line arguments used to invoke your program. You can take different actions depending on their values.

```

Module mainModule
    Function Main(ByVal cmdArgs() As String) As Integer
        MsgBox("The Main procedure is starting the application.")
        Dim returnValue As Integer = 0
        ' See if there are any arguments.
        If cmdArgs.Length > 0 Then
            For argNum As Integer = 0 To UBound(cmdArgs, 1)
                ' Insert code to examine cmdArgs(argNum) and take
                ' appropriate action based on its value.
            Next
        End If
        ' Insert call to appropriate starting place in your code.
        ' On return, assign appropriate value to returnValue.
        ' 0 usually means successful completion.
        MsgBox("The application is terminating with error level " &
            CStr(returnValue) & ".")
        Return returnValue
    End Function
End Module

```

- You can declare `Main` to examine the command-line arguments but not return an exit code, as follows.

```

Module mainModule
    Sub Main(ByVal cmdArgs() As String)
        MsgBox("The Main procedure is starting the application.")
        Dim returnValue As Integer = 0
        ' See if there are any arguments.
        If cmdArgs.Length > 0 Then
            For argNum As Integer = 0 To UBound(cmdArgs, 1)
                ' Insert code to examine cmdArgs(argNum) and take
                ' appropriate action based on its value.
            Next
        End If
        ' Insert call to appropriate starting place in your code.
        MsgBox("The application is terminating.")
    End Sub
End Module

```

See also

- [MsgBox](#)
- [Length](#)
- [UBound](#)
- [Structure of a Visual Basic Program](#)

- [/main](#)
- [Shared](#)
- [Sub Statement](#)
- [Function Statement](#)
- [Integer Data Type](#)
- [String Data Type](#)

References and the Imports Statement (Visual Basic)

9/13/2019 • 2 minutes to read • [Edit Online](#)

You can make external objects available to your project by choosing the **Add Reference** command on the **Project** menu. References in Visual Basic can point to assemblies, which are like type libraries but contain more information.

The Imports Statement

Assemblies include one or more namespaces. When you add a reference to an assembly, you can also add an `Imports` statement to a module that controls the visibility of that assembly's namespaces within the module. The `Imports` statement provides a scoping context that lets you use only the portion of the namespace necessary to supply a unique reference.

The `Imports` statement has the following syntax:

```
Imports [Aliasname =] Namespace
```

`Aliasname` refers to a short name you can use within code to refer to an imported namespace. `Namespace` is a namespace available through either a project reference, through a definition within the project, or through a previous `Imports` statement.

A module may contain any number of `Imports` statements. They must appear after any `Option` statements, if present, but before any other code.

NOTE

Do not confuse project references with the `Imports` statement or the `Declare` statement. Project references make external objects, such as objects in assemblies, available to Visual Basic projects. The `Imports` statement is used to simplify access to project references, but does not provide access to these objects. The `Declare` statement is used to declare a reference to an external procedure in a dynamic-link library (DLL).

Using Aliases with the Imports Statement

The `Imports` statement makes it easier to access methods of classes by eliminating the need to explicitly type the fully qualified names of references. Aliases let you assign a friendlier name to just one part of a namespace. For example, the carriage return/line feed sequence that causes a single piece of text to be displayed on multiple lines is part of the `ControlChars` module in the `Microsoft.VisualBasic` namespace. To use this constant in a program without an alias, you would need to type the following code:

```
MsgBox("Some text" & Microsoft.VisualBasic.ControlChars.CrLf &
      "Some more text")
```

`Imports` statements must always be the first lines immediately following any `Option` statements in a module. The following code fragment shows how to import and assign an alias to the `Microsoft.VisualBasic.ControlChars` module:

```
Imports CtrlChrs = Microsoft.VisualBasic.ControlChars
```

Future references to this namespace can be considerably shorter:

```
MsgBox("Some text" & CtrlChrs.CrLf & "Some more text")
```

If an `Imports` statement does not include an alias name, elements defined within the imported namespace can be used in the module without qualification. If the alias name is specified, it must be used as a qualifier for names contained within that namespace.

See also

- [ControlChars](#)
- [Microsoft.VisualBasic](#)
- [Namespaces in Visual Basic](#)
- [Assemblies in .NET](#)
- [Imports Statement \(.NET Namespace and Type\)](#)

Namespaces in Visual Basic

9/13/2019 • 5 minutes to read • [Edit Online](#)

Namespaces organize the objects defined in an assembly. Assemblies can contain multiple namespaces, which can in turn contain other namespaces. Namespaces prevent ambiguity and simplify references when using large groups of objects such as class libraries.

For example, the .NET Framework defines the `ListBox` class in the `System.Windows.Forms` namespace. The following code fragment shows how to declare a variable using the fully qualified name for this class:

```
Dim LBox As System.Windows.Forms.ListBox
```

Avoiding Name Collisions

.NET Framework namespaces address a problem sometimes called *namespace pollution*, in which the developer of a class library is hampered by the use of similar names in another library. These conflicts with existing components are sometimes called *name collisions*.

For example, if you create a new class named `ListBox`, you can use it inside your project without qualification. However, if you want to use the .NET Framework `ListBox` class in the same project, you must use a fully qualified reference to make the reference unique. If the reference is not unique, Visual Basic produces an error stating that the name is ambiguous. The following code example demonstrates how to declare these objects:

```
' Define a new object based on your ListBox class.  
Dim LBC As New ListBox  
' Define a new Windows.Forms ListBox control.  
Dim MyLB As New System.Windows.Forms.ListBox
```

The following illustration shows two namespace hierarchies, both containing an object named `ListBox`:



By default, every executable file you create with Visual Basic contains a namespace with the same name as your project. For example, if you define an object within a project named `ListBoxProject`, the executable file `ListBoxProject.exe` contains a namespace called `ListBoxProject`.

Multiple assemblies can use the same namespace. Visual Basic treats them as a single set of names. For example, you can define classes for a namespace called `SomeNameSpace` in an assembly named `Assemb1`, and define additional classes for the same namespace from an assembly named `Assemb2`.

Fully Qualified Names

Fully qualified names are object references that are prefixed with the name of the namespace in which the object is defined. You can use objects defined in other projects if you create a reference to the class (by choosing **Add Reference** from the **Project** menu) and then use the fully qualified name for the object in your code. The following code fragment shows how to use the fully qualified name for an object from another project's namespace:

```
Dim LBC As New ListBoxProject.Form1.ListBox
```

Fully qualified names prevent naming conflicts because they make it possible for the compiler to determine which object is being used. However, the names themselves can get long and cumbersome. To get around this, you can use the `Imports` statement to define an *alias*—an abbreviated name you can use in place of a fully qualified name. For example, the following code example creates aliases for two fully qualified names, and uses these aliases to define two objects.

```
Imports LBControl = System.Windows.Forms.ListBox  
Imports MyListBox = ListBoxProject.Form1.ListBox
```

```
Dim LBC As LBControl  
Dim MyLB As MyListBox
```

If you use the `Imports` statement without an alias, you can use all the names in that namespace without qualification, provided they are unique to the project. If your project contains `Imports` statements for namespaces that contain items with the same name, you must fully qualify that name when you use it. Suppose, for example, your project contained the following two `Imports` statements:

```
' This namespace contains a class called Class1.  
Imports MyProj1  
' This namespace also contains a class called Class1.  
Imports MyProj2
```

If you attempt to use `Class1` without fully qualifying it, Visual Basic produces an error stating that the name `Class1` is ambiguous.

Namespace Level Statements

Within a namespace, you can define items such as modules, interfaces, classes, delegates, enumerations, structures, and other namespaces. You cannot define items such as properties, procedures, variables and events at the namespace level. These items must be declared within containers such as modules, structures, or classes.

Global Keyword in Fully Qualified Names

If you have defined a nested hierarchy of namespaces, code inside that hierarchy might be blocked from accessing the `System` namespace of the .NET Framework. The following example illustrates a hierarchy in which the `SpecialSpace.System` namespace blocks access to `System`.

```
Namespace SpecialSpace  
    Namespace System  
        Class abc  
            Function getValue() As System.Int32  
                Dim n As System.Int32  
                Return n  
            End Function  
        End Class  
    End Namespace  
End Namespace
```

As a result, the Visual Basic compiler cannot successfully resolve the reference to `System.Int32`, because `SpecialSpace.System` does not define `Int32`. You can use the `Global` keyword to start the qualification chain at

the outermost level of the .NET Framework class library. This allows you to specify the [System](#) namespace or any other namespace in the class library. The following example illustrates this.

```
Namespace SpecialSpace
    Namespace System
        Class abc
            Function getValue() As Global.System.Int32
                Dim n As Global.System.Int32
                Return n
            End Function
        End Class
    End Namespace
End Namespace
```

You can use `Global` to access other root-level namespaces, such as [Microsoft.VisualBasic](#), and any namespace associated with your project.

Global Keyword in Namespace Statements

You can also use the `Global` keyword in a [Namespace Statement](#). This lets you define a namespace out of the root namespace of your project.

All namespaces in your project are based on the root namespace for the project. Visual Studio assigns your project name as the default root namespace for all code in your project. For example, if your project is named `ConsoleApplication1`, its programming elements belong to namespace `ConsoleApplication1`. If you declare `Namespace Magnetosphere`, references to `Magnetosphere` in the project will access `ConsoleApplication1.Magnetosphere`.

The following examples use the `Global` keyword to declare a namespace out of the root namespace for the project.

```
Namespace Global.Magnetosphere
End Namespace

Namespace Global
    Namespace Magnetosphere
        End Namespace
    End Namespace
```

In a namespace declaration, `Global` cannot be nested in another namespace.

You can use the [Application Page, Project Designer \(Visual Basic\)](#) to view and modify the **Root Namespace** of the project. For new projects, the **Root Namespace** defaults to the project name. To cause `Global` to be the top-level namespace, you can clear the **Root Namespace** entry so that the box is empty. Clearing **Root Namespace** removes the need for the `Global` keyword in namespace declarations.

If a `Namespace` statement declares a name that is also a namespace in the .NET Framework, the .NET Framework namespace becomes unavailable if the `Global` keyword is not used in a fully qualified name. To enable access to that .NET Framework namespace without using the `Global` keyword, you can include the `Global` keyword in the `Namespace` statement.

The following example has the `Global` keyword in the `System.Text` namespace declaration.

If the `Global` keyword was not present in the namespace declaration, [StringBuilder](#) could not be accessed without specifying `Global.System.Text.StringBuilder`. For a project named `ConsoleApplication1`, references to

`System.Text` would access `ConsoleApplication1.System.Text` if the `Global` keyword was not used.

```
Module Module1
    Sub Main()
        Dim encoding As New System.Text.TitanEncoding

        ' If the namespace defined below is System.Text
        ' instead of Global.System.Text, then this statement
        ' causes a compile-time error.
        Dim sb As New System.Text.StringBuilder
    End Sub
End Module

Namespace Global.System.Text
    Class TitanEncoding

    End Class
End Namespace
```

See also

- [ListBox](#)
- [System.Windows.Forms](#)
- [Assemblies in .NET](#)
- [References and the Imports Statement](#)
- [Imports Statement \(.NET Namespace and Type\)](#)
- [Writing Code in Office Solutions](#)

Visual Basic Naming Conventions

4/28/2019 • 2 minutes to read • [Edit Online](#)

When you name an element in your Visual Basic application, the first character of that name must be an alphabetic character or an underscore. Note, however, that names beginning with an underscore are not compliant with the [Language Independence and Language-Independent Components \(CLS\)](#).

The following suggestions apply to naming.

- Begin each separate word in a name with a capital letter, as in `FindLastRecord` and `RedrawMyForm`.
- Begin function and method names with a verb, as in `InitNameArray` or `CloseDialog`.
- Begin class, structure, module, and property names with a noun, as in `EmployeeName` or `CarAccessory`.
- Begin interface names with the prefix "I", followed by a noun or a noun phrase, like `IComponent`, or with an adjective describing the interface's behavior, like `IPersistable`. Do not use the underscore, and use abbreviations sparingly, because abbreviations can cause confusion.
- Begin event handler names with a noun describing the type of event followed by the "`EventHandler`" suffix, as in "`MouseEventHandler`".
- In names of event argument classes, include the "`EventArgs`" suffix.
- If an event has a concept of "before" or "after," use a suffix in present or past tense, as in '`ControlAdd`' or "`ControlAdded`".
- For long or frequently used terms, use abbreviations to keep name lengths reasonable, for example, "HTML", instead of "Hypertext Markup Language". In general, variable names greater than 32 characters are difficult to read on a monitor set to a low resolution. Also, make sure your abbreviations are consistent throughout the entire application. Randomly switching in a project between "HTML" and "Hypertext Markup Language" can lead to confusion.
- Avoid using names in an inner scope that are the same as names in an outer scope. Errors can result if the wrong variable is accessed. If a conflict occurs between a variable and the keyword of the same name, you must identify the keyword by preceding it with the appropriate type library. For example, if you have a variable called `Date`, you can use the intrinsic `Date` function only by calling `DateTime.Date`.

See also

- [Keywords as Element Names in Code](#)
- [Me, My, MyBase, and MyClass](#)
- [Declared Element Names](#)
- [Program Structure and Code Conventions](#)
- [Visual Basic Language Reference](#)

Visual Basic Coding Conventions

10/7/2019 • 6 minutes to read • [Edit Online](#)

Microsoft develops samples and documentation that follow the guidelines in this topic. If you follow the same coding conventions, you may gain the following benefits:

- Your code will have a consistent look, so that readers can better focus on content, not layout.
- Readers understand your code more quickly because they can make assumptions based on previous experience.
- You can copy, change, and maintain the code more easily.
- You help ensure that your code demonstrates "best practices" for Visual Basic.

Naming Conventions

- For information about naming guidelines, see [Naming Guidelines](#) topic.
- Do not use "My" or "my" as part of a variable name. This practice creates confusion with the `My` objects.
- You do not have to change the names of objects in auto-generated code to make them fit the guidelines.

Layout Conventions

- Insert tabs as spaces, and use smart indenting with four-space indents.
- Use **Pretty listing (reformatting) of code** to reformat your code in the code editor. For more information, see [Options, Text Editor, Basic \(Visual Basic\)](#).
- Use only one statement per line. Don't use the Visual Basic line separator character (:).
- Avoid using the explicit line continuation character "_" in favor of implicit line continuation wherever the language allows it.
- Use only one declaration per line.
- If **Pretty listing (reformatting) of code** doesn't format continuation lines automatically, manually indent continuation lines one tab stop. However, always left-align items in a list.

```
a As Integer,  
b As Integer
```

- Add at least one blank line between method and property definitions.

Commenting Conventions

- Put comments on a separate line instead of at the end of a line of code.
- Start comment text with an uppercase letter, and end comment text with a period.
- Insert one space between the comment delimiter ('') and the comment text.

```
' Here is a comment.
```

- Do not surround comments with formatted blocks of asterisks.

Program Structure

- When you use the `Main` method, use the default construct for new console applications, and use `My` for command-line arguments.

```
Sub Main()
    For Each argument As String In My.Application.CommandLineArgs
        ' Add code here to use the string variable.
    Next
End Sub
```

Language Guidelines

String Data Type

- Use [string interpolation](#) to concatenate short strings, as shown in the following code.

```
MsgBox($"hello{vbCrLf}goodbye")
```

- To append strings in loops, use the [StringBuilder](#) object.

```
Dim longString As New System.Text.StringBuilder
For count As Integer = 1 To 1000
    longString.Append(count)
Next
```

Relaxed Delegates in Event Handlers

Do not explicitly qualify the arguments (`Object` and `EventArgs`) to event handlers. If you are not using the event arguments that are passed to an event (for example, `sender` as `Object`, `e` as `EventArgs`), use relaxed delegates, and leave out the event arguments in your code:

```
Public Sub Form1_Load() Handles Form1.Load
End Sub
```

Unsigned Data Type

- Use `Integer` rather than unsigned types, except where they are necessary.

Arrays

- Use the short syntax when you initialize arrays on the declaration line. For example, use the following syntax.

```
Dim letters1 As String() = {"a", "b", "c"}
```

Do not use the following syntax.

```
Dim letters2() As String = New String() {"a", "b", "c"}
```

- Put the array designator on the type, not on the variable. For example, use the following syntax:

```
Dim letters4 As String() = {"a", "b", "c"}
```

Do not use the following syntax:

```
Dim letters3() As String = {"a", "b", "c"}
```

- Use the {} syntax when you declare and initialize arrays of basic data types. For example, use the following syntax:

```
Dim letters5 As String() = {"a", "b", "c"}
```

Do not use the following syntax:

```
Dim letters6(2) As String  
letters6(0) = "a"  
letters6(1) = "b"  
letters6(2) = "c"
```

Use the With Keyword

When you make a series of calls to one object, consider using the `With` keyword:

```
With orderLog  
    .Log = "Application"  
    .Source = "Application Name"  
    .MachineName = "Computer Name"  
End With
```

Use the Try...Catch and Using Statements when you use Exception Handling

Do not use `On Error Goto`.

Use the IsNot Keyword

Use the `IsNot` keyword instead of `Not...Is Nothing`.

New Keyword

- Use short instantiation. For example, use the following syntax:

```
Dim employees As New List(Of String)
```

The preceding line is equivalent to this:

```
Dim employees2 As List(Of String) = New List(Of String)
```

- Use object initializers for new objects instead of the parameterless constructor:

```
Dim orderLog As New EventLog With {  
    .Log = "Application",  
    .Source = "Application Name",  
    .MachineName = "Computer Name"}
```

Event Handling

- Use `Handles` rather than `AddHandler`:

```
Private Sub ToolStripMenuItem1_Click() Handles ToolStripMenuItem1.Click
End Sub
```

- Use `AddressOf`, and do not instantiate the delegate explicitly:

```
Dim closeItem As New ToolStripMenuItem(
    "Close", Nothing, AddressOf ToolStripMenuItem1_Click)
Me.MainMenuStrip.Items.Add(closeItem)
```

- When you define an event, use the short syntax, and let the compiler define the delegate:

```
Public Event SampleEvent As EventHandler(Of SampleEventArgs)
' or
Public Event SampleEvent(ByVal source As Object,
    ByVal e As SampleEventArgs)
```

- Do not verify whether an event is `Nothing` (null) before you call the `RaiseEvent` method. `RaiseEvent` checks for `Nothing` before it raises the event.

Using Shared Members

Call `Shared` members by using the class name, not from an instance variable.

Use XML Literals

XML literals simplify the most common tasks that you encounter when you work with XML (for example, load, query, and transform). When you develop with XML, follow these guidelines:

- Use XML literals to create XML documents and fragments instead of calling XML APIs directly.
- Import XML namespaces at the file or project level to take advantage of the performance optimizations for XML literals.
- Use the XML axis properties to access elements and attributes in an XML document.
- Use embedded expressions to include values and to create XML from existing values instead of using API calls such as the `Add` method:

```

Private Function GetHtmlDocument(
    ByVal items As IEnumerable(Of XElement)) As String

    Dim htmlDoc = <html>
        <body>
            <table border="0" cellspacing="2">
                <%=
                    From item In items
                    Select <tr>
                        <td style="width:480">
                            <%= item.<title>.Value %>
                        </td>
                        <td><%= item.<pubDate>.Value %></td>
                    </tr>
                %>
            </table>
        </body>
    </html>

    Return htmlDoc.ToString()
End Function

```

LINQ Queries

- Use meaningful names for query variables:

```

Dim seattleCustomers = From cust In customers
    Where cust.City = "Seattle"

```

- Provide names for elements in a query to make sure that property names of anonymous types are correctly capitalized using Pascal casing:

```

Dim customerOrders = From customer In customers
    Join order In orders
        On customer.CustomerID Equals order.CustomerID
    Select Customer = customer, Order = order

```

- Rename properties when the property names in the result would be ambiguous. For example, if your query returns a customer name and an order ID, rename them instead of leaving them as `Name` and `ID` in the result:

```

Dim customerOrders2 = From cust In customers
    Join ord In orders
        On cust.CustomerID Equals ord.CustomerID
    Select CustomerName = cust.Name,
        OrderID = ord.ID

```

- Use type inference in the declaration of query variables and range variables:

```

Dim customerList = From cust In customers

```

- Align query clauses under the `From` statement:

```

Dim newyorkCustomers = From cust In customers
    Where cust.City = "New York"
    Select cust.LastName, cust.CompanyName

```

- Use `Where` clauses before other query clauses so that later query clauses operate on the filtered set of data:

```
Dim newyorkCustomers2 = From cust In customers
    Where cust.City = "New York"
    Order By cust.LastName
```

- Use the `Join` clause to explicitly define a join operation instead of using the `Where` clause to implicitly define a join operation:

```
Dim customerList2 = From cust In customers
    Join order In orders
        On cust.CustomerID Equals order.CustomerID
    Select cust, order
```

See also

- [Secure Coding Guidelines](#)

Conditional Compilation in Visual Basic

8/22/2019 • 2 minutes to read • [Edit Online](#)

In *conditional compilation*, particular blocks of code in a program are compiled selectively while others are ignored.

For example, you may want to write debugging statements that compare the speed of different approaches to the same programming task, or you may want to localize an application for multiple languages. Conditional compilation statements are designed to run during compile time, not at run time.

You denote blocks of code to be conditionally compiled with the `#If...Then...#Else` directive. For example, to create French- and German-language versions of the same application from the same source code, you embed platform-specific code segments in `#If...Then` statements using the predefined constants `FrenchVersion` and `GermanVersion`. The following example demonstrates how:

```
#If FrenchVersion Then
    ' <code specific to the French language version>.
#ElseIf GermanVersion Then
    ' <code specific to the German language version>.
#Else
    ' <code specific to other versions>.
#End If
```

If you set the value of the `FrenchVersion` conditional compilation constant to `True` at compile time, the conditional code for the French version is compiled. If you set the value of the `GermanVersion` constant to `True`, the compiler uses the German version. If neither is set to `True`, the code in the last `Else` block runs.

NOTE

Autocompletion will not function when editing code and using conditional compilation directives if the code is not part of the current branch.

Declaring Conditional Compilation Constants

You can set conditional compilation constants in one of three ways:

- In the **Project Designer**
- At the command line when using the command-line compiler
- In your code

Conditional compilation constants have a special scope and cannot be accessed from standard code. The scope of a conditional compilation constant is dependent on the way it is set. The following table lists the scope of constants declared using each of the three ways mentioned above.

HOW CONSTANT IS SET	SCOPE OF CONSTANT
Project Designer	Public to all files in the project
Command line	Public to all files passed to the command-line compiler

HOW CONSTANT IS SET	SCOPE OF CONSTANT
#Const statement in code	Private to the file in which it is declared
TO SET CONSTANTS IN THE PROJECT DESIGNER	
- Before creating your executable file, set constants in the Project Designer by following the steps provided in Managing Project and Solution Properties .	
TO SET CONSTANTS AT THE COMMAND LINE	
<p>- Use the /d switch to enter conditional compilation constants, as in the following example:</p> <pre>vbc MyProj.vb /d:conFrenchVersion=-1:conANSI=0</pre> <p>No space is required between the /d switch and the first constant. For more information, see /define (Visual Basic). Command-line declarations override declarations entered in the Project Designer, but do not erase them. Arguments set in Project Designer remain in effect for subsequent compilations.</p> <p>When writing constants in the code itself, there are no strict rules as to their placement, since their scope is the entire module in which they are declared.</p>	
TO SET CONSTANTS IN YOUR CODE	
<p>- Place the constants in the declaration block of the module in which they are used. This helps keep your code organized and easier to read.</p>	

Related Topics

TITLE	DESCRIPTION
Program Structure and Code Conventions	Provides suggestions for making your code easy to read and maintain.

Reference

[#Const Directive](#)

[#If...Then...#Else Directives](#)

[/define \(Visual Basic\)](#)

How to: Break and Combine Statements in Code (Visual Basic)

9/14/2019 • 2 minutes to read • [Edit Online](#)

When writing your code, you might at times create lengthy statements that necessitate horizontal scrolling in the Code Editor. Although this doesn't affect the way your code runs, it makes it difficult for you or anyone else to read the code as it appears on the monitor. In such cases, you should consider breaking the single long statement into several lines.

To break a single statement into multiple lines

Use the line-continuation character, which is an underscore (`_`), at the point at which you want the line to break. The underscore must be immediately preceded by a space and immediately followed by a line terminator (carriage return) or (starting with version 16.0) a comment followed by a carriage return.

NOTE

In some cases, if you omit the line-continuation character, the Visual Basic compiler will implicitly continue the statement on the next line of code. For a list of syntax elements for which you can omit the line-continuation character, see "Implicit Line Continuation" in [Statements](#).

In the following example, the statement is broken into four lines with line-continuation characters terminating all but the last line.

```
cmd.CommandText = _  
    "SELECT * FROM Titles JOIN Publishers " _  
    & "ON Publishers.PubId = Titles.PubID " _  
    & "WHERE Publishers.State = 'CA'"
```

Using this sequence makes your code easier to read, both online and when printed.

The line-continuation character must be the last character on a line. You can't follow it with anything else on the same line.

Some limitations exist as to where you can use the line-continuation character; for example, you can't use it in the middle of an argument name. You can break an argument list with the line-continuation character, but the individual names of the arguments must remain intact.

You can't continue a comment by using a line-continuation character. The compiler doesn't examine the characters in a comment for special meaning. For a multiple-line comment, repeat the comment symbol (`'`) on each line.

Although placing each statement on a separate line is the recommended method, Visual Basic also allows you to place multiple statements on the same line.

To place multiple statements on the same line

Separate the statements with a colon (`:`), as in the following example:

```
text1.Text = "Hello" : text1.BackColor = System.Drawing.Color.Red
```

See also

- [Program Structure and Code Conventions](#)
- [Statements](#)

How to: Collapse and Hide Sections of Code (Visual Basic)

9/17/2019 • 2 minutes to read • [Edit Online](#)

The `#Region` directive enables you to collapse and hide sections of code in Visual Basic files. The `#Region` directive lets you specify a block of code that you can expand or collapse when using the Visual Studio code editor. The ability to hide code selectively makes your files more manageable and easier to read. For more information, see [Outlining](#).

`#Region` directives support code block semantics such as `#If...#End If`. This means they cannot begin in one block and end in another; the start and end must be in the same block. `#Region` directives are not supported within functions.

To collapse and hide a section of code

Place the section of code between the `#Region` and `#End Region` statements, as in the following example:

```
#Region "This is the code to be collapsed"
    Private components As System.ComponentModel.Container
    Dim WithEvents Form1 As System.Windows.Forms.Form

    Private Sub InitializeComponent()
        components = New System.ComponentModel.Container
        Me.Text = "Form1"
    End Sub
#End Region
```

The `#Region` block can be used multiple times in a code file; thus, users can define their own blocks of procedures and classes that can, in turn, be collapsed. `#Region` blocks can also be nested within other `#Region` blocks.

NOTE

Hiding code does not prevent it from being compiled and does not affect `#If...#End If` statements.

See also

- [Conditional Compilation](#)
- [#Region Directive](#)
- [#If...Then...#Else Directives](#)
- [Outlining](#)

How to: Label Statements (Visual Basic)

9/17/2019 • 2 minutes to read • [Edit Online](#)

Statement blocks are made up of lines of code delimited by colons. Lines of code preceded by an identifying string or integer are said to be *labeled*. Statement labels are used to mark a line of code to identify it for use with statements such as `On Error Goto`.

Labels may be either valid Visual Basic identifiers—such as those that identify programming elements—or integer literals. A label must appear at the beginning of a line of source code and must be followed by a colon, regardless of whether it is followed by a statement on the same line.

The compiler identifies labels by checking whether the beginning of the line matches any already-defined identifier. If it does not, the compiler assumes it is a label.

Labels have their own declaration space and do not interfere with other identifiers. A label's scope is the body of the method. Label declaration takes precedence in any ambiguous situation.

NOTE

Labels can be used only on executable statements inside methods.

To label a line of code

Place an identifier, followed by a colon, at the beginning of the line of source code.

For example, the following lines of code are labeled with `Jump` and `120`, respectively:

```
Jump:  FileOpen(1, "testFile", OpenMode.Input)
      ' ...
120:   FileClose(1)
```

See also

- [Statements](#)
- [Declared Element Names](#)
- [Program Structure and Code Conventions](#)

Special Characters in Code (Visual Basic)

8/22/2019 • 3 minutes to read • [Edit Online](#)

Sometimes you have to use special characters in your code, that is, characters that are not alphabetical or numeric. The punctuation and special characters in the Visual Basic character set have various uses, from organizing program text to defining the tasks that the compiler or the compiled program performs. They do not specify an operation to be performed.

Parentheses

Use parentheses when you define a procedure, such as a `Sub` or `Function`. You must enclose all procedure argument lists in parentheses. You also use parentheses for putting variables or arguments into logical groups, especially to override the default order of operator precedence in a complex expression. The following example illustrates this.

```
Dim a, b, c, d, e As Double
a = 3.2
b = 7.6
c = 2
d = b + c / a
e = (b + c) / a
```

Following execution of the previous code, the value of `d` is 8.225 and the value of `e` is 3. The calculation for `d` uses the default precedence of `/` over `+` and is equivalent to `d = b + (c / a)`. The parentheses in the calculation for `e` override the default precedence.

Separators

Separators do what their name suggests: they separate sections of code. In Visual Basic, the separator character is the colon (`:`). Use separators when you want to include multiple statements on a single line instead of separate lines. This saves space and improves the readability of your code. The following example shows three statements separated by colons.

```
a = 3.2 : b = 7.6 : c = 2
```

For more information, see [How to: Break and Combine Statements in Code](#).

The colon (`:`) character is also used to identify a statement label. For more information, see [How to: Label Statements](#).

Concatenation

Use the `&` operator for *concatenation*, or linking strings together. Do not confuse it with the `+` operator, which adds together numeric values. If you use the `+` operator to concatenate when you operate on numeric values, you can obtain incorrect results. The following example demonstrates this.

```
var1 = "10.01"
var2 = 11
resultA = var1 + var2
resultB = var1 & var2
```

Following execution of the previous code, the value of `resultA` is 21.01 and the value of `resultB` is "10.0111".

Member Access Operators

To access a member of a type, you use the dot (.) or exclamation point (!) operator between the type name and the member name.

Dot (.) Operator

Use the . operator on a class, structure, interface, or enumeration as a member access operator. The member can be a field, property, event, or method. The following example illustrates this.

```
Dim nextForm As New System.Windows.Forms.Form
' Access Text member (property) of Form class (on nextForm object).
nextForm.Text = "This is the next form"
' Access Close member (method) on nextForm.
nextForm.Close()
```

Exclamation Point (!) Operator

Use the ! operator only on a class or interface as a dictionary access operator. The class or interface must have a default property that accepts a single `String` argument. The identifier immediately following the ! operator becomes the argument value passed to the default property as a string. The following example demonstrates this.

```
Public Class hasDefault
    Default Public ReadOnly Property index(ByVal s As String) As Integer
        Get
            Return 32768 + AscW(s)
        End Get
    End Property
End Class
Public Class testHasDefault
    Public Sub compareAccess()
        Dim hD As hasDefault = New hasDefault()
        MsgBox("Traditional access returns " & hD.index("X") & vbCrLf &
               "Default property access returns " & hD("X") & vbCrLf &
               "Dictionary access returns " & hD!X)
    End Sub
End Class
```

The three output lines of `MsgBox` all display the value 32856. The first line uses the traditional access to property `index`, the second makes use of the fact that `index` is the default property of class `hasDefault`, and the third uses dictionary access to the class.

Note that the second operand of the ! operator must be a valid Visual Basic identifier not enclosed in double quotation marks (""). In other words, you cannot use a string literal or string variable. The following change to the last line of the `MsgBox` call generates an error because "x" is an enclosed string literal.

```
"Dictionary access returns " & hD!"X")
```

NOTE

References to default collections must be explicit. In particular, you cannot use the `!` operator on a late-bound variable.

The `!` character is also used as the `Single` type character.

See also

- [Program Structure and Code Conventions](#)
- [Type Characters](#)

Comments in Code (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

As you read the code examples, you often encounter the comment symbol (''). This symbol tells the Visual Basic compiler to ignore the text following it, or the *comment*. Comments are brief explanatory notes added to code for the benefit of those reading it.

It is good programming practice to begin all procedures with a brief comment describing the functional characteristics of the procedure (what it does). This is for your own benefit and the benefit of anyone else who examines the code. You should separate the implementation details (how the procedure does it) from comments that describe the functional characteristics. When you include implementation details in the description, remember to update them when you update the function.

Comments can follow a statement on the same line, or occupy an entire line. Both are illustrated in the following code.

```
' This is a comment beginning at the left edge of the screen.  
text1.Text = "Hi!"    ' This is an inline comment.
```

If your comment requires more than one line, use the comment symbol on each line, as the following example illustrates.

```
' This comment is too long to fit on a single line, so we break  
' it into two lines. Some comments might need three or more lines.
```

Commenting Guidelines

The following table provides general guidelines for what types of comments can precede a section of code. These are suggestions; Visual Basic does not enforce rules for adding comments. Write what works best, both for you and for anyone else who reads your code.

Comment type	Comment description
Purpose	Describes what the procedure does (not how it does it)
Assumptions	Lists each external variable, control, open file, or other element accessed by the procedure
Effects	Lists each affected external variable, control, or file, and the effect it has (only if it is not obvious)
Inputs	Specifies the purpose of the argument
Returns	Explains the values returned by the procedure

Remember the following points:

- Every important variable declaration should be preceded by a comment describing the use of the variable being declared.

- Variables, controls, and procedures should be named clearly enough that commenting is needed only for complex implementation details.
- Comments cannot follow a line-continuation sequence on the same line.

You can add or remove comment symbols for a block of code by selecting one or more lines of code and choosing the **Comment** () and **Uncomment** () buttons on the **Edit** toolbar.

NOTE

You can also add comments to your code by preceding the text with the `REM` keyword. However, the `'` symbol and the **Comment/Uncomment** buttons are easier to use and require less space and memory.

See also

- [Basic Instincts - Documenting Your Code With XML Comments](#)
- [How to: Create XML Documentation](#)
- [XML Comment Tags](#)
- [Program Structure and Code Conventions](#)
- [REM Statement](#)

Keywords as Element Names in Code (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

Any program element — such as a variable, class, or member — can have the same name as a restricted keyword. For example, you can create a variable named `Loop`. However, to refer to your version of it — which has the same name as the restricted `Loop` keyword — you must either precede it with a full qualification string or enclose it in square brackets (`[]`), as the following example shows.

```
' The following statement precedes Loop with a full qualification string.  
sampleFormLoop.Visible = True  
' The following statement encloses Loop in square brackets.  
[Loop].Visible = True
```

If you do not do either of these, then Visual Basic assumes use of the intrinsic `Loop` keyword and produces an error, as in the following example:

```
' The following statement causes a compiler error.  
  
Loop.Visible = True
```

You can use square brackets when referring to forms and controls, and when declaring a variable or defining a procedure with the same name as a restricted keyword. It can be easy to forget to qualify names or include square brackets, and thus introduce errors into your code and make it harder to read. For this reason, we recommend that you not use restricted keywords as the names of program elements. However, if a future version of Visual Basic defines a new keyword that conflicts with an existing form or control name, then you can use this technique when updating your code to work with the new version.

NOTE

Your program also might include element names provided by other referenced assemblies. If these names conflict with restricted keywords, then placing square brackets around them causes Visual Basic to interpret them as your defined elements.

See also

- [Visual Basic Naming Conventions](#)
- [Program Structure and Code Conventions](#)
- [Keywords](#)

Me, My, MyBase, and MyClass in Visual Basic

10/7/2019 • 2 minutes to read • [Edit Online](#)

`Me`, `My`, `MyBase`, and `MyClass` in Visual Basic have similar names, but different purposes. This topic describes each of these entities in order to distinguish them.

Me

The `Me` keyword provides a way to refer to the specific instance of a class or structure in which the code is currently executing. `Me` behaves like either an object variable or a structure variable referring to the current instance. Using `Me` is particularly useful for passing information about the currently executing instance of a class or structure to a procedure in another class, structure, or module.

For example, suppose you have the following procedure in a module.

```
Sub ChangeFormColor(FormName As Form)
    Randomize()
    FormName.BackColor = Color.FromArgb(Rnd() * 256, Rnd() * 256, Rnd() * 256)
End Sub
```

You can call this procedure and pass the current instance of the `Form` class as an argument by using the following statement.

```
ChangeFormColor(Me)
```

My

The `My` feature provides easy and intuitive access to a number of .NET Framework classes, enabling the Visual Basic user to interact with the computer, application, settings, resources, and so on.

MyBase

The `MyBase` keyword behaves like an object variable referring to the base class of the current instance of a class.

`MyBase` is commonly used to access base class members that are overridden or shadowed in a derived class.

`MyBase.New` is used to explicitly call a base class constructor from a derived class constructor.

MyClass

The `MyClass` keyword behaves like an object variable referring to the current instance of a class as originally implemented. `Myclass` is similar to `Me`, but all method calls on it are treated as if the method were `NotOverridable`.

See also

- [Inheritance Basics](#)

Visual Basic Limitations

4/28/2019 • 2 minutes to read • [Edit Online](#)

Earlier versions of Visual Basic enforced boundaries in code, such as the length of variable names, the number of variables allowed in modules, and module size. In Visual Basic .NET, these restrictions have been relaxed, giving you greater freedom in writing and arranging your code.

Physical limits are dependent more on run-time memory than on compile-time considerations. If you use prudent programming practices, and divide large applications into multiple classes and modules, then there is very little chance of encountering an internal Visual Basic limitation.

The following are some limitations that you might encounter in extreme cases:

- **Name Length.** There is a maximum number of characters for the name of every declared programming element. This maximum applies to an entire qualification string if the element name is qualified. See [Declared Element Names](#).
- **Line Length.** There is a maximum of 65535 characters in a physical line of source code. The logical source code line can be longer if you use line continuation characters. See [How to: Break and Combine Statements in Code](#).
- **Array Dimensions.** There is a maximum number of dimensions you can declare for an array. This limits how many indexes you can use to specify an array element. See [Array Dimensions in Visual Basic](#).
- **String Length.** There is a maximum number of Unicode characters you can store in a single string. See [String Data Type](#).
- **Environment String Length.** There is a maximum of 32768 characters for any environment string used as a command-line argument. This is a limitation on all platforms.

See also

- [Program Structure and Code Conventions](#)
- [Visual Basic Naming Conventions](#)

Visual Basic Language Features

9/7/2018 • 2 minutes to read • [Edit Online](#)

The following topics introduce and discuss the essential components of Visual Basic, an object-oriented programming language. After creating the user interface for your application using forms and controls, you need to write the code that defines the application's behavior. As with any modern programming language, Visual Basic supports a number of common programming constructs and language elements.

If you have programmed in other languages, much of the material covered in this section might seem familiar. While most of the constructs are similar to those in other languages, the event-driven nature of Visual Basic introduces some subtle differences.

If you are new to programming, the material in this section serves as an introduction to the basic building blocks for writing code. Once you understand the basics, you can create powerful applications using Visual Basic.

In This Section

[Arrays](#)

Discusses making your code more compact and powerful by declaring and using arrays, which hold multiple related values.

[Collection Initializers](#)

Describes collection initializers, which enable you to create a collection and populate it with an initial set of values.

[Constants and Enumerations](#)

Discusses storing unchanging values for repeated use, including sets of related constant values.

[Control Flow](#)

Shows how to regulate the flow of your program's execution.

[Data Types](#)

Describes what kinds of data a programming element can hold and how that data is stored.

[Declared Elements](#)

Covers programming elements you can declare, their names and characteristics, and how the compiler resolves references to them.

[Delegates](#)

Provides an introduction to delegates and how they are used in Visual Basic.

[Early and Late Binding](#)

Describes binding, which is performed by the compiler when an object is assigned to an object variable, and the differences between early-bound and late-bound objects.

[Error Types](#)

Provides an overview of syntax errors, run-time errors, and logic errors.

[Events](#)

Shows how to declare and use events.

[Interfaces](#)

Describes what interfaces are and how you can use them in your applications.

[LINQ](#)

Provides links to topics that introduce Language-Integrated Query (LINQ) features and programming.

[Objects and Classes](#)

Provides an overview of objects and classes, how they are used, their relationships to each other, and the properties, methods, and events they expose.

[Operators and Expressions](#)

Describes the code elements that manipulate value-holding elements, how to use them efficiently, and how to combine them to yield new values.

[Procedures](#)

Describes `Sub`, `Function`, `Property`, and `Operator` procedures, as well as advanced topics such as recursive and overloaded procedures.

[Statements](#)

Describes declaration and executable statements.

[Strings](#)

Provides links to topics that describe the basic concepts about using strings in Visual Basic.

[Variables](#)

Introduces variables and describes how to use them in Visual Basic.

[XML](#)

Provides links to topics that describe how to use XML in Visual Basic.

Related Sections

[Collections](#)

Describes some of the types of collections that are provided by the .NET Framework. Demonstrates how to use simple collections and collections of key/value pairs.

[Visual Basic Language Reference](#)

Provides reference information on various aspects of Visual Basic programming.

Arrays in Visual Basic

6/7/2019 • 28 minutes to read • [Edit Online](#)

An array is a set of values, which are termed *elements*, that are logically related to each other. For example, an array may consist of the number of students in each grade in a grammar school; each element of the array is the number of students in a single grade. Similarly, an array may consist of a student's grades for a class; each element of the array is a single grade.

It is possible individual variables to store each of our data items. For example, if our application analyzes student grades, we can use a separate variable for each student's grade, such as `englishGrade1`, `englishGrade2`, etc. This approach has three major limitations:

- We have to know at design time exactly how many grades we have to handle.
- Handling large numbers of grades quickly becomes unwieldy. This in turn makes an application much more likely to have serious bugs.
- It is difficult to maintain. Each new grade that we add requires that the application be modified, recompiled, and redeployed.

By using an array, you can refer to these related values by the same name, and use a number that's called an *index* or *subscript* to identify an individual element based on its position in the array. The indexes of an array range from 0 to one less than the total number of elements in the array. When you use Visual Basic syntax to define the size of an array, you specify its highest index, not the total number of elements in the array. You can work with the array as a unit, and the ability to iterate its elements frees you from needing to know exactly how many elements it contains at design time.

Some quick examples before explanation:

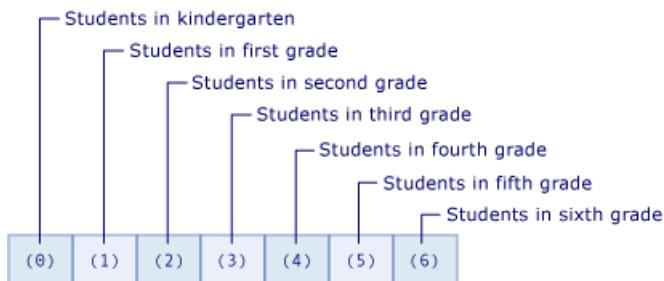
```
' Declare a single-dimension array of 5 numbers.  
Dim numbers(4) As Integer  
  
' Declare a single-dimension array and set its 4 values.  
Dim numbers = New Integer() {1, 2, 4, 8}  
  
' Change the size of an existing array to 16 elements and retain the current values.  
ReDim Preserve numbers(15)  
  
' Redefine the size of an existing array and reset the values.  
ReDim numbers(15)  
  
' Declare a 6 x 6 multidimensional array.  
Dim matrix(5, 5) As Double  
  
' Declare a 4 x 3 multidimensional array and set array element values.  
Dim matrix = New Integer(3, 2) {{1, 2, 3}, {2, 3, 4}, {3, 4, 5}, {4, 5, 6}}  
  
' Declare a jagged array  
Dim sales()() As Double = New Double(11)() {}
```

Array elements in a simple array

Let's create an array named `students` to store the number of students in each grade in a grammar school. The indexes of the elements range from 0 through 6. Using this array is simpler than declaring seven variables.

The following illustration shows the `students` array. For each element of the array:

- The index of the element represents the grade (index 0 represents kindergarten).
- The value that's contained in the element represents the number of students in that grade.



The following example contains the Visual Basic code that creates and uses the array:

```
Module SimpleArray
    Public Sub Main()
        ' Declare an array with 7 elements.
        Dim students(6) As Integer

        ' Assign values to each element.
        students(0) = 23
        students(1) = 19
        students(2) = 21
        students(3) = 17
        students(4) = 19
        students(5) = 20
        students(6) = 22

        ' Display the value of each element.
        For ctr As Integer = 0 To 6
            Dim grade As String = If(ctr = 0, "kindergarten", $"grade {ctr}")
            Console.WriteLine($"Students in {grade}: {students(ctr)}")
        Next
    End Sub
End Module
```

' The example displays the following output:
' Students in kindergarten: 23
' Students in grade 1: 19
' Students in grade 2: 21
' Students in grade 3: 17
' Students in grade 4: 19
' Students in grade 5: 20
' Students in grade 6: 22

The example does three things:

- It declares a `students` array with seven elements. The number `6` in the array declaration indicates the last index in the array; it is one less than the number of elements in the array.
- It assigns values to each element in the array. Array elements are accessed by using the array name and including the index of the individual element in parentheses.
- It lists each value of the array. The example uses a `For` statement to access each element of the array by its index number.

The `students` array in the preceding example is a one-dimensional array because it uses one index. An array that uses more than one index or subscript is called *multidimensional*. For more information, see the rest of this article and [Array Dimensions in Visual Basic](#).

Creating an array

You can define the size of an array in several ways:

- You can specify the size when the array is declared:

```
' Declare an array with 10 elements.  
Dim cargoWeights(9) As Double  
' Declare a 24 x 2 array.  
Dim hourlyTemperatures(23, 1) As Integer  
' Declare a jagged array with 31 elements.  
Dim januaryInquiries(30)() As String
```

- You can use a `New` clause to supply the size of an array when it's created:

```
' Declare an array with 10 elements.  
Dim cargoWeights(9) As Double  
' Declare a 24 x 2 array.  
Dim hourlyTemperatures(23, 1) As Integer  
' Declare a jagged array with 31 elements.  
Dim januaryInquiries(30)() As String
```

If you have an existing array, you can redefine its size by using the `ReDim` statement. You can specify that the `ReDim` statement keep the values that are in the array, or you can specify that it create an empty array. The following example shows different uses of the `ReDim` statement to modify the size of an existing array.

```
' Assign a new array size and retain the current values.  
ReDim Preserve cargoWeights(20)  
' Assign a new array size and retain only the first five values.  
ReDim Preserve cargoWeights(4)  
' Assign a new array size and discard all current element values.  
ReDim cargoWeights(15)
```

For more information, see the [ReDim Statement](#).

Storing values in an array

You can access each location in an array by using an index of type `Integer`. You can store and retrieve values in an array by referencing each array location by using its index enclosed in parentheses. Indexes for multidimensional arrays are separated by commas (,). You need one index for each array dimension.

The following example shows some statements that store and retrieve values in arrays.

```

Module Example
    Public Sub Main()
        ' Create a 10-element integer array.
        Dim numbers(9) As Integer
        Dim value As Integer = 2

        ' Write values to it.
        For ctr As Integer = 0 To 9
            numbers(ctr) = value
            value *= 2
        Next

        ' Read and sum the array values.
        Dim sum As Integer
        For ctr As Integer = 0 To 9
            sum += numbers(ctr)
        Next
        Console.WriteLine($"The sum of the values is {sum:N0}")
    End Sub
End Module
' The example displays the following output:
'   The sum of the values is 2,046

```

Populating an array with array literals

By using an array literal, you can populate an array with an initial set of values at the same time that you create it. An array literal consists of a list of comma-separated values that are enclosed in braces ({}).

When you create an array by using an array literal, you can either supply the array type or use type inference to determine the array type. The following example shows both options.

```

' Array literals with explicit type definition.
Dim numbers = New Integer() {1, 2, 4, 8}
' Array literals with type inference.
Dim doubles = {1.5, 2, 9.9, 18}
' Array literals with explicit type definition.
Dim articles() As String = { "the", "a", "an" }

' Array literals with explicit widening type definition.
Dim values() As Double = { 1, 2, 3, 4, 5 }

```

When you use type inference, the type of the array is determined by the *dominant type* in the list of literal values. The dominant type is the type to which all other types in the array can widen. If this unique type can't be determined, the dominant type is the unique type to which all other types in the array can narrow. If neither of these unique types can be determined, the dominant type is `Object`. For example, if the list of values that's supplied to the array literal contains values of type `Integer`, `Long`, and `Double`, the resulting array is of type `Double`. Because `Integer` and `Long` widen only to `Double`, `Double` is the dominant type. For more information, see [Widening and Narrowing Conversions](#).

NOTE

You can use type inference only for arrays that are defined as local variables in a type member. If an explicit type definition is absent, arrays defined with array literals at the class level are of type `Object[]`. For more information, see [Local type inference](#).

Note that the previous example defines `values` as an array of type `Double` even though all the array literals are of type `Integer`. You can create this array because the values in the array literal can widen to `Double` values.

You can also create and populate a multidimensional array by using *nested array literals*. Nested array literals must have a number of dimensions that's consistent with the resulting array. The following example creates a two-dimensional array of integers by using nested array literals.

```
' Create and populate a 2 x 2 array.  
Dim grid1 = {{1, 2}, {3, 4}}  
' Create and populate a 2 x 2 array with 3 elements.  
Dim grid2(,) = {{1, 2}, {3, 4}, {5, 6}}
```

When using nested array literals to create and populate an array, an error occurs if the number of elements in the nested array literals don't match. An error also occurs if you explicitly declare the array variable to have a different number of dimensions than the array literals.

Just as you can for one-dimensional arrays, you can rely on type inference when creating a multidimensional array with nested array literals. The inferred type is the dominant type for all the values in all the array literals for all nesting level. The following example creates a two-dimensional array of type `Double[,]` from values that are of type `Integer` and `Double`.

```
Dim arr = {{1, 2.0}, {3, 4}, {5, 6}, {7, 8}}
```

For additional examples, see [How to: Initialize an Array Variable in Visual Basic](#).

Iterating through an array

When you iterate through an array, you access each element in the array from the lowest index to the highest or from the highest to the lowest. Typically, use either the [For...Next Statement](#) or the [For Each...Next Statement](#) to iterate through the elements of an array. When you don't know the upper bounds of the array, you can call the [Array.GetUpperBound](#) method to get the highest value of the index. Although lowest index value is almost always 0, you can call the [Array.GetLowerBound](#) method to get the lowest value of the index.

The following example iterates through a one-dimensional array by using the [For...Next](#) statement.

```
Module IterateArray  
    Public Sub Main()  
        Dim numbers = {10, 20, 30}  
  
        For index = 0 To numbers.GetUpperBound(0)  
            Console.WriteLine(numbers(index))  
        Next  
    End Sub  
End Module  
' The example displays the following output:  
' 10  
' 20  
' 30
```

The following example iterates through a multidimensional array by using a [For...Next](#) statement. The [GetUpperBound](#) method has a parameter that specifies the dimension. `GetUpperBound(0)` returns the highest index of the first dimension, and `GetUpperBound(1)` returns the highest index of the second dimension.

```

Module IterateArray
    Public Sub Main()
        Dim numbers = {{1, 2}, {3, 4}, {5, 6}}

        For index0 = 0 To numbers.GetUpperBound(0)
            For index1 = 0 To numbers.GetUpperBound(1)
                Console.WriteLine($"{numbers(index0, index1)} ")
            Next
            Console.WriteLine()
        Next
    End Sub
End Module
' The example displays the following output:
' Output
'   1 2
'   3 4
'   5 6

```

The following example uses a [For Each...Next Statement](#) to iterate through a one-dimensional array and a two-dimensional array.

```

Module IterateWithForEach
    Public Sub Main()
        ' Declare and iterate through a one-dimensional array.
        Dim numbers1 = {10, 20, 30}

        For Each number In numbers1
            Console.WriteLine(number)
        Next
        Console.WriteLine()

        Dim numbers = {{1, 2}, {3, 4}, {5, 6}}


        For Each number In numbers
            Console.WriteLine(number)
        Next
    End Sub
End Module
' The example displays the following output:
'   10
'   20
'   30
'
'   1
'   2
'   3
'   4
'   5
'   6

```

Array size

The size of an array is the product of the lengths of all its dimensions. It represents the total number of elements currently contained in the array. For example, the following example declares a 2-dimensional array with four elements in each dimension. As the output from the example shows, the array's size is 16 (or $(3 + 1) * (3 + 1)$).

```

Module Example
    Public Sub Main()
        Dim arr(3, 3) As Integer
        Console.WriteLine(arr.Length)
    End Sub
End Module
' The example displays the following output:
'     16

```

NOTE

This discussion of array size does not apply to jagged arrays. For information on jagged arrays and determining the size of a jagged array, see the [Jagged arrays](#) section.

You can find the size of an array by using the [Array.Length](#) property. You can find the length of each dimension of a multidimensional array by using the [Array.GetLength](#) method.

You can resize an array variable by assigning a new array object to it or by using the [ReDim Statement](#) statement. The following example uses the [ReDim](#) statement to change a 100-element array to a 51-element array.

```

Module Example
    Public Sub Main()
        Dim arr(99) As Integer
        Console.WriteLine(arr.Length)

        Redim arr(50)
        Console.WriteLine(arr.Length)
    End Sub
End Module
' The example displays the following output:
'     100
'     51

```

There are several things to keep in mind when dealing with the size of an array.

Dimension Length	The index of each dimension is 0-based, which means it ranges from 0 to its upper bound. Therefore, the length of a given dimension is one greater than the declared upper bound of that dimension.
Length Limits	The length of every dimension of an array is limited to the maximum value of the Integer data type, which is Int32.MaxValue or $(2 ^ 31) - 1$. However, the total size of an array is also limited by the memory available on your system. If you attempt to initialize an array that exceeds the amount of available memory, the runtime throws an OutOfMemoryException .

Size and Element Size	An array's size is independent of the data type of its elements. The size always represents the total number of elements, not the number of bytes that they consume in memory.
Memory Consumption	It is not safe to make any assumptions regarding how an array is stored in memory. Storage varies on platforms of different data widths, so the same array can consume more memory on a 64-bit system than on a 32-bit system. Depending on system configuration when you initialize an array, the common language runtime (CLR) can assign storage either to pack elements as close together as possible, or to align them all on natural hardware boundaries. Also, an array requires a storage overhead for its control information, and this overhead increases with each added dimension.

The array type

Every array has a data type, which differs from the data type of its elements. There is no single data type for all arrays. Instead, the data type of an array is determined by the number of dimensions, or *rank*, of the array, and the data type of the elements in the array. Two array variables are of the same data type only when they have the same rank and their elements have the same data type. The lengths of the dimensions of an array do not influence the array data type.

Every array inherits from the [System.Array](#) class, and you can declare a variable to be of type `Array`, but you cannot create an array of type `Array`. For example, although the following code declares the `arr` variable to be of type `Array` and calls the [Array.CreateInstance](#) method to instantiate the array, the array's type proves to be `Object[]`.

```
Module Example
    Public Sub Main()
        Dim arr As Array = Array.CreateInstance(GetType(Object), 19)
        Console.WriteLine(arr.Length)
        Console.WriteLine(arr.GetType().Name)
    End Sub
End Module
' The example displays the following output:
'   19
'   Object[]
```

Also, the [ReDim Statement](#) cannot operate on a variable declared as type `Array`. For these reasons, and for type safety, it is advisable to declare every array as a specific type.

You can find out the data type of either an array or its elements in several ways.

- You can call the [GetType](#) method on the variable to get a [Type](#) object that represents the run-time type of the variable. The [Type](#) object holds extensive information in its properties and methods.
- You can pass the variable to the [TypeName](#) function to get a `String` with the name of run-time type.

The following example calls the both the `GetType` method and the `TypeName` function to determine the type of an array. The array type is `Byte(,)`. Note that the [Type BaseType](#) property also indicates that the base type of the byte array is the [Array](#) class.

```

Module Example
    Public Sub Main()
        Dim bytes(9,9) As Byte
        Console.WriteLine($"Type of {nameof(bytes)} array: {bytes.GetType().Name}")
        Console.WriteLine($"Base class of {nameof(bytes)}: {bytes.GetType().BaseType.Name}")
        Console.WriteLine()
        Console.WriteLine($"Type of {nameof(bytes)} array: {TypeName(bytes)}")
    End Sub
End Module
' The example displays the following output:
' Type of bytes array: Byte[,]
' Base class of bytes: Array
'
' Type of bytes array: Byte(,)


```

Arrays as return values and parameters

To return an array from a `Function` procedure, specify the array data type and the number of dimensions as the return type of the [Function Statement](#). Within the function, declare a local array variable with same data type and number of dimensions. In the [Return Statement](#), include the local array variable without parentheses.

To specify an array as a parameter to a `Sub` or `Function` procedure, define the parameter as an array with a specified data type and number of dimensions. In the call to the procedure, pass an array variable with the same data type and number of dimensions.

In the following example, the `GetNumbers` function returns an `Integer()`, a one-dimensional array of type `Integer`. The `ShowNumbers` procedure accepts an `Integer()` argument.

```

Module ReturnValuesAndParams
    Public Sub Main()
        Dim numbers As Integer() = GetNumbers()
        ShowNumbers(numbers)
    End Sub

    Private Function GetNumbers() As Integer()
        Dim numbers As Integer() = {10, 20, 30}
        Return numbers
    End Function

    Private Sub ShowNumbers(numbers As Integer())
        For index = 0 To numbers.GetUpperBound(0)
            Console.WriteLine($"{numbers(index)} ")
        Next
    End Sub
End Module
' The example displays the following output:
' 10
' 20
' 30


```

In the following example, the `GetNumbersMultiDim` function returns an `Integer(,)`, a two-dimensional array of type `Integer`. The `ShowNumbersMultiDim` procedure accepts an `Integer(,)` argument.

```

Module Example
    Public Sub Main()
        Dim numbers As Integer(,) = GetNumbersMultidim()
        ShowNumbersMultidim(numbers)
    End Sub

    Private Function GetNumbersMultidim() As Integer(,)
        Dim numbers As Integer(,) = {{1, 2}, {3, 4}, {5, 6}}
        Return numbers
    End Function

    Private Sub ShowNumbersMultidim(numbers As Integer(,))
        For index0 = 0 To numbers.GetUpperBound(0)
            For index1 = 0 To numbers.GetUpperBound(1)
                Console.WriteLine($"{numbers(index0, index1)} ")
            Next
            Console.WriteLine()
        Next
    End Sub
End Module
' The example displays the following output:
'      1 2
'      3 4
'      5 6

```

Jagged arrays

Sometimes the data structure in your application is two-dimensional but not rectangular. For example, you might use an array to store data about the high temperature of each day of the month. The first dimension of the array represents the month, but the second dimension represents the number of days, and the number of days in a month is not uniform. A *jagged array*, which is also called an *array of arrays*, is designed for such scenarios. A jagged array is an array whose elements are also arrays. A jagged array and each element in a jagged array can have one or more dimensions.

The following example uses an array of months, each element of which is an array of days. The example uses a jagged array because different months have different numbers of days. The example shows how to create a jagged array, assign values to it, and retrieve and display its values.

```

Imports System.Globalization

Module JaggedArray
    Public Sub Main()
        ' Declare the jagged array of 12 elements. Each element is an array of Double.
        Dim sales(11)() As Double
        ' Set each element of the sales array to a Double array of the appropriate size.
        For month As Integer = 0 To 11
            ' The number of days in the month determines the appropriate size.
            Dim daysInMonth As Integer =
                DateTime.DaysInMonth(Year(Now), month + 1)
            sales(month) = New Double(daysInMonth - 1) {}
        Next

        ' Store values in each element.
        For month As Integer = 0 To 11
            For dayOfMonth = 0 To sales(month).GetUpperBound(0)
                sales(month)(dayOfMonth) = (month * 100) + dayOfMonth
            Next
        Next

        ' Retrieve and display the array values.
        Dim monthNames = DateTimeFormatInfo.CurrentInfo.AbbreviatedMonthNames

```

```

' Display the month names.
Console.WriteLine()
For ctr = 0 To sales.GetUpperBound(0)
    Console.WriteLine($"{monthNames(ctr)}")
Next
Console.WriteLine()
' Display data for each day in each month.
For dayInMonth = 0 To 30
    Console.WriteLine($"{dayInMonth + 1,2}. ")
    For monthNumber = 0 To sales.GetUpperBound(0)
        If dayInMonth > sales(monthNumber).GetUpperBound(0) Then
            Console.WriteLine("      ")
        Else
            Console.WriteLine($"{sales(monthNumber)(dayInMonth),-5} ")
        End If
    Next
    Console.WriteLine()
Next
End Sub
End Module
' The example displays the following output:
'   Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov   Dec
' 1.  0    100   200   300   400   500   600   700   800   900   1000  1100
' 2.  1    101   201   301   401   501   601   701   801   901   1001  1101
' 3.  2    102   202   302   402   502   602   702   802   902   1002  1102
' 4.  3    103   203   303   403   503   603   703   803   903   1003  1103
' 5.  4    104   204   304   404   504   604   704   804   904   1004  1104
' 6.  5    105   205   305   405   505   605   705   805   905   1005  1105
' 7.  6    106   206   306   406   506   606   706   806   906   1006  1106
' 8.  7    107   207   307   407   507   607   707   807   907   1007  1107
' 9.  8    108   208   308   408   508   608   708   808   908   1008  1108
' 10. 9   109   209   309   409   509   609   709   809   909   1009  1109
' 11. 10  110   210   310   410   510   610   710   810   910   1010  1110
' 12. 11  111   211   311   411   511   611   711   811   911   1011  1111
' 13. 12  112   212   312   412   512   612   712   812   912   1012  1112
' 14. 13  113   213   313   413   513   613   713   813   913   1013  1113
' 15. 14  114   214   314   414   514   614   714   814   914   1014  1114
' 16. 15  115   215   315   415   515   615   715   815   915   1015  1115
' 17. 16  116   216   316   416   516   616   716   816   916   1016  1116
' 18. 17  117   217   317   417   517   617   717   817   917   1017  1117
' 19. 18  118   218   318   418   518   618   718   818   918   1018  1118
' 20. 19  119   219   319   419   519   619   719   819   919   1019  1119
' 21. 20  120   220   320   420   520   620   720   820   920   1020  1120
' 22. 21  121   221   321   421   521   621   721   821   921   1021  1121
' 23. 22  122   222   322   422   522   622   722   822   922   1022  1122
' 24. 23  123   223   323   423   523   623   723   823   923   1023  1123
' 25. 24  124   224   324   424   524   624   724   824   924   1024  1124
' 26. 25  125   225   325   425   525   625   725   825   925   1025  1125
' 27. 26  126   226   326   426   526   626   726   826   926   1026  1126
' 28. 27  127   227   327   427   527   627   727   827   927   1027  1127
' 29. 28          328   428   528   628   728   828   928   1028  1128
' 30. 29          229   329   429   529   629   729   829   929   1029  1129
' 31. 30          230   430   630   730   830   930   1030  1130

```

The previous example assigns values to the jagged array on an element-by-element basis by using a `For...Next` loop. You can also assign values to the elements of a jagged array by using nested array literals. However, the attempt to use nested array literals (for example, `Dim valuesJagged = {{1, 2}, {2, 3, 4}}`) generates compiler error [BC30568](#). To correct the error, enclose the inner array literals in parentheses. The parentheses force the array literal expression to be evaluated, and the resulting values are used with the outer array literal, as the following example shows.

```

Module Example
    Public Sub Main()
        Dim values1d = { 1, 2, 3 }
        Dim values2d = {{1, 2}, {2, 3}, {3, 4}}
        Dim valuesjagged = {{(1, 2)}, {(2, 3, 4)}}
    End Sub
End Module

```

A jagged array is a one-dimensional array whose elements contain arrays. Therefore, the [Array.Length](#) property and the [Array.GetLength\(0\)](#) method return the number of elements in the one-dimensional array, and [Array.GetLength\(1\)](#) throws an [IndexOutOfRangeException](#) because a jagged array is not multidimensional. You determine the number of elements in each subarray by retrieving the value of each subarray's [Array.Length](#) property. The following example illustrates how to determine the number of elements in a jagged array.

```

Module Example
    Public Sub Main()
        Dim jagged = { {(1, 2)}, {(2, 3, 4)}, {(5, 6)}, {(7, 8, 9, 10)} }
        Console.WriteLine($"The value of jagged.Length: {jagged.Length}.")
        Dim total = jagged.Length
        For ctr As Integer = 0 To jagged.GetUpperBound(0)
            Console.WriteLine($"Element {ctr + 1} has {jagged(ctr).Length} elements.")
            total += jagged(ctr).Length
        Next
        Console.WriteLine($"The total number of elements in the jagged array: {total}")
    End Sub
End Module
' The example displays the following output:
'   The value of jagged.Length: 4.
'   Element 1 has 3 elements.
'   Element 2 has 4 elements.
'   Element 3 has 2 elements.
'   Element 4 has 4 elements.
'   The total number of elements in the jagged array: 15

```

Zero-length arrays

Visual Basic differentiates between a uninitialized array (an array whose value is [Nothing](#)) and a *zero-length array* or empty array (an array that has no elements.) An uninitialized array is one that has not been dimensioned or had any values assigned to it. For example:

```
Dim arr() As String
```

A zero-length array is declared with a dimension of -1. For example:

```
Dim arrZ(-1) As String
```

You might need to create a zero-length array under the following circumstances:

- Without risking a [NullReferenceException](#) exception, your code must access members of the [Array](#) class, such as [Length](#) or [Rank](#), or call a Visual Basic function such as [UBound](#).
- You want to keep your code simple by not having to check for [Nothing](#) as a special case.

- Your code interacts with an application programming interface (API) that either requires you to pass a zero-length array to one or more procedures or returns a zero-length array from one or more procedures.

Splitting an array

In some cases, you may need to split a single array into multiple arrays. This involves identifying the point or points at which the array is to be split, and then splitting the array into two or more separate arrays.

NOTE

This section does not discuss splitting a single string into a string array based on some delimiter. For information on splitting a string, see the [String.Split](#) method.

The most common criteria for splitting an array are:

- The number of elements in the array. For example, you might want to split an array of more than a specified number of elements into a number of approximately equal parts. For this purpose, you can use the value returned by either the [Array.Length](#) or [Array.GetLength](#) method.
- The value of an element, which serves as a delimiter that indicates where the array should be split. You can search for a specific value by calling the [Array.FindIndex](#) and [Array.FindLastIndex](#) methods.

Once you've determined the index or indexes at which the array should be split, you can then create the individual arrays by calling the [Array.Copy](#) method.

The following example splits an array into two arrays of approximately equal size. (If the total number of array elements is odd, the first array has one more element than the second.)

```
Module Example
    Public Sub Main()
        ' Create an array of 100 elements.
        Dim arr(99) As Integer
        ' Populate the array.
        Dim rnd As New Random()
        For ctr = 0 To arr.GetUpperBound(0)
            arr(ctr) = rnd.Next()
        Next

        ' Determine how many elements should be in each array.
        Dim divisor = 2
        Dim remainder As Integer
        Dim boundary = Math.DivRem(arr.GetLength(0), divisor, remainder)

        ' Copy the array.
        Dim arr1(boundary - 1 + remainder), arr2(boundary - 1) As Integer
        Array.Copy(arr, 0, arr1, 0, boundary + remainder)
        Array.Copy(arr, boundary + remainder, arr2, 0, arr.Length - boundary)
    End Sub
End Module
```

The following example splits a string array into two arrays based on the presence of an element whose value is "zzz", which serves as the array delimiter. The new arrays do not include the element that contains the delimiter.

```

Module Example
    Public Sub Main()
        Dim rnd As New Random()

        ' Create an array of 100 elements.
        Dim arr(99) As String
        ' Populate each element with an arbitrary ASCII character.
        For ctr = 0 To arr.GetUpperBound(0)
            arr(ctr) = ChrW(Rnd.Next(&h21, &h7F))
        Next
        ' Get a random number that will represent the point to insert the delimiter.
        arr(rnd.Next(0, arr.GetUpperBound(0))) = "zzz"

        ' Find the delimiter.
        Dim location = Array.FindIndex(arr, Function(x) x = "zzz")

        ' Create the arrays.
        Dim arr1(location - 1) As String
        Dim arr2(arr.GetUpperBound(0) - location - 1) As String

        ' Populate the two arrays.
        Array.Copy(arr, 0, arr1, 0, location)
        Array.Copy(arr, location + 1, arr2, 0, arr.GetUpperBound(0) - location)
    End Sub
End Module

```

Joining arrays

You can also combine a number of arrays into a single larger array. To do this, you also use the [Array.Copy](#) method.

NOTE

This section does not discuss joining a string array into a single string. For information on joining a string array, see the [String.Join](#) method.

Before copying the elements of each array into the new array, you must first ensure that you have initialized the array so that it is large enough to accommodate the new array. You can do this in one of two ways:

- Use the [ReDim Preserve](#) statement to dynamically expand the array before adding new elements to it. This is the easiest technique, but it can result in performance degradation and excessive memory consumption when you are copying large arrays.
- Calculate the total number of elements needed for the new large array, then add the elements of each source array to it.

The following example uses the second approach to add four arrays with ten elements each to a single array.

```

Imports System.Collections.Generic
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim tasks As New List(Of Task(Of Integer()))
        ' Generate four arrays.
        For ctr = 0 To 3
            Dim value = ctr
            tasks.Add(Task.Run(Function()
                Dim arr(9) As Integer
                For ndx = 0 To arr.GetUpperBound(0)
                    arr(ndx) = value
                Next
                Return arr
            End Function))
        Next
        Task.WaitAll(tasks.ToArray())
        ' Compute the number of elements in all arrays.
        Dim elements = 0
        For Each task In tasks
            elements += task.Result.Length
        Next
        Dim newArray(elements - 1) As Integer
        Dim index = 0
        For Each task In tasks
            Dim n = task.Result.Length
            Array.Copy(task.Result, 0, newArray, index, n)
            index += n
        Next
        Console.WriteLine($"The new array has {newArray.Length} elements.")
    End Sub
End Module
' The example displays the following output:
'     The new array has 40 elements.

```

Since in this case the source arrays are all small, we can also dynamically expand the array as we add the elements of each new array to it. The following example does that.

```

Imports System.Collections.Generic
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim tasks As New List(Of Task(Of Integer()))
        ' Generate four arrays.
        For ctr = 0 To 3
            Dim value = ctr
            tasks.Add(Task.Run(Function()
                Dim arr(9) As Integer
                For ndx = 0 To arr.GetUpperBound(0)
                    arr(ndx) = value
                Next
                Return arr
            End Function))
        Next
        Task.WaitAll(tasks.ToArray())
        ' Dimension the target array and copy each element of each source array to it.
        Dim newArray() As Integer = {}
        ' Define the next position to copy to in newArray.
        Dim index = 0
        For Each task In tasks
            Dim n = Task.Result.Length
            ReDim Preserve newArray(newArray.GetUpperBound(0) + n)
            Array.Copy(task.Result, 0, newArray, index, n)
            index += n
        Next
        Console.WriteLine($"The new array has {newArray.Length} elements.")
    End Sub
End Module
' The example displays the following output:
'   The new array has 40 elements.

```

Collections as an alternative to arrays

Arrays are most useful for creating and working with a fixed number of strongly typed objects. Collections provide a more flexible way to work with groups of objects. Unlike arrays, which require that you explicitly change the size of an array with the [ReDim Statement](#), collections grow and shrink dynamically as the needs of an application change.

When you use [ReDim](#) to redimension an array, Visual Basic creates a new array and releases the previous one. This takes execution time. Therefore, if the number of items you are working with changes frequently, or you cannot predict the maximum number of items you need, you'll usually obtain better performance by using a collection.

For some collections, you can assign a key to any object that you put into the collection so that you can quickly retrieve the object by using the key.

If your collection contains elements of only one data type, you can use one of the classes in the [System.Collections.Generic](#) namespace. A generic collection enforces type safety so that no other data type can be added to it.

For more information about collections, see [Collections](#).

Related topics

TERM	DEFINITION
Array Dimensions in Visual Basic	Explains rank and dimensions in arrays.
How to: Initialize an Array Variable in Visual Basic	Describes how to populate arrays with initial values.
How to: Sort An Array in Visual Basic	Shows how to sort the elements of an array alphabetically.
How to: Assign One Array to Another Array	Describes the rules and steps for assigning an array to another array variable.
Troubleshooting Arrays	Discusses some common problems that arise when working with arrays.

See also

- [System.Array](#)
- [Dim Statement](#)
- [ReDim Statement](#)

Collection Initializers (Visual Basic)

3/8/2019 • 5 minutes to read • [Edit Online](#)

Collection initializers provide a shortened syntax that enables you to create a collection and populate it with an initial set of values. Collection initializers are useful when you are creating a collection from a set of known values, for example, a list of menu options or categories, an initial set of numeric values, a static list of strings such as day or month names, or geographic locations such as a list of states that is used for validation.

For more information about collections, see [Collections](#).

You identify a collection initializer by using the `From` keyword followed by braces (`{}`). This is similar to the array literal syntax that is described in [Arrays](#). The following examples show various ways to use collection initializers to create collections.

```
' Create an array of type String().
Dim winterMonths = {"December", "January", "February"}

' Create an array of type Integer()
Dim numbers = {1, 2, 3, 4, 5}

' Create a list of menu options. (Requires an extension method
' named Add for List(Of MenuOption)
Dim menuOptions = New List(Of MenuOption) From {{1, "Home"}, 
                                                {2, "Products"}, 
                                                {3, "News"}, 
                                                {4, "Contact Us"}}
```

NOTE

C# also provides collection initializers. C# collection initializers provide the same functionality as Visual Basic collection initializers. For more information about C# collection initializers, see [Object and Collection Initializers](#).

Syntax

A collection initializer consists of a list of comma-separated values that are enclosed in braces (`{}`), preceded by the `From` keyword, as shown in the following code.

```
Dim names As New List(Of String) From {"Christa", "Brian", "Tim"}
```

When you create a collection, such as a `List<T>` or a `Dictionary<TKey,TValue>`, you must supply the collection type before the collection initializer, as shown in the following code.

```
Public Class AppMenu
    Public Property Items As List(Of String) =
        New List(Of String) From {"Home", "About", "Contact"}
End Class
```

NOTE

You cannot combine both a collection initializer and an object initializer to initialize the same collection object. You can use object initializers to initialize objects in a collection initializer.

Creating a Collection by Using a Collection Initializer

When you create a collection by using a collection initializer, each value that is supplied in the collection initializer is passed to the appropriate `Add` method of the collection. For example, if you create a `List<T>` by using a collection initializer, each string value in the collection initializer is passed to the `Add` method. If you want to create a collection by using a collection initializer, the specified type must be valid collection type. Examples of valid collection types include classes that implement the `IEnumerable<T>` interface or inherit the `CollectionBase` class. The specified type must also expose an `Add` method that meets the following criteria.

- The `Add` method must be available from the scope in which the collection initializer is being called. The `Add` method does not have to be public if you are using the collection initializer in a scenario where non-public methods of the collection can be accessed.
- The `Add` method must be an instance member or `Shared` member of the collection class, or an extension method.
- An `Add` method must exist that can be matched, based on overload resolution rules, to the types that are supplied in the collection initializer.

For example, the following code example shows how to create a `List(Of Customer)` collection by using a collection initializer. When the code is run, each `Customer` object is passed to the `Add(Customer)` method of the generic list.

```
Dim customers = New List(Of Customer) From
{
    New Customer("City Power & Light", "http://www.cpandl.com/"),
    New Customer("Wide World Importers", "http://www.wideworldimporters.com/"),
    New Customer("Lucerne Publishing", "http://www.lucernepublishing.com/")
}
```

The following code example shows equivalent code that does not use a collection initializer.

```
Dim customers = New List(Of Customer)
customers.Add(New Customer("City Power & Light", "http://www.cpandl.com/"))
customers.Add(New Customer("Wide World Importers", "http://www.wideworldimporters.com/"))
customers.Add(New Customer("Lucerne Publishing", "http://www.lucernepublishing.com/"))
```

If the collection has an `Add` method that has parameters that match the constructor for the `Customer` object, you could nest parameter values for the `Add` method within collection initializers, as discussed in the next section. If the collection does not have such an `Add` method, you can create one as an extension method. For an example of how to create an `Add` method as an extension method for a collection, see [How to: Create an Add Extension Method Used by a Collection Initializer](#). For an example of how to create a custom collection that can be used with a collection initializer, see [How to: Create a Collection Used by a Collection Initializer](#).

Nesting Collection Initializers

You can nest values within a collection initializer to identify a specific overload of an `Add` method for the collection that is being created. The values passed to the `Add` method must be separated by commas and enclosed in braces (`{}`), like you would do in an array literal or collection initializer.

When you create a collection by using nested values, each element of the nested value list is passed as an argument to the `Add` method that matches the element types. For example, the following code example creates a `Dictionary< TKey, TValue >` in which the keys are of type `Integer` and the values are of type `String`. Each of the nested value lists is matched to the `Add` method for the `Dictionary`.

```
Dim days = New Dictionary(Of Integer, String) From  
    {{0, "Sunday"}, {1, "Monday"}}
```

The previous code example is equivalent to the following code.

```
Dim days = New Dictionary(Of Integer, String)  
days.Add(0, "Sunday")  
days.Add(1, "Monday")
```

Only nested value lists from the first level of nesting are sent to the `Add` method for the collection type. Deeper levels of nesting are treated as array literals and the nested value lists are not matched to the `Add` method of any collection.

Related Topics

TITLE	DESCRIPTION
How to: Create an Add Extension Method Used by a Collection Initializer	Shows how to create an extension method called <code>Add</code> that can be used to populate a collection with values from a collection initializer.
How to: Create a Collection Used by a Collection Initializer	Shows how to enable use of a collection initializer by including an <code>Add</code> method in a collection class that implements <code>IEnumerable</code> .

See also

- [Collections](#)
- [Arrays](#)
- [Object Initializers: Named and Anonymous Types](#)
- [New Operator](#)
- [Auto-Implemented Properties](#)
- [How to: Initialize an Array Variable in Visual Basic](#)
- [Local Type Inference](#)
- [Anonymous Types](#)
- [Introduction to LINQ in Visual Basic](#)
- [How to: Create a List of Items](#)

Constants and Enumerations in Visual Basic

5/4/2018 • 2 minutes to read • [Edit Online](#)

Constants are a way to use meaningful names in place of a value that does not change. Constants store values that, as the name implies, remain constant throughout the execution of an application. You can use constants to provide meaningful names, instead of numbers, making your code more readable.

Enumerations provide a convenient way to work with sets of related constants, and to associate constant values with names. For example, you can declare an enumeration for a set of integer constants associated with the days of the week, and then use the names of the days rather than their integer values in your code.

In This Section

TERM	DEFINITION
Constants Overview	Topics in this section describe constants and their uses.
Enumerations Overview	Topics in this section describe enumerations and their uses.

Related Sections

TERM	DEFINITION
Const Statement	Describes the <code>Const</code> statement, which is used to declare constants.
Enum Statement	Describes the <code>Enum</code> statement, which is used to create enumerations.
Option Explicit Statement	Describes the <code>Option Explicit</code> statement, which is used at module level to force explicit declaration of all variables in that module.
Option Infer Statement	Describes the <code>Option Infer</code> statement, which enables the use of local type inference in declaring variables.
Option Strict Statement	Describes the <code>Option Strict</code> statement, which restricts implicit data type conversions to only widening conversions, disallows late binding, and disallows implicit typing that results in an <code>object</code> type.

Control Flow in Visual Basic

10/1/2019 • 2 minutes to read • [Edit Online](#)

Left unregulated, a program proceeds through its statements from beginning to end. Some very simple programs can be written with only this unidirectional flow. However, much of the power and utility of any programming language comes from the ability to change execution order with control statements and loops.

Control structures allow you to regulate the flow of your program's execution. Using control structures, you can write Visual Basic code that makes decisions or that repeats actions. Other control structures let you guarantee disposal of a resource or run a series of statements on the same object reference.

In This Section

[Decision Structures](#)

Describes control structures used for branching.

[Loop Structures](#)

Discusses control structures used to repeat processes.

[Other Control Structures](#)

Describes control structures used for resource disposal and object access.

[Nested Control Structures](#)

Covers control structures inside other control structures.

Related Sections

[Control Flow Summary](#)

Provides links to language reference pages on this subject.

Data Types in Visual Basic

4/2/2019 • 2 minutes to read • [Edit Online](#)

The *data type* of a programming element refers to what kind of data it can hold and how it stores that data. Data types apply to all values that can be stored in computer memory or participate in the evaluation of an expression. Every variable, literal, constant, enumeration, property, procedure parameter, procedure argument, and procedure return value has a data type.

Declared Data Types

You define a programming element with a declaration statement, and you specify its data type with the `As` clause. The following table shows the statements you use to declare various elements.

PROGRAMMING ELEMENT	DATA TYPE DECLARATION
Variable	In a Dim Statement <code>Dim amount As Double</code> <code>Static yourName As String</code> <code>Public billsPaid As Decimal = 0</code>
Literal	With a literal type character; see "Literal Type Characters" in Type Characters <code>Dim searchChar As Char = "." C</code>
Constant	In a Const Statement <code>Const modulus As Single = 4.17825F</code>
Enumeration	In an Enum Statement <code>Public Enum colors</code>
Property	In a Property Statement <code>Property region() As String</code>
Procedure parameter	In a Sub Statement , Function Statement , or Operator Statement <code>Sub addSale(ByVal amount As Double)</code>
Procedure argument	In the calling code; each argument is a programming element that has already been declared, or an expression containing declared elements <code>subString = Left(inputString , 5)</code>

PROGRAMMING ELEMENT	DATA TYPE DECLARATION
Procedure return value	In a Function Statement or Operator Statement <pre>Function convert(ByVal b As Byte) As String</pre>

For a list of Visual Basic data types, see [Data Types](#).

See also

- [Type Characters](#)
- [Elementary Data Types](#)
- [Composite Data Types](#)
- [Generic Types in Visual Basic](#)
- [Value Types and Reference Types](#)
- [Type Conversions in Visual Basic](#)
- [Structures](#)
- [Tuples](#)
- [Troubleshooting Data Types](#)
- [Data Types](#)
- [Efficient Use of Data Types](#)

Type characters (Visual Basic)

1/23/2019 • 3 minutes to read • [Edit Online](#)

In addition to specifying a data type in a declaration statement, you can force the data type of some programming elements with a *type character*. The type character must immediately follow the element, with no intervening characters of any kind.

The type character is not part of the name of the element. An element defined with a type character can be referenced without the type character.

Identifier type characters

Visual Basic supplies a set of *identifier type characters* that you can use in a declaration to specify the data type of a variable or constant. The following table shows the available identifier type characters with examples of usage.

IDENTIFIER TYPE CHARACTER	DATA TYPE	EXAMPLE
%	Integer	Dim L%
&	Long	Dim M&
@	Decimal	Const W@ = 37.5
!	Single	Dim Q!
#	Double	Dim X#
\$	String	Dim V\$ = "Secret"

No identifier type characters exist for the `Boolean`, `Byte`, `Char`, `Date`, `Object`, `SByte`, `Short`, `UInteger`, `ULong`, or `UShort` data types, or for any composite data types such as arrays or structures.

In some cases, you can append the `$` character to a Visual Basic function, for example `Left$` instead of `Left`, to obtain a returned value of type `String`.

In all cases, the identifier type character must immediately follow the identifier name.

Literal type characters

A *literal* is a textual representation of a particular value of a data type.

Default literal types

The form of a literal as it appears in your code ordinarily determines its data type. The following table shows these default types.

TEXTUAL FORM OF LITERAL	DEFAULT DATA TYPE	EXAMPLE
Numeric, no fractional part	Integer	2147483647

TEXTUAL FORM OF LITERAL	DEFAULT DATA TYPE	EXAMPLE
Numeric, no fractional part, too large for Integer	Long	2147483648
Numeric, fractional part	Double	1.2
Enclosed in double quotation marks	String	"A"
Enclosed within number signs	Date	#5/17/1993 9:32 AM#

Forced literal types

Visual Basic supplies a set of *literal type characters*, which you can use to force a literal to assume a data type other than the one its form indicates. You do this by appending the character to the end of the literal. The following table shows the available literal type characters with examples of usage.

LITERAL TYPE CHARACTER	DATA TYPE	EXAMPLE
S	Short	I = 347S
I	Integer	J = 347I
L	Long	K = 347L
D	Decimal	X = 347D
F	Single	Y = 347F
R	Double	Z = 347R
US	UShort	L = 347US
UI	UInteger	M = 347UI
UL	ULong	N = 347UL
C	Char	Q = ".C"

No literal type characters exist for the Boolean, Byte, Date, Object, SByte, or String data types, or for any composite data types such as arrays or structures.

Literals can also use the identifier type characters (% , & , @ , ! , # , \$), as can variables, constants, and expressions. However, the literal type characters (S , I , L , D , F , R , C) can be used only with literals.

In all cases, the literal type character must immediately follow the literal value.

Hexadecimal, binary, and octal literals

The compiler normally interprets an integer literal to be in the decimal (base 10) number system. You can also define an integer literal as a hexadecimal (base 16) number with the &H prefix, as a binary (base 2) number with the &B prefix, and as an octal (base 8) number with the &O prefix. The digits that follow the prefix must be appropriate for the number system. The following table illustrates this.

NUMBER BASE	PREFIX	VALID DIGIT VALUES	EXAMPLE
Hexadecimal (base 16)	&H	0-9 and A-F	&HFFFF
Binary (base 2)	&B	0-1	&B01111100
Octal (base 8)	&O	0-7	&077

Starting in Visual Basic 2017, you can use the underscore character (`_`) as a group separator to enhance the readability of an integral literal. The following example uses the `_` character to group a binary literal into 8-bit groups:

```
Dim number As Integer = &B00100010_11000101_11001111_11001101
```

You can follow a prefixed literal with a literal type character. The following example shows this.

```
Dim counter As Short = &H8000S
Dim flags As UShort = &H8000US
```

In the previous example, `counter` has the decimal value of -32768, and `flags` has the decimal value of +32768.

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As Integer = &H_C305_F860
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see [setting the Visual Basic language version](#).

See also

- [Data Types](#)
- [Elementary Data Types](#)
- [Value Types and Reference Types](#)
- [Type Conversions in Visual Basic](#)
- [Troubleshooting Data Types](#)
- [Variable Declaration](#)
- [Data Types](#)

Elementary Data Types (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic supplies a set of predefined data types, which you can use for many of your programming elements. This section describes these types and how to use them.

NOTE

Every elementary data type in Visual Basic is supported by a structure or a class that is in the [System](#) namespace. The compiler uses each data type keyword as an alias for the underlying structure or class. For example, declaring a variable by using the reserved word `Byte` is the same as declaring it by using the fully qualified structure name `System.Byte`.

In This Section

[Numeric Data Types](#)

Describes the integral and non-integral numeric types.

[Character Data Types](#)

Describes the `char` and `String` types.

[Miscellaneous Data Types](#)

Describes the `Boolean`, `Date`, and `Object` types.

Related Sections

[Data Types](#)

Introduces the Visual Basic data types and describes how to use them.

[Data Types](#)

Provides an overview of the elementary data types supplied by Visual Basic.

Numeric Data Types (Visual Basic)

4/2/2019 • 4 minutes to read • [Edit Online](#)

Visual Basic supplies several *numeric data types* for handling numbers in various representations. *Integral* types represent only whole numbers (positive, negative, and zero), and *nonintegral* types represent numbers with both integer and fractional parts.

For a table showing a side-by-side comparison of the Visual Basic data types, see [Data Types](#).

Integral Numeric Types

Integral data types are those that represent only numbers without fractional parts.

The *signed* integral data types are [SByte Data Type](#) (8-bit), [Short Data Type](#) (16-bit), [Integer Data Type](#) (32-bit), and [Long Data Type](#) (64-bit). If a variable always stores integers rather than fractional numbers, declare it as one of these types.

The *unsigned* integral types are [Byte Data Type](#) (8-bit), [UShort Data Type](#) (16-bit), [UInteger Data Type](#) (32-bit), and [ULong Data Type](#) (64-bit). If a variable contains binary data, or data of unknown nature, declare it as one of these types.

Performance

Arithmetic operations are faster with integral types than with other data types. They are fastest with the [Integer](#) and [UInteger](#) types in Visual Basic.

Large Integers

If you need to hold an integer larger than the [Integer](#) data type can hold, you can use the [Long](#) data type instead. [Long](#) variables can hold numbers from -9,223,372,036,854,775,808 through 9,223,372,036,854,775,807.

Operations with [Long](#) are slightly slower than with [Integer](#).

If you need even larger values, you can use the [Decimal Data Type](#). You can hold numbers from -79,228,162,514,264,337,593,543,950,335 through 79,228,162,514,264,337,593,543,950,335 in a [Decimal](#) variable if you do not use any decimal places. However, operations with [Decimal](#) numbers are considerably slower than with any other numeric data type.

Small Integers

If you do not need the full range of the [Integer](#) data type, you can use the [Short](#) data type, which can hold integers from -32,768 through 32,767. For the smallest integer range, the [SByte](#) data type holds integers from -128 through 127. If you have a very large number of variables that hold small integers, the common language runtime can sometimes store your [Short](#) and [SByte](#) variables more efficiently and save memory consumption. However, operations with [Short](#) and [SByte](#) are somewhat slower than with [Integer](#).

Unsigned Integers

If you know that your variable never needs to hold a negative number, you can use the *unsigned* types [Byte](#), [UShort](#), [UInteger](#), and [ULong](#). Each of these data types can hold a positive integer twice as large as its corresponding signed type ([SByte](#), [Short](#), [Integer](#), and [Long](#)). In terms of performance, each unsigned type is exactly as efficient as its corresponding signed type. In particular, [UInteger](#) shares with [Integer](#) the distinction of being the most efficient of all the elementary numeric data types.

Nonintegral Numeric Types

Nonintegral data types are those that represent numbers with both integer and fractional parts.

The nonintegral numeric data types are [Decimal](#) (128-bit fixed point), [Single Data Type](#) (32-bit floating point), and [Double Data Type](#) (64-bit floating point). They are all signed types. If a variable can contain a fraction, declare it as one of these types.

[Decimal](#) is not a floating-point data type. [Decimal](#) numbers have a binary integer value and an integer scaling factor that specifies what portion of the value is a decimal fraction.

You can use [Decimal](#) variables for money values. The advantage is the precision of the values. The [Double](#) data type is faster and requires less memory, but it is subject to rounding errors. The [Decimal](#) data type retains complete accuracy to 28 decimal places.

Floating-point ([single](#) and [Double](#)) numbers have larger ranges than [Decimal](#) numbers but can be subject to rounding errors. Floating-point types support fewer significant digits than [Decimal](#) but can represent values of greater magnitude.

Nonintegral number values can be expressed as mmmEeee, in which mmm is the *mantissa* (the significant digits) and eee is the *exponent* (a power of 10). The highest positive values of the nonintegral types are 7.9228162514264337593543950335E+28 for [Decimal](#), 3.4028235E+38 for [Single](#), and 1.79769313486231570E+308 for [Double](#).

Performance

[Double](#) is the most efficient of the fractional data types, because the processors on current platforms perform floating-point operations in double precision. However, operations with [Double](#) are not as fast as with the integral types such as [Integer](#).

Small Magnitudes

For numbers with the smallest possible magnitude (closest to 0), [Double](#) variables can hold numbers as small as -4.94065645841246544E-324 for negative values and 4.94065645841246544E-324 for positive values.

Small Fractional Numbers

If you do not need the full range of the [Double](#) data type, you can use the [Single](#) data type, which can hold floating-point numbers from -3.4028235E+38 through 3.4028235E+38. The smallest magnitudes for [Single](#) variables are -1.401298E-45 for negative values and 1.401298E-45 for positive values. If you have a very large number of variables that hold small floating-point numbers, the common language runtime can sometimes store your [Single](#) variables more efficiently and save memory consumption.

See also

- [Elementary Data Types](#)
- [Character Data Types](#)
- [Miscellaneous Data Types](#)
- [Troubleshooting Data Types](#)
- [How to: Call a Windows Function that Takes Unsigned Types](#)

Character Data Types (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic provides *character data types* to deal with printable and displayable characters. While they both deal with Unicode characters, `char` holds a single character whereas `string` contains an indefinite number of characters.

For a table that displays a side-by-side comparison of the Visual Basic data types, see [Data Types](#).

Char Type

The `char` data type is a single two-byte (16-bit) Unicode character. If a variable always stores exactly one character, declare it as `char`. For example:

```
' Initialize the prefix variable to the character 'a'.
Dim prefix As Char = "a"
```

Each possible value in a `char` or `String` variable is a *code point*, or character code, in the Unicode character set. Unicode characters include the basic ASCII character set, various other alphabet letters, accents, currency symbols, fractions, diacritics, and mathematical and technical symbols.

NOTE

The Unicode character set reserves the code points D800 through DFFF (55296 through 55551 decimal) for *surrogate pairs*, which require two 16-bit values to represent a single code point. A `char` variable cannot hold a surrogate pair, and a `String` uses two positions to hold such a pair.

For more information, see [Char Data Type](#).

String Type

The `String` data type is a sequence of zero or more two-byte (16-bit) Unicode characters. If a variable can contain an indefinite number of characters, declare it as `String`. For example:

```
' Initialize the name variable to "Monday".
Dim name As String = "Monday"
```

For more information, see [String Data Type](#).

See also

- [Elementary Data Types](#)
- [Composite Data Types](#)
- [Generic Types in Visual Basic](#)
- [Value Types and Reference Types](#)
- [Type Conversions in Visual Basic](#)
- [Troubleshooting Data Types](#)
- [Type Characters](#)

Miscellaneous Data Types (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic supplies several data types that are not oriented toward numbers or characters. Instead, they deal with specialized data such as yes/no values, date/time values, and object addresses.

For a table showing a side-by-side comparison of the Visual Basic data types, see [Data Types](#).

Boolean Type

The [Boolean Data Type](#) is an unsigned value that is interpreted as either `True` or `False`. Its data width depends on the implementing platform. If a variable can contain only two-state values such as true/false, yes/no, or on/off, declare it as `Boolean`.

Date Type

The [Date Data Type](#) is a 64-bit value that holds both date and time information. Each increment represents 100 nanoseconds of elapsed time since the beginning (12:00 AM) of January 1 of the year 1 in the Gregorian calendar. If a variable can contain a date value, a time value, or both, declare it as `Date`.

Object Type

The [Object Data Type](#) is a 32-bit address that points to an object instance within your application or in some other application. An `object` variable can refer to any object your application recognizes, or to data of any data type. This includes both *value types*, such as `Integer`, `Boolean`, and structure instances, and *reference types*, which are instances of objects created from classes such as `String` and `Form`, and array instances.

If a variable stores a pointer to an instance of a class that you do not know at compile time, or if it can point to data of various data types, declare it as `Object`.

The advantage of the `Object` data type is that you can use it to store data of any data type. The disadvantage is that you incur extra operations that take more execution time and make your application perform slower. If you use an `Object` variable for value types, you incur *boxing* and *unboxing*. If you use it for reference types, you incur *late binding*.

See also

- [Type Characters](#)
- [Elementary Data Types](#)
- [Numeric Data Types](#)
- [Character Data Types](#)
- [Troubleshooting Data Types](#)
- [Early and Late Binding](#)

Composite Data Types (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

In addition to the elementary data types Visual Basic supplies, you can also assemble items of different types to create *composite data types* such as structures, arrays, and classes. You can build composite data types from elementary types and from other composite types. For example, you can define an array of structure elements, or a structure with array members.

Data Types

A composite type is different from the data type of any of its components. For example, an array of `Integer` elements is not of the `Integer` data type.

An array data type is normally represented using the element type, parentheses, and commas as necessary. For example, a one-dimensional array of `String` elements is represented as `String()`, and a two-dimensional array of `Boolean` elements is represented as `Boolean(,)`.

Structure Types

There is no single data type comprising all structures. Instead, each definition of a structure represents a unique data type, even if two structures define identical elements in the same order. However, if you create two or more instances of the same structure, Visual Basic considers them to be of the same data type.

Tuples

A tuple is a lightweight structure that contains two or more fields whose types are predefined. Tuples are supported starting with Visual Basic 2017. Tuples are most commonly used to return multiple values from a single method call without having to pass arguments by reference or packaging the returned fields in a more heavy-weight class or structure. See the [Tuples](#) topic for more information on tuples.

Array Types

There is no single data type comprising all arrays. The data type of a particular instance of an array is determined by the following:

- The fact of being an array
- The rank (number of dimensions) of the array
- The element type of the array

In particular, the length of a given dimension is not part of the instance's data type. The following example illustrates this.

```
Dim arrayA( ) As Byte = New Byte(12) {}
Dim arrayB( ) As Byte = New Byte(100) {}
Dim arrayC( ) As Short = New Short(100) {}
Dim arrayD( , ) As Short
Dim arrayE( , ) As Short = New Short(4, 10) {}
```

In the preceding example, array variables `arrayA` and `arrayB` are considered to be of the same data type — `Byte()` — even though they are initialized to different lengths. Variables `arrayB` and `arrayC` are not of the same

type because their element types are different. Variables `arrayC` and `arrayD` are not of the same type because their ranks are different. Variables `arrayD` and `arrayE` have the same type — `Short(,)` — because their ranks and element types are the same, even though `arrayD` is not yet initialized.

For more information on arrays, see [Arrays](#).

Class Types

There is no single data type comprising all classes. Although one class can inherit from another class, each is a separate data type. Multiple instances of the same class are of the same data type. If you assign one class instance variable to another, not only do they have the same data type, they point to the same class instance in memory.

For more information on classes, see [Objects and Classes](#).

See also

- [Data Types](#)
- [Elementary Data Types](#)
- [Generic Types in Visual Basic](#)
- [Value Types and Reference Types](#)
- [Type Conversions in Visual Basic](#)
- [Structures](#)
- [Troubleshooting Data Types](#)
- [How to: Hold More Than One Value in a Variable](#)

How to: Hold More Than One Value in a Variable (Visual Basic)

9/17/2019 • 2 minutes to read • [Edit Online](#)

A variable holds more than one value if you declare it to be of a *composite data type*.

[Composite Data Types](#) include structures, arrays, and classes. A variable of a composite data type can hold a combination of elementary data types and other composite types. Structures and classes can hold code as well as data.

To hold more than one value in a variable

1. Determine what composite data type you want to use for your variable.
2. If the composite data type is not already defined, define it so that your variable can use it.
 - Define a structure with a [Structure Statement](#).
 - Define an array with a [Dim Statement](#).
 - Define a class with a [Class Statement](#).
3. Declare your variable with a `Dim` statement.
4. Follow the variable name with an `As` clause.
5. Follow the `As` keyword with the name of the appropriate composite data type.

See also

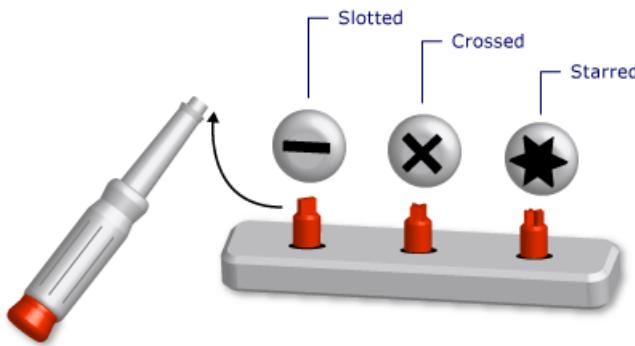
- [Data Types](#)
- [Type Characters](#)
- [Composite Data Types](#)
- [Structures](#)
- [Arrays](#)
- [Objects and Classes](#)
- [Value Types and Reference Types](#)

Generic Types in Visual Basic (Visual Basic)

5/14/2019 • 7 minutes to read • [Edit Online](#)

A *generic type* is a single programming element that adapts to perform the same functionality for a variety of data types. When you define a generic class or procedure, you do not have to define a separate version for each data type for which you might want to perform that functionality.

An analogy is a screwdriver set with removable heads. You inspect the screw you need to turn and select the correct head for that screw (slotted, crossed, starred). Once you insert the correct head in the screwdriver handle, you perform the exact same function with the screwdriver, namely turning the screw.



When you define a generic type, you parameterize it with one or more data types. This allows the using code to tailor the data types to its requirements. Your code can declare several different programming elements from the generic element, each one acting on a different set of data types. But the declared elements all perform the identical logic, no matter what data types they are using.

For example, you might want to create and use a queue class that operates on a specific data type such as `String`. You can declare such a class from `System.Collections.Generic.Queue<T>`, as the following example shows.

```
Public stringQ As New System.Collections.Generic.Queue(Of String)
```

You can now use `stringQ` to work exclusively with `String` values. Because `stringQ` is specific for `String` instead of being generalized for `object` values, you do not have late binding or type conversion. This saves execution time and reduces run-time errors.

For more information on using a generic type, see [How to: Use a Generic Class](#).

Example of a Generic Class

The following example shows a skeleton definition of a generic class.

```
Public Class classHolder(Of t)
    Public Sub processNewItem(ByVal newItem As t)
        Dim tempItem As t
        ' Insert code that processes an item of data type t.
    End Sub
End Class
```

In the preceding skeleton, `t` is a *type parameter*, that is, a placeholder for a data type that you supply when you declare the class. Elsewhere in your code, you can declare various versions of `classHolder` by supplying

various data types for `t`. The following example shows two such declarations.

```
Public integerClass As New classHolder(Of Integer)
Friend stringClass As New classHolder(Of String)
```

The preceding statements declare *constructed classes*, in which a specific type replaces the type parameter. This replacement is propagated throughout the code within the constructed class. The following example shows what the `processNewItem` procedure looks like in `integerClass`.

```
Public Sub processNewItem(ByVal newItem As Integer)
    Dim tempItem As Integer
    ' Inserted code now processes an Integer item.
End Sub
```

For a more complete example, see [How to: Define a Class That Can Provide Identical Functionality on Different Data Types](#).

Eligible Programming Elements

You can define and use generic classes, structures, interfaces, procedures, and delegates. Note that the .NET Framework defines several generic classes, structures, and interfaces that represent commonly used generic elements. The [System.Collections.Generic](#) namespace provides dictionaries, lists, queues, and stacks. Before defining your own generic element, see if it is already available in [System.Collections.Generic](#).

Procedures are not types, but you can define and use generic procedures. See [Generic Procedures in Visual Basic](#).

Advantages of Generic Types

A generic type serves as a basis for declaring several different programming elements, each of which operates on a specific data type. The alternatives to a generic type are:

1. A single type operating on the `Object` data type.
2. A set of *type-specific* versions of the type, each version individually coded and operating on one specific data type such as `String`, `Integer`, or a user-defined type such as `customer`.

A generic type has the following advantages over these alternatives:

- **Type Safety.** Generic types enforce compile-time type checking. Types based on `Object` accept any data type, and you must write code to check whether an input data type is acceptable. With generic types, the compiler can catch type mismatches before run time.
- **Performance.** Generic types do not have to *box* and *unbox* data, because each one is specialized for one data type. Operations based on `Object` must box input data types to convert them to `Object` and unbox data destined for output. Boxing and unboxing reduce performance.

Types based on `Object` are also late-bound, which means that accessing their members requires extra code at run time. This also reduces performance.

- **Code Consolidation.** The code in a generic type has to be defined only once. A set of type-specific versions of a type must replicate the same code in each version, with the only difference being the specific data type for that version. With generic types, the type-specific versions are all generated from the original generic type.
- **Code Reuse.** Code that does not depend on a particular data type can be reused with various data types

if it is generic. You can often reuse it even with a data type that you did not originally predict.

- **IDE Support.** When you use a constructed type declared from a generic type, the integrated development environment (IDE) can give you more support while you are developing your code. For example, IntelliSense can show you the type-specific options for an argument to a constructor or method.
- **Generic Algorithms.** Abstract algorithms that are type-independent are good candidates for generic types. For example, a generic procedure that sorts items using the [IComparable](#) interface can be used with any data type that implements [IComparable](#).

Constraints

Although the code in a generic type definition should be as type-independent as possible, you might need to require a certain capability of any data type supplied to your generic type. For example, if you want to compare two items for the purpose of sorting or collating, their data type must implement the [IComparable](#) interface. You can enforce this requirement by adding a *constraint* to the type parameter.

Example of a Constraint

The following example shows a skeleton definition of a class with a constraint that requires the type argument to implement [IComparable](#).

```
Public Class itemManager(Of t As IComparable)
    ' Insert code that defines class members.
End Class
```

If subsequent code attempts to construct a class from `itemManager` supplying a type that does not implement [IComparable](#), the compiler signals an error.

Types of Constraints

Your constraint can specify the following requirements in any combination:

- The type argument must implement one or more interfaces
- The type argument must be of the type of, or inherit from, at most one class
- The type argument must expose a parameterless constructor accessible to the code that creates objects from it
- The type argument must be a *reference type*, or it must be a *value type*

If you need to impose more than one requirement, you use a comma-separated *constraint list* inside braces (`{ }`). To require an accessible constructor, you include the [New Operator](#) keyword in the list. To require a reference type, you include the `Class` keyword; to require a value type, you include the `Structure` keyword.

For more information on constraints, see [Type List](#).

Example of Multiple Constraints

The following example shows a skeleton definition of a generic class with a constraint list on the type parameter. In the code that creates an instance of this class, the type argument must implement both the [IComparable](#) and [IDisposable](#) interfaces, be a reference type, and expose an accessible parameterless constructor.

```
Public Class thisClass(Of t As {IComparable, IDisposable, Class, New})
    ' Insert code that defines class members.
End Class
```

Important Terms

Generic types introduce and use the following terms:

- *Generic Type*. A definition of a class, structure, interface, procedure, or delegate for which you supply at least one data type when you declare it.
- *Type Parameter*. In a generic type definition, a placeholder for a data type you supply when you declare the type.
- *Type Argument*. A specific data type that replaces a type parameter when you declare a constructed type from a generic type.
- *Constraint*. A condition on a type parameter that restricts the type argument you can supply for it. A constraint can require that the type argument must implement a particular interface, be or inherit from a particular class, have an accessible parameterless constructor, or be a reference type or a value type. You can combine these constraints, but you can specify at most one class.
- *Constructed Type*. A class, structure, interface, procedure, or delegate declared from a generic type by supplying type arguments for its type parameters.

See also

- [Data Types](#)
- [Type Characters](#)
- [Value Types and Reference Types](#)
- [Type Conversions in Visual Basic](#)
- [Troubleshooting Data Types](#)
- [Data Types](#)
- [Of](#)
- [As](#)
- [Object Data Type](#)
- [Covariance and Contravariance](#)
- [Iterators](#)

How to: Define a Class That Can Provide Identical Functionality on Different Data Types (Visual Basic)

7/10/2019 • 4 minutes to read • [Edit Online](#)

You can define a class from which you can create objects that provide identical functionality on different data types. To do this, you specify one or more *type parameters* in the definition. The class can then serve as a template for objects that use various data types. A class defined in this way is called a *generic class*.

The advantage of defining a generic class is that you define it just once, and your code can use it to create many objects that use a wide variety of data types. This results in better performance than defining the class with the `Object` type.

In addition to classes, you can also define and use generic structures, interfaces, procedures, and delegates.

To define a class with a type parameter

1. Define the class in the normal way.
2. Add `(of typeparameter)` immediately after the class name to specify a type parameter.
3. If you have more than one type parameter, make a comma-separated list inside the parentheses. Do not repeat the `of` keyword.
4. If your code performs operations on a type parameter other than simple assignment, follow that type parameter with an `As` clause to add one or more *constraints*. A constraint guarantees that the type supplied for that type parameter satisfies a requirement such as the following:
 - Supports an operation, such as `>`, that your code performs
 - Supports a member, such as a method, that your code accesses
 - Exposes a parameterless constructor

If you do not specify any constraints, the only operations and members your code can use are those supported by the [Object Data Type](#). For more information, see [Type List](#).

5. Identify every class member that is to be declared with a supplied type, and declare it `As typeparameter`. This applies to internal storage, procedure parameters, and return values.
6. Be sure your code uses only operations and methods that are supported by any data type it can supply to `itemType`.

The following example defines a class that manages a very simple list. It holds the list in the internal array `items`, and the using code can declare the data type of the list elements. A parameterized constructor allows the using code to set the upper bound of `items`, and the parameterless constructor sets this to 9 (for a total of 10 items).

```

Public Class simpleList(Of itemType)
    Private items() As itemType
    Private top As Integer
    Private nextp As Integer
    Public Sub New()
        Me.New(9)
    End Sub
    Public Sub New(ByVal t As Integer)
        MyBase.New()
        items = New itemType(t) {}
        top = t
        nextp = 0
    End Sub
    Public Sub add(ByVal i As itemType)
        insert(i, nextp)
    End Sub
    Public Sub insert(ByVal i As itemType, ByVal p As Integer)
        If p > nextp OrElse p < 0 Then
            Throw New System.ArgumentOutOfRangeException("p",
                " less than 0 or beyond next available list position")
        ElseIf nextp > top Then
            Throw New System.ArgumentException("No room to insert at ",
                "p")
        ElseIf p < nextp Then
            For j As Integer = nextp To p + 1 Step -1
                items(j) = items(j - 1)
            Next j
        End If
        items(p) = i
        nextp += 1
    End Sub
    Public Sub remove(ByVal p As Integer)
        If p >= nextp OrElse p < 0 Then
            Throw New System.ArgumentOutOfRangeException("p",
                " less than 0 or beyond last list item")
        ElseIf nextp = 0 Then
            Throw New System.ArgumentException("List empty; cannot remove ",
                "p")
        ElseIf p < nextp - 1 Then
            For j As Integer = p To nextp - 2
                items(j) = items(j + 1)
            Next j
        End If
        nextp -= 1
    End Sub
    Public ReadOnly Property listLength() As Integer
        Get
            Return nextp
        End Get
    End Property
    Public ReadOnly Property listItem(ByVal p As Integer) As itemType
        Get
            If p >= nextp OrElse p < 0 Then
                Throw New System.ArgumentOutOfRangeException("p",
                    " less than 0 or beyond last list item")
            End If
            Return items(p)
        End Get
    End Property
End Class

```

You can declare a class from `simpleList` to hold a list of `Integer` values, another class to hold a list of `String` values, and another to hold `Date` values. Except for the data type of the list members, objects created from all these classes behave identically.

The type argument that the using code supplies to `itemType` can be an intrinsic type such as `Boolean` or `Double`, a structure, an enumeration, or any type of class, including one that your application defines.

You can test the class `simpleList` with the following code.

```
Public Sub useSimpleList()
    Dim iList As New simpleList(Of Integer)(2)
    Dim sList As New simpleList(Of String)(3)
    Dim dList As New simpleList(Of Date)(2)
    iList.add(10)
    iList.add(20)
    iList.add(30)
    sList.add("First")
    sList.add("extra")
    sList.add("Second")
    sList.add("Third")
    sList.remove(1)
    dList.add(#1/1/2003#)
    dList.add(#3/3/2003#)
    dList.insert(#2/2/2003#, 1)
    Dim s =
        "Simple list of 3 Integer items (reported length " &
        CStr(iList.listLength) & ")" &
        vbCrLf & CStr(iList.listItem(0)) &
        vbCrLf & CStr(iList.listItem(1)) &
        vbCrLf & CStr(iList.listItem(2)) &
        vbCrLf &
        "Simple list of 4 - 1 String items (reported length " &
        CStr(sList.listLength) & ")" &
        vbCrLf & CStr(sList.listItem(0)) &
        vbCrLf & CStr(sList.listItem(1)) &
        vbCrLf & CStr(sList.listItem(2)) &
        vbCrLf &
        "Simple list of 2 + 1 Date items (reported length " &
        CStr(dList.listLength) & ")" &
        vbCrLf & CStr(dList.listItem(0)) &
        vbCrLf & CStr(dList.listItem(1)) &
        vbCrLf & CStr(dList.listItem(2))
    MsgBox(s)
End Sub
```

See also

- [Data Types](#)
- [Generic Types in Visual Basic](#)
- [Language Independence and Language-Independent Components](#)
- [Of](#)
- [Type List](#)
- [How to: Use a Generic Class](#)
- [Object Data Type](#)

How to: Use a Generic Class (Visual Basic)

5/14/2019 • 2 minutes to read • [Edit Online](#)

A class that takes *type parameters* is called a *generic class*. If you are using a generic class, you can generate a *constructed class* from it by supplying a *type argument* for each of these parameters. You can then declare a variable of the constructed class type, and you can create an instance of the constructed class and assign it to that variable.

In addition to classes, you can also define and use generic structures, interfaces, procedures, and delegates.

The following procedure takes a generic class defined in the .NET Framework and creates an instance from it.

To use a class that takes a type parameter

1. At the beginning of your source file, include an [Imports Statement \(.NET Namespace and Type\)](#) to import the `System.Collections.Generic` namespace. This allows you to refer to the `System.Collections.Generic.Queue<T>` class without having to fully qualify it to differentiate it from other queue classes such as `System.Collections.Queue`.
2. Create the object in the normal way, but add `(Of type)` immediately after the class name.

The following example uses the same class (`System.Collections.Generic.Queue<T>`) to create two queue objects that hold items of different data types. It adds items to the end of each queue and then removes and displays items from the front of each queue.

```
Public Sub usequeue()
    Dim queueDouble As New System.Collections.Generic.Queue(Of Double)
    Dim queueString As New System.Collections.Generic.Queue(Of String)
    queueDouble.Enqueue(1.1)
    queueDouble.Enqueue(2.2)
    queueDouble.Enqueue(3.3)
    queueDouble.Enqueue(4.4)
    queueString.Enqueue("First string of three")
    queueString.Enqueue("Second string of three")
    queueString.Enqueue("Third string of three")
    Dim s As String = "Queue of Double items (reported length " &
        CStr(queueDouble.Count) & ")"
    For i As Integer = 1 To queueDouble.Count
        s &= vbCrLf & CStr(queueDouble.Dequeue())
    Next i
    s &= vbCrLf & "Queue of String items (reported length " &
        CStr(queueString.Count) & ")"
    For i As Integer = 1 To queueString.Count
        s &= vbCrLf & queueString.Dequeue()
    Next i
    MsgBox(s)
End Sub
```

See also

- [Data Types](#)
- [Generic Types in Visual Basic](#)
- [Language Independence and Language-Independent Components](#)
- [Of](#)
- [Imports Statement \(.NET Namespace and Type\)](#)

- How to: Define a Class That Can Provide Identical Functionality on Different Data Types
- Iterators

Generic Procedures in Visual Basic

4/2/2019 • 3 minutes to read • [Edit Online](#)

A *generic procedure*, also called a *generic method*, is a procedure defined with at least one type parameter. This allows the calling code to tailor the data types to its requirements each time it calls the procedure.

A procedure is not generic simply by virtue of being defined inside a generic class or a generic structure. To be generic, the procedure must take at least one type parameter, in addition to any normal parameters it might take. A generic class or structure can contain nongeneric procedures, and a nongeneric class, structure, or module can contain generic procedures.

A generic procedure can use its type parameters in its normal parameter list, in its return type if it has one, and in its procedure code.

Type Inference

You can call a generic procedure without supplying any type arguments at all. If you call it this way, the compiler attempts to determine the appropriate data types to pass to the procedure's type arguments. This is called *type inference*. The following code shows a call in which the compiler infers that it should pass type `String` to the type parameter `t`.

```
Public Sub testSub(Of t)(ByVal arg As t)
End Sub
Public Sub callTestSub()
    testSub("Use this string")
End Sub
```

If the compiler cannot infer the type arguments from the context of your call, it reports an error. One possible cause of such an error is an array rank mismatch. For example, suppose you define a normal parameter as an array of a type parameter. If you call the generic procedure supplying an array of a different rank (number of dimensions), the mismatch causes type inference to fail. The following code shows a call in which a two-dimensional array is passed to a procedure that expects a one-dimensional array.

```
Public Sub demoSub(Of t)(ByVal arg() As t)
End Sub

Public Sub callDemoSub()
    Dim twoDimensions(,) As Integer
    demoSub(twoDimensions)
End Sub
```

You can invoke type inference only by omitting all the type arguments. If you supply one type argument, you must supply them all.

Type inference is supported only for generic procedures. You cannot invoke type inference on generic classes, structures, interfaces, or delegates.

Example

Description

The following example defines a generic `Function` procedure to find a particular element in an array. It defines

one type parameter and uses it to construct the two parameters in the parameter list.

Code

```
Public Function findElement(Of T As IComparable) (
    ByVal searchArray As T(), ByVal searchValue As T) As Integer

    If searchArray.GetLength(0) > 0 Then
        For i As Integer = 0 To searchArray.GetUpperBound(0)
            If searchArray(i).CompareTo(searchValue) = 0 Then Return i
        Next i
    End If

    Return -1
End Function
```

Comments

The preceding example requires the ability to compare `searchValue` against each element of `searchArray`. To guarantee this ability, it constrains the type parameter `T` to implement the `IComparable<T>` interface. The code uses the `CompareTo` method instead of the `=` operator, because there is no guarantee that a type argument supplied for `T` supports the `=` operator.

You can test the `findElement` procedure with the following code.

```
Public Sub tryFindElement()
    Dim stringArray() As String = {"abc", "def", "xyz"}
    Dim stringSearch As String = "abc"
    Dim integerArray() As Integer = {7, 8, 9}
    Dim integerSearch As Integer = 8
    Dim dateArray() As Date = {[#4/17/1969#, #9/20/1998#, #5/31/2004#]}
    Dim dateSearch As Date = Microsoft.VisualBasic.DateAndTime.Today
    MsgBox(CStr(findElement(Of String)(stringArray, stringSearch)))
    MsgBox(CStr(findElement(Of Integer)(integerArray, integerSearch)))
    MsgBox(CStr(findElement(Of Date)(dateArray, dateSearch)))
End Sub
```

The preceding calls to `MsgBox` display "0", "1", and "-1" respectively.

See also

- [Generic Types in Visual Basic](#)
- [How to: Define a Class That Can Provide Identical Functionality on Different Data Types](#)
- [How to: Use a Generic Class](#)
- [Procedures](#)
- [Procedure Parameters and Arguments](#)
- [Type List](#)
- [Parameter List](#)

Nullable Value Types (Visual Basic)

9/27/2019 • 5 minutes to read • [Edit Online](#)

Sometimes you work with a value type that does not have a defined value in certain circumstances. For example, a field in a database might have to distinguish between having an assigned value that is meaningful and not having an assigned value. Value types can be extended to take either their normal values or a null value. Such an extension is called a *nullable type*.

Each nullable type is constructed from the generic `Nullable<T>` structure. Consider a database that tracks work-related activities. The following example constructs a nullable `Boolean` type and declares a variable of that type. You can write the declaration in three ways:

```
Dim ridesBusToWork1? As Boolean  
Dim ridesBusToWork2 As Boolean?  
Dim ridesBusToWork3 As Nullable(Of Boolean)
```

The variable `ridesBusToWork` can hold a value of `True`, a value of `False`, or no value at all. Its initial default value is no value at all, which in this case could mean that the information has not yet been obtained for this person. In contrast, `False` could mean that the information has been obtained and the person does not ride the bus to work.

You can declare variables and properties with nullable types, and you can declare an array with elements of a nullable type. You can declare procedures with nullable types as parameters, and you can return a nullable type from a `Function` procedure.

You cannot construct a nullable type on a reference type such as an array, a `String`, or a class. The underlying type must be a value type. For more information, see [Value Types and Reference Types](#).

Using a Nullable Type Variable

The most important members of a nullable type are its `HasValue` and `Value` properties. For a variable of a nullable type, `HasValue` tells you whether the variable contains a defined value. If `HasValue` is `True`, you can read the value from `Value`. Note that both `HasValue` and `Value` are `ReadOnly` properties.

Default Values

When you declare a variable with a nullable type, its `HasValue` property has a default value of `False`. This means that by default the variable has no defined value, instead of the default value of its underlying value type. In the following example, the variable `numberOfChildren` initially has no defined value, even though the default value of the `Integer` type is 0.

```
Dim numberOfChildren? As Integer
```

A null value is useful to indicate an undefined or unknown value. If `numberOfChildren` had been declared as `Integer`, there would be no value that could indicate that the information is not currently available.

Storing Values

You store a value in a variable or property of a nullable type in the typical way. The following example assigns a value to the variable `numberOfChildren` declared in the previous example.

```
numberOfChildren = 2
```

If a variable or property of a nullable type contains a defined value, you can cause it to revert to its initial state of not having a value assigned. You do this by setting the variable or property to `Nothing`, as the following example shows.

```
numberOfChildren = Nothing
```

NOTE

Although you can assign `Nothing` to a variable of a nullable type, you cannot test it for `Nothing` by using the equal sign. Comparison that uses the equal sign, `someVar = Nothing`, always evaluates to `Nothing`. You can test the variable's `HasValue` property for `False`, or test by using the `Is` or `IsNot` operator.

Retrieving Values

To retrieve the value of a variable of a nullable type, you should first test its `HasValue` property to confirm that it has a value. If you try to read the value when `HasValue` is `False`, Visual Basic throws an `InvalidOperationException` exception. The following example shows the recommended way to read the variable `numberOfChildren` of the previous examples.

```
If numberOfChildren.HasValue Then  
    MsgBox("There are " & CStr(numberOfChildren) & " children.")  
Else  
    MsgBox("It is not known how many children there are.")  
End If
```

Comparing Nullable Types

When nullable `Boolean` variables are used in Boolean expressions, the result can be `True`, `False`, or `Nothing`. The following is the truth table for `And` and `Or`. Because `b1` and `b2` now have three possible values, there are nine combinations to evaluate.

B1	B2	B1 AND B2	B1 OR B2
<code>Nothing</code>	<code>Nothing</code>	<code>Nothing</code>	<code>Nothing</code>
<code>Nothing</code>	<code>True</code>	<code>Nothing</code>	<code>True</code>
<code>Nothing</code>	<code>False</code>	<code>False</code>	<code>Nothing</code>
<code>True</code>	<code>Nothing</code>	<code>Nothing</code>	<code>True</code>
<code>True</code>	<code>True</code>	<code>True</code>	<code>True</code>
<code>True</code>	<code>False</code>	<code>False</code>	<code>True</code>
<code>False</code>	<code>Nothing</code>	<code>False</code>	<code>Nothing</code>
<code>False</code>	<code>True</code>	<code>False</code>	<code>True</code>
<code>False</code>	<code>False</code>	<code>False</code>	<code>False</code>

When the value of a Boolean variable or expression is `Nothing`, it is neither `true` nor `false`. Consider the

following example.

```
Dim b1? As Boolean
Dim b2? As Boolean
b1 = True
b2 = Nothing

' The following If statement displays "Expression is not true".
If (b1 And b2) Then
    Console.WriteLine("Expression is true")
Else
    Console.WriteLine("Expression is not true")
End If

' The following If statement displays "Expression is not false".
If Not (b1 And b2) Then
    Console.WriteLine("Expression is false")
Else
    Console.WriteLine("Expression is not false")
End If
```

In this example, `b1 And b2` evaluates to `Nothing`. As a result, the `Else` clause is executed in each `If` statement, and the output is as follows:

```
Expression is not true
```

```
Expression is not false
```

NOTE

`AndAlso` and `OrElse`, which use short-circuit evaluation, must evaluate their second operands when the first evaluates to `Nothing`.

Propagation

If one or both of the operands of an arithmetic, comparison, shift, or type operation is nullable, the result of the operation is also nullable. If both operands have values that are not `Nothing`, the operation is performed on the underlying values of the operands, as if neither were a nullable type. In the following example, variables `compare1` and `sum1` are implicitly typed. If you rest the mouse pointer over them, you will see that the compiler infers nullable types for both of them.

```
' Variable n is a nullable type, but both m and n have proper values.
Dim m As Integer = 3
Dim n? As Integer = 2

' The comparison evaluated is 3 > 2, but compare1 is inferred to be of
' type Boolean?.
Dim compare1 = m > n
' The values summed are 3 and 2, but sum1 is inferred to be of type Integer?.
Dim sum1 = m + n

' The following line displays: 3 * 2 * 5 * True
Console.WriteLine($"{m} * {n} * {sum1} * {compare1}")
```

If one or both operands have a value of `Nothing`, the result will be `Nothing`.

```
' Change the value of n to Nothing.  
n = Nothing  
  
Dim compare2 = m > n  
Dim sum2 = m + n  
  
' Because the values of n, compare2, and sum2 are all Nothing, the  
' following line displays: 3 * <null> * <null> * <null>  
Console.WriteLine($"{m} * {If(n, "<null>")} * {If(sum2, "<null>")} * {If(compare2, "<null>")})
```

Using Nullable Types with Data

A database is one of the most important places to use nullable types. Not all database objects currently support nullable types, but the designer-generated table adapters do. See [TableAdapter support for nullable types](#).

See also

- [InvalidOperationException](#)
- [HasValue](#)
- [Data Types](#)
- [Value Types and Reference Types](#)
- [Troubleshooting Data Types](#)
- [Fill datasets by using TableAdapters](#)
- [If Operator](#)
- [Local Type Inference](#)
- [Is Operator](#)
- [IsNot Operator](#)
- [Using Nullable Value Types \(C#\)](#)

Value Types and Reference Types

10/18/2019 • 2 minutes to read • [Edit Online](#)

There are two kinds of types in Visual Basic: reference types and value types. Variables of reference types store references to their data (objects), while variables of value types directly contain their data. With reference types, two variables can reference the same object; therefore, operations on one variable can affect the object referenced by the other variable. With value types, each variable has its own copy of the data, and it is not possible for operations on one variable to affect the other (except in the case of the [ByRef modifier on parameters](#)).

Value Types

A data type is a *value type* if it holds the data within its own memory allocation. Value types include the following:

- All numeric data types
- `Boolean`, `Char`, and `Date`
- All structures, even if their members are reference types
- Enumerations, since their underlying type is always `SByte`, `Short`, `Integer`, `Long`, `Byte`, `UShort`, `UInteger`, or `ULong`

Every structure is a value type, even if it contains reference type members. For this reason, value types such as `Char` and `Integer` are implemented by .NET Framework structures.

You can declare a value type by using the reserved keyword, for example, `Decimal`. You can also use the `New` keyword to initialize a value type. This is especially useful if the type has a constructor that takes parameters. An example of this is the `Decimal(Int32, Int32, Int32, Boolean, Byte)` constructor, which builds a new `Decimal` value from the supplied parts.

Reference Types

A *reference type* stores a reference to its data. Reference types include the following:

- `String`
- All arrays, even if their elements are value types
- Class types, such as `Form`
- Delegates

A class is a *reference type*. Note that every array is a reference type, even if its members are value types.

Since every reference type represents an underlying .NET Framework class, you must use the `New Operator` keyword when you initialize it. The following statement initializes an array.

```
Dim totals() As Single = New Single(8) {}
```

Elements That Are Not Types

The following programming elements do not qualify as types, because you cannot specify any of them as a data type for a declared element:

- Namespaces
- Modules
- Events
- Properties and procedures
- Variables, constants, and fields

Working with the Object Data Type

You can assign either a reference type or a value type to a variable of the `Object` data type. An `Object` variable always holds a reference to the data, never the data itself. However, if you assign a value type to an `Object` variable, it behaves as if it holds its own data. For more information, see [Object Data Type](#).

You can find out whether an `Object` variable is acting as a reference type or a value type by passing it to the `IsReference` method in the `Information` class of the `Microsoft.VisualBasic` namespace. `Information.IsReference` returns `True` if the content of the `Object` variable represents a reference type.

See also

- [Nullable Value Types](#)
- [Type Conversions in Visual Basic](#)
- [Structure Statement](#)
- [Efficient Use of Data Types](#)
- [Object Data Type](#)
- [Data Types](#)

Type Conversions in Visual Basic

8/24/2018 • 2 minutes to read • [Edit Online](#)

The process of changing a value from one data type to another type is called *conversion*. Conversions are either *widening* or *narrowing*, depending on the data capacities of the types involved. They are also *implicit* or *explicit*, depending on the syntax in the source code.

In This Section

[Widening and Narrowing Conversions](#)

Explains conversions classified by whether the destination type can hold the data.

[Implicit and Explicit Conversions](#)

Discusses conversions classified by whether Visual Basic performs them automatically.

[Conversions Between Strings and Other Types](#)

Illustrates converting between strings and numeric, `Boolean`, or date/time values.

[How to: Convert an Object to Another Type in Visual Basic](#)

Shows how to convert an `Object` variable to any other data type.

[Array Conversions](#)

Steps you through the process of converting between arrays of different data types.

Related Sections

[Data Types](#)

Introduces the Visual Basic data types and describes how to use them.

[Data Types](#)

Lists the elementary data types supplied by Visual Basic.

[Troubleshooting Data Types](#)

Discusses some common problems that can arise when working with data types.

Widening and Narrowing Conversions (Visual Basic)

10/18/2019 • 4 minutes to read • [Edit Online](#)

An important consideration with a type conversion is whether the result of the conversion is within the range of the destination data type.

A *widening conversion* changes a value to a data type that can allow for any possible value of the original data. Widening conversions preserve the source value but can change its representation. This occurs if you convert from an integral type to `Decimal`, or from `Char` to `String`.

A *narrowing conversion* changes a value to a data type that might not be able to hold some of the possible values. For example, a fractional value is rounded when it is converted to an integral type, and a numeric type being converted to `Boolean` is reduced to either `True` or `False`.

Widening Conversions

The following table shows the standard widening conversions.

DATA TYPE	WIDENS TO DATA TYPES ¹
<code>SByte</code>	<code>SByte</code> , <code>Short</code> , <code>Integer</code> , <code>Long</code> , <code>Decimal</code> , <code>Single</code> , <code>Double</code>
<code>Byte</code>	<code>Byte</code> , <code>Short</code> , <code>UShort</code> , <code>Integer</code> , <code>UInteger</code> , <code>Long</code> , <code>ULong</code> , <code>Decimal</code> , <code>Single</code> , <code>Double</code>
<code>Short</code>	<code>Short</code> , <code>Integer</code> , <code>Long</code> , <code>Decimal</code> , <code>Single</code> , <code>Double</code>
<code>UShort</code>	<code>UShort</code> , <code>Integer</code> , <code>UInteger</code> , <code>Long</code> , <code>ULong</code> , <code>Decimal</code> , <code>Single</code> , <code>Double</code>
<code>Integer</code>	<code>Integer</code> , <code>Long</code> , <code>Decimal</code> , <code>Single</code> , <code>Double</code> ²
<code>UInteger</code>	<code>UInteger</code> , <code>Long</code> , <code>ULong</code> , <code>Decimal</code> , <code>Single</code> , <code>Double</code> ²
<code>Long</code>	<code>Long</code> , <code>Decimal</code> , <code>Single</code> , <code>Double</code> ²
<code>ULong</code>	<code>ULong</code> , <code>Decimal</code> , <code>Single</code> , <code>Double</code> ²
<code>Decimal</code>	<code>Decimal</code> , <code>Single</code> , <code>Double</code> ²
<code>Single</code>	<code>Single</code> , <code>Double</code>
<code>Double</code>	<code>Double</code>
Any enumerated type (Enum)	Its underlying integral type and any type to which the underlying type widens.
<code>Char</code>	<code>Char</code> , <code>String</code>

DATA TYPE	WIDENS TO DATA TYPES
Char array	Char array, String
Any type	Object
Any derived type	Any base type from which it is derived ³ .
Any type	Any interface it implements.
Nothing	Any data type or object type.

¹ By definition, every data type widens to itself.

² Conversions from Integer, UInteger, Long, ULong, or Decimal to Single or Double might result in loss of precision, but never in loss of magnitude. In this sense they do not incur information loss.

³ It might seem surprising that a conversion from a derived type to one of its base types is widening. The justification is that the derived type contains all the members of the base type, so it qualifies as an instance of the base type. In the opposite direction, the base type does not contain any new members defined by the derived type.

Widening conversions always succeed at run time and never incur data loss. You can always perform them implicitly, whether the [Option Strict Statement](#) sets the type checking switch to `on` or to `off`.

Narrowing Conversions

The standard narrowing conversions include the following:

- The reverse directions of the widening conversions in the preceding table (except that every type widens to itself)
- Conversions in either direction between Boolean and any numeric type
- Conversions from any numeric type to any enumerated type (`Enum`)
- Conversions in either direction between String and any numeric type, Boolean, or Date
- Conversions from a data type or object type to a type derived from it

Narrowing conversions do not always succeed at run time, and can fail or incur data loss. An error occurs if the destination data type cannot receive the value being converted. For example, a numeric conversion can result in an overflow. The compiler does not allow you to perform narrowing conversions implicitly unless the [Option Strict Statement](#) sets the type checking switch to `off`.

NOTE

The narrowing-conversion error is suppressed for conversions from the elements in a `For Each...Next` collection to the loop control variable. For more information and examples, see the "Narrowing Conversions" section in [For Each...Next Statement](#).

When to Use Narrowing Conversions

You use a narrowing conversion when you know the source value can be converted to the destination data type without error or data loss. For example, if you have a `String` that you know contains either "True" or "False," you can use the `CBool` keyword to convert it to `Boolean`.

Exceptions During Conversion

Because widening conversions always succeed, they do not throw exceptions. Narrowing conversions, when they fail, most commonly throw the following exceptions:

- [InvalidCastException](#) — if no conversion is defined between the two types
- [OverflowException](#) — (integral types only) if the converted value is too large for the target type

If a class or structure defines a [CType Function](#) to serve as a conversion operator to or from that class or structure, that `CType` can throw any exception it deems appropriate. In addition, that `CType` might call Visual Basic functions or .NET Framework methods, which in turn could throw a variety of exceptions.

Changes During Reference Type Conversions

A conversion from a *reference type* copies only the pointer to the value. The value itself is neither copied nor changed in any way. The only thing that can change is the data type of the variable holding the pointer. In the following example, the data type is converted from the derived class to its base class, but the object that both variables now point to is unchanged.

```
' Assume class cSquare inherits from class cShape.  
Dim shape As cShape  
Dim square As cSquare = New cSquare  
' The following statement performs a widening  
' conversion from a derived class to its base class.  
shape = square
```

See also

- [Data Types](#)
- [Type Conversions in Visual Basic](#)
- [Implicit and Explicit Conversions](#)
- [Conversions Between Strings and Other Types](#)
- [How to: Convert an Object to Another Type in Visual Basic](#)
- [Array Conversions](#)
- [Data Types](#)
- [Type Conversion Functions](#)

Implicit and Explicit Conversions (Visual Basic)

7/26/2019 • 4 minutes to read • [Edit Online](#)

An *implicit conversion* does not require any special syntax in the source code. In the following example, Visual Basic implicitly converts the value of `k` to a single-precision floating-point value before assigning it to `q`.

```
Dim k As Integer
Dim q As Double
' Integer widens to Double, so you can do this with Option Strict On.
k = 432
q = k
```

An *explicit conversion* uses a type conversion keyword. Visual Basic provides several such keywords, which coerce an expression in parentheses to the desired data type. These keywords act like functions, but the compiler generates the code inline, so execution is slightly faster than with a function call.

In the following extension of the preceding example, the `CInt` keyword converts the value of `q` back to an integer before assigning it to `k`.

```
' q had been assigned the value 432 from k.
q = Math.Sqrt(q)
k = CInt(q)
' k now has the value 21 (rounded square root of 432).
```

Conversion Keywords

The following table shows the available conversion keywords.

TYPE CONVERSION KEYWORD	CONVERTS AN EXPRESSION TO DATA TYPE	ALLOWABLE DATA TYPES OF EXPRESSION TO BE CONVERTED
<code>CBool</code>	Boolean Data Type	Any numeric type (including <code>Byte</code> , <code>SByte</code> , and enumerated types), <code>String</code> , <code>Object</code>
<code>CByte</code>	Byte Data Type	Any numeric type (including <code>SByte</code> and enumerated types), <code>Boolean</code> , <code>String</code> , <code>Object</code>
<code>CChar</code>	Char Data Type	<code>String</code> , <code>Object</code>
<code>CDate</code>	Date Data Type	<code>String</code> , <code>Object</code>
<code>CDbl</code>	Double Data Type	Any numeric type (including <code>Byte</code> , <code>SByte</code> , and enumerated types), <code>Boolean</code> , <code>String</code> , <code>Object</code>
<code>CDec</code>	Decimal Data Type	Any numeric type (including <code>Byte</code> , <code>SByte</code> , and enumerated types), <code>Boolean</code> , <code>String</code> , <code>Object</code>

TYPE CONVERSION KEYWORD	CONVERTS AN EXPRESSION TO DATA TYPE	ALLOWABLE DATA TYPES OF EXPRESSION TO BE CONVERTED
<code>CInt</code>	Integer Data Type	Any numeric type (including <code>Byte</code> , <code>SByte</code> , and enumerated types), <code>Boolean</code> , <code>String</code> , <code>Object</code>
<code>CLng</code>	Long Data Type	Any numeric type (including <code>Byte</code> , <code>SByte</code> , and enumerated types), <code>Boolean</code> , <code>String</code> , <code>Object</code>
<code>cobj</code>	Object Data Type	Any type
<code>CSByte</code>	SByte Data Type	Any numeric type (including <code>Byte</code> and enumerated types), <code>Boolean</code> , <code>String</code> , <code>Object</code>
<code>CShort</code>	Short Data Type	Any numeric type (including <code>Byte</code> , <code>SByte</code> , and enumerated types), <code>Boolean</code> , <code>String</code> , <code>Object</code>
<code>CSng</code>	Single Data Type	Any numeric type (including <code>Byte</code> , <code>SByte</code> , and enumerated types), <code>Boolean</code> , <code>String</code> , <code>Object</code>
<code>CStr</code>	String Data Type	Any numeric type (including <code>Byte</code> , <code>SByte</code> , and enumerated types), <code>Boolean</code> , <code>Char</code> , <code>Char array</code> , <code>Date</code> , <code>Object</code>
<code> CType</code>	Type specified following the comma (,)	<p>When converting to an <i>elementary data type</i> (including an array of an elementary type), the same types as allowed for the corresponding conversion keyword</p> <p>When converting to a <i>composite data type</i>, the interfaces it implements and the classes from which it inherits</p> <p>When converting to a class or structure on which you have overloaded <code> CType</code>, that class or structure</p>
<code>CUInt</code>	UInteger Data Type	Any numeric type (including <code>Byte</code> , <code>SByte</code> , and enumerated types), <code>Boolean</code> , <code>String</code> , <code>Object</code>
<code>CULng</code>	ULong Data Type	Any numeric type (including <code>Byte</code> , <code>SByte</code> , and enumerated types), <code>Boolean</code> , <code>String</code> , <code>Object</code>
<code>CUShort</code>	UShort Data Type	Any numeric type (including <code>Byte</code> , <code>SByte</code> , and enumerated types), <code>Boolean</code> , <code>String</code> , <code>Object</code>

The CType Function

The [CType Function](#) operates on two arguments. The first is the expression to be converted, and the second is the destination data type or object class. Note that the first argument must be an expression, not a type.

`CType` is an *inline function*, meaning the compiled code makes the conversion, often without generating a function call. This improves performance.

For a comparison of `CType` with the other type conversion keywords, see [DirectCast Operator](#) and [TryCast Operator](#).

Elementary Types

The following example demonstrates the use of `CType`.

```
k = CType(q, Integer)
' The following statement coerces w to the specific object class Label.
f = CType(w, Label)
```

Composite Types

You can use `CType` to convert values to composite data types as well as to elementary types. You can also use it to coerce an object class to the type of one of its interfaces, as in the following example.

```
' Assume class cZone implements interface iZone.
Dim h As Object
' The first argument to CType must be an expression, not a type.
Dim cZ As cZone
' The following statement coerces a cZone object to its interface iZone.
h = CType(cZ, iZone)
```

Array Types

`CType` can also convert array data types, as in the following example.

```
Dim v() As classV
Dim obArray() As Object
' Assume some object array has been assigned to obArray.
' Check for run-time type compatibility.
If TypeOf obArray Is classV()
    ' obArray can be converted to classV.
    v = CType(obArray, classV())
End If
```

For more information and an example, see [Array Conversions](#).

Types Defining CType

You can define `CType` on a class or structure you have defined. This allows you to convert values to and from the type of your class or structure. For more information and an example, see [How to: Define a Conversion Operator](#).

NOTE

Values used with a conversion keyword must be valid for the destination data type, or an error occurs. For example, if you attempt to convert a `Long` to an `Integer`, the value of the `Long` must be within the valid range for the `Integer` data type.

Specifying `CType` to convert from one class type to another fails at run time if the source type does not derive from the destination type. Such a failure throws an `InvalidOperationException` exception.

However, if one of the types is a structure or class you have defined, and if you have defined `CType` on that structure or class, a conversion can succeed if it satisfies the requirements of your `CType`. See [How to: Define a Conversion Operator](#).

Performing an explicit conversion is also known as *casting* an expression to a given data type or object class.

See also

- [Type Conversions in Visual Basic](#)
- [Conversions Between Strings and Other Types](#)
- [How to: Convert an Object to Another Type in Visual Basic](#)
- [Structures](#)
- [Data Types](#)
- [Type Conversion Functions](#)
- [Troubleshooting Data Types](#)

Conversions Between Strings and Other Types (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

You can convert a numeric, `Boolean`, or date/time value to a `String`. You can also convert in the reverse direction — from a string value to numeric, `Boolean`, or `Date` — provided the contents of the string can be interpreted as a valid value of the destination data type. If they cannot, a run-time error occurs.

The conversions for all these assignments, in either direction, are narrowing conversions. You should use the type conversion keywords (`CBool`, `CByte`, `CDate`, `CDbl`, `CDec`, `CInt`, `CLng`, `CSByte`, `CShort`, `CSng`, `CStr`, `CUInt`, `CULng`, `CUShort`, and `CType`). The `Format` and `Val` functions give you additional control over conversions between strings and numbers.

If you have defined a class or structure, you can define type conversion operators between `String` and the type of your class or structure. For more information, see [How to: Define a Conversion Operator](#).

Conversion of Numbers to Strings

You can use the `Format` function to convert a number to a formatted string, which can include not only the appropriate digits but also formatting symbols such as a currency sign (such as `$`), thousands separators or *digit grouping symbols* (such as `,`), and a decimal separator (such as `.`). `Format` automatically uses the appropriate symbols according to the **Regional Options** settings specified in the Windows **Control Panel**.

Note that the concatenation (`&`) operator can convert a number to a string implicitly, as the following example shows.

```
' The following statement converts count to a String value.  
Str = "The total count is " & count
```

Conversion of Strings to Numbers

You can use the `Val` function to explicitly convert the digits in a string to a number. `Val` reads the string until it encounters a character other than a digit, space, tab, line feed, or period. The sequences "&O" and "&H" alter the base of the number system and terminate the scanning. Until it stops reading, `Val` converts all appropriate characters to a numeric value. For example, the following statement returns the value `141.825`.

```
Val(" 14 1.825 miles")
```

When Visual Basic converts a string to a numeric value, it uses the **Regional Options** settings specified in the Windows **Control Panel** to interpret the thousands separator, decimal separator, and currency symbol. This means that a conversion might succeed under one setting but not another. For example, `"$14.20"` is acceptable in the English (United States) locale but not in any French locale.

See also

- [Type Conversions in Visual Basic](#)
- [Widening and Narrowing Conversions](#)
- [Implicit and Explicit Conversions](#)
- [How to: Convert an Object to Another Type in Visual Basic](#)

- [Array Conversions](#)
- [Data Types](#)
- [Type Conversion Functions](#)
- [Introduction to International Applications Based on the .NET Framework](#)

How to: Convert an Object to Another Type in Visual Basic

10/18/2019 • 2 minutes to read • [Edit Online](#)

You convert an `Object` variable to another data type by using a conversion keyword such as [CType Function](#).

Example

The following example converts an `Object` variable to an `Integer` and a `String`.

```
Public Sub objectConversion(ByVal anObject As Object)
    Dim anInteger As Integer
    Dim aString As String
    anInteger = CType(anObject, Integer)
    aString = CType(anObject, String)
End Sub
```

If you know that the contents of an `Object` variable are of a particular data type, it is better to convert the variable to that data type. If you continue to use the `Object` variable, you incur either *boxing* and *unboxing* (for a value type) or *late binding* (for a reference type). These operations all take extra execution time and make your performance slower.

Compiling the Code

This example requires:

- A reference to the [System](#) namespace.

See also

- [Object](#)
- [Type Conversions in Visual Basic](#)
- [Widening and Narrowing Conversions](#)
- [Implicit and Explicit Conversions](#)
- [Conversions Between Strings and Other Types](#)
- [Array Conversions](#)
- [Structures](#)
- [Data Types](#)
- [Type Conversion Functions](#)

Array Conversions (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

You can convert an array type to a different array type provided you meet the following conditions:

- **Equal Rank.** The ranks of the two arrays must be the same, that is, they must have the same number of dimensions. However, the lengths of the respective dimensions do not need to be the same.
- **Element Data Type.** The data types of the elements of both arrays must be reference types. You cannot convert an `Integer` array to a `Long` array, or even to an `Object` array, because at least one value type is involved. For more information, see [Value Types and Reference Types](#).
- **Convertibility.** A conversion, either widening or narrowing, must be possible between the element types of the two arrays. An example that fails this requirement is an attempted conversion between a `String` array and an array of a class derived from `System.Attribute`. These two types have nothing in common, and no conversion of any kind exists between them.

A conversion of one array type to another is widening or narrowing depending on whether the conversion of the respective elements is widening or narrowing. For more information, see [Widening and Narrowing Conversions](#).

Conversion to an Object Array

When you declare an `Object` array without initializing it, its element type is `Object` as long as it remains uninitialized. When you set it to an array of a specific class, it takes on the type of that class. However, its underlying type is still `Object`, and you can subsequently set it to another array of an unrelated class. Since all classes derive from `Object`, you can change the array's element type from any class to any other class.

In the following example, no conversion exists between types `student` and `String`, but both derive from `Object`, so all assignments are valid.

```
' Assume student has already been defined as a class.  
Dim testArray() As Object  
' testArray is still an Object array at this point.  
Dim names() As String = New String(3) {"Name0", "Name1", "Name2", "Name3"}  
testArray = New student(3) {}  
' testArray is now of type student().  
testArray = names  
' testArray is now a String array.
```

Underlying Type of an Array

If you originally declare an array with a specific class, its underlying element type is that class. If you subsequently set it to an array of another class, there must be a conversion between the two classes.

In the following example, `students` is a `student` array. Since no conversion exists between `String` and `student`, the last statement fails.

```
Dim students() As student  
Dim names() As String = New String(3) {"Name0", "Name1", "Name2", "Name3"}  
students = New Student(3) {}  
' The following statement fails at compile time.  
students = names
```

See also

- [Data Types](#)
- [Type Conversions in Visual Basic](#)
- [Implicit and Explicit Conversions](#)
- [Conversions Between Strings and Other Types](#)
- [How to: Convert an Object to Another Type in Visual Basic](#)
- [Data Types](#)
- [Type Conversion Functions](#)
- [Arrays](#)

Structures (Visual Basic)

8/24/2018 • 2 minutes to read • [Edit Online](#)

A *structure* is a generalization of the user-defined type (UDT) supported by previous versions of Visual Basic. In addition to fields, structures can expose properties, methods, and events. A structure can implement one or more interfaces, and you can declare individual access levels for each field.

You can combine data items of different types to create a structure. A structure associates one or more *elements* with each other and with the structure itself. When you declare a structure, it becomes a *composite data type*, and you can declare variables of that type.

Structures are useful when you want a single variable to hold several related pieces of information. For example, you might want to keep an employee's name, telephone extension, and salary together. You could use several variables for this information, or you could define a structure and use it for a single employee variable. The advantage of the structure becomes apparent when you have many employees and therefore many instances of the variable.

In This Section

[How to: Declare a Structure](#)

Shows how to declare a structure and its elements.

[Structure Variables](#)

Covers assigning a structure to a variable and accessing its elements.

[Structures and Other Programming Elements](#)

Summarizes how structures interact with arrays, objects, procedures, and each other.

[Structures and Classes](#)

Describes the similarities and differences between structures and classes.

Related Sections

[Data Types](#)

Introduces the Visual Basic data types and describes how to use them.

[Data Types](#)

Lists the elementary data types supplied by Visual Basic.

How to: Declare a Structure (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

You begin a structure declaration with the [Structure Statement](#), and you end it with the `End Structure` statement. Between these two statements you must declare at least one *element*. The elements can be of any data type, but at least one must be either a nonshared variable or a nonshared, noncustom event.

You cannot initialize any of the structure elements in the structure declaration. When you declare a variable to be of a structure type, you assign values to the elements by accessing them through the variable.

For a discussion of the differences between structures and classes, see [Structures and Classes](#).

For demonstration purposes, consider a situation where you want to keep track of an employee's name, telephone extension, and salary. A structure allows you to do this in a single variable.

To declare a structure

1. Create the beginning and ending statements for the structure.

You can specify the access level of a structure using the [Public](#), [Protected](#), [Friend](#), or [Private](#) keyword, or you can let it default to `Public`.

```
Private Structure employee  
End Structure
```

2. Add elements to the body of the structure.

A structure must have at least one element. You must declare every element and specify an access level for it. If you use the [Dim Statement](#) without any keywords, the accessibility defaults to `Public`.

```
Private Structure employee  
    Public givenName As String  
    Public familyName As String  
    Public phoneExtension As Long  
    Private salary As Decimal  
    Public Sub giveRaise(raise As Double)  
        salary *= raise  
    End Sub  
    Public Event salaryReviewTime()  
End Structure
```

The `salary` field in the preceding example is `Private`, which means it is inaccessible outside the structure, even from the containing class. However, the `giveRaise` procedure is `Public`, so it can be called from outside the structure. Similarly, you can raise the `salaryReviewTime` event from outside the structure.

In addition to variables, `Sub` procedures, and events, you can also define constants, `Function` procedures, and properties in a structure. You can designate at most one property as the *default property*, provided it takes at least one argument. You can handle an event with a [Shared](#) `Sub` procedure. For more information, see [How to: Declare and Call a Default Property in Visual Basic](#).

See also

- [Data Types](#)
- [Elementary Data Types](#)

- [Composite Data Types](#)
- [Value Types and Reference Types](#)
- [Structures](#)
- [Troubleshooting Data Types](#)
- [Structure Variables](#)
- [Structures and Other Programming Elements](#)
- [Structures and Classes](#)
- [User-Defined Data Type](#)

Structure Variables (Visual Basic)

7/30/2019 • 2 minutes to read • [Edit Online](#)

Once you have created a structure, you can declare procedure-level and module-level variables as that type. For example, you can create a structure that records information about a computer system. The following example demonstrates this.

```
Public Structure systemInfo
    Public CPU As String
    Public memory As Long
    Public purchaseDate As Date
End Structure
```

You can now declare variables of that type. The following declaration illustrates this.

```
Dim mySystem, yourSystem As systemInfo
```

NOTE

In classes and modules, structures declared using the [Dim Statement](#) default to public access. If you intend a structure to be private, make sure you declare it using the [Private keyword](#).

Access to Structure Values

To assign and retrieve values from the elements of a structure variable, you use the same syntax as you use to set and get properties on an object. You place the member access operator (`.`) between the structure variable name and the element name. The following example accesses elements of the variables previously declared as type `systemInfo`.

```
mySystem.CPU = "486"
Dim tooOld As Boolean
If yourSystem.purchaseDate < #1/1/1992# Then tooOld = True
```

Assigning Structure Variables

You can also assign one variable to another if both are of the same structure type. This copies all the elements of one structure to the corresponding elements in the other. The following declaration illustrates this.

```
yourSystem = mySystem
```

If a structure element is a reference type, such as a `String`, `Object`, or array, the pointer to the data is copied. In the previous example, if `systemInfo` had included an object variable, then the preceding example would have copied the pointer from `mySystem` to `yourSystem`, and a change to the object's data through one structure would be in effect when accessed through the other structure.

See also

- [Data Types](#)
- [Elementary Data Types](#)
- [Composite Data Types](#)
- [Value Types and Reference Types](#)
- [Structures](#)
- [Troubleshooting Data Types](#)
- [How to: Declare a Structure](#)
- [Structures and Other Programming Elements](#)
- [Structures and Classes](#)
- [Structure Statement](#)

Structures and Other Programming Elements (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

You can use structures in conjunction with arrays, objects, and procedures, as well as with each other. The interactions use the same syntax as these elements use individually.

NOTE

You cannot initialize any of the structure elements in the structure declaration. You can assign values only to elements of a variable that has been declared to be of a structure type.

Structures and Arrays

A structure can contain an array as one or more of its elements. The following example illustrates this.

```
Public Structure systemInfo
    Public cPU As String
    Public memory As Long
    Public diskDrives() As String
    Public purchaseDate As Date
End Structure
```

You access the values of an array within a structure the same way you access a property on an object. The following example illustrates this.

```
Dim mySystem As systemInfo
ReDim mySystem.diskDrives(3)
mySystem.diskDrives(0) = "1.44 MB"
```

You can also declare an array of structures. The following example illustrates this.

```
Dim allSystems(100) As systemInfo
```

You follow the same rules to access the components of this data architecture. The following example illustrates this.

```
ReDim allSystems(5).diskDrives(3)
allSystems(5).CPU = "386SX"
allSystems(5).diskDrives(2) = "100M SCSI"
```

Structures and Objects

A structure can contain an object as one or more of its elements. The following example illustrates this.

```

Protected Structure userInput
    Public userName As String
    Public inputForm As System.Windows.Forms.Form
    Public userFileNumber As Integer
End Structure

```

You should use a specific object class in such a declaration, rather than `Object`.

Structures and Procedures

You can pass a structure as a procedure argument. The following example illustrates this.

```

Public currentCPUName As String = "700MHz Pentium compatible"
Public currentMemorySize As Long = 256
Public Sub fillSystem(ByRef someSystem As systemInfo)
    someSystem.cPU = currentCPUName
    someSystem.memory = currentMemorySize
    someSystem.purchaseDate = Now
End Sub

```

The preceding example passes the structure *by reference*, which allows the procedure to modify its elements so that the changes take effect in the calling code. If you want to protect a structure against such modification, pass it by value.

You can also return a structure from a `Function` procedure. The following example illustrates this.

```

Dim allSystems(100) As systemInfo
Function findByDate(ByVal searchDate As Date) As systemInfo
    Dim i As Integer
    For i = 1 To 100
        If allSystems(i).purchaseDate = searchDate Then Return allSystems(i)
    Next i
    ' Process error: system with desired purchase date not found.
End Function

```

Structures Within Structures

Structures can contain other structures. The following example illustrates this.

```

Public Structure driveInfo
    Public type As String
    Public size As Long
End Structure
Public Structure systemInfo
    Public cPU As String
    Public memory As Long
    Public diskDrives() As driveInfo
    Public purchaseDate As Date
End Structure

```

```

Dim allSystems(100) As systemInfo
ReDim allSystems(1).diskDrives(3)
allSystems(1).diskDrives(0).type = "Floppy"

```

You can also use this technique to encapsulate a structure defined in one module within a structure defined in a different module.

Structures can contain other structures to an arbitrary depth.

See also

- [Data Types](#)
- [Elementary Data Types](#)
- [Composite Data Types](#)
- [Value Types and Reference Types](#)
- [Structures](#)
- [Troubleshooting Data Types](#)
- [How to: Declare a Structure](#)
- [Structure Variables](#)
- [Structures and Classes](#)
- [Structure Statement](#)

Structures and Classes (Visual Basic)

4/28/2019 • 4 minutes to read • [Edit Online](#)

Visual Basic unifies the syntax for structures and classes, with the result that both entities support most of the same features. However, there are also important differences between structures and classes.

Classes have the advantage of being reference types — passing a reference is more efficient than passing a structure variable with all its data. On the other hand, structures do not require allocation of memory on the global heap.

Because you cannot inherit from a structure, structures should be used only for objects that do not need to be extended. Use structures when the object you wish to create has a small instance size, and take into account the performance characteristics of classes versus structures.

Similarities

Structures and classes are similar in the following respects:

- Both are *container* types, meaning that they contain other types as members.
- Both have members, which can include constructors, methods, properties, fields, constants, enumerations, events, and event handlers. However, do not confuse these members with the declared *elements* of a structure.
- Members of both can have individualized access levels. For example, one member can be declared `Public` and another `Private`.
- Both can implement interfaces.
- Both can have shared constructors, with or without parameters.
- Both can expose a *default property*, provided that property takes at least one parameter.
- Both can declare and raise events, and both can declare delegates.

Differences

Structures and classes differ in the following particulars:

- Structures are *value types*; classes are *reference types*. A variable of a structure type contains the structure's data, rather than containing a reference to the data as a class type does.
- Structures use stack allocation; classes use heap allocation.
- All structure elements are `Public` by default; class variables and constants are `Private` by default, while other class members are `Public` by default. This behavior for class members provides compatibility with the Visual Basic 6.0 system of defaults.
- A structure must have at least one nonshared variable or nonshared, noncustom event element; a class can be completely empty.
- Structure elements cannot be declared as `Protected`; class members can.
- A structure procedure can handle events only if it is a `Shared Sub` procedure, and only by means of the `AddHandler Statement`; any class procedure can handle events, using either the `Handles` keyword or the

`AddHandler` statement. For more information, see [Events](#).

- Structure variable declarations cannot specify initializers or initial sizes for arrays; class variable declarations can.
- Structures implicitly inherit from the [System.ValueType](#) class and cannot inherit from any other type; classes can inherit from any class or classes other than [System.ValueType](#).
- Structures are not inheritable; classes are.
- Structures are never terminated, so the common language runtime (CLR) never calls the [Finalize](#) method on any structure; classes are terminated by the garbage collector (GC), which calls [Finalize](#) on a class when it detects there are no active references remaining.
- A structure does not require a constructor; a class does.
- Structures can have nonshared constructors only if they take parameters; classes can have them with or without parameters.

Every structure has an implicit public constructor without parameters. This constructor initializes all the structure's data elements to their default values. You cannot redefine this behavior.

Instances and Variables

Because structures are value types, each structure variable is permanently bound to an individual structure instance. But classes are reference types, and an object variable can refer to various class instances at different times. This distinction affects your usage of structures and classes in the following ways:

- **Initialization.** A structure variable implicitly includes an initialization of the elements using the structure's parameterless constructor. Therefore, `Dim s As struct1` is equivalent to `Dim s As struct1 = New struct1()`.
- **Assigning Variables.** When you assign one structure variable to another, or pass a structure instance to a procedure argument, the current values of all the variable elements are copied to the new structure. When you assign one object variable to another, or pass an object variable to a procedure, only the reference pointer is copied.
- **Assigning Nothing.** You can assign the value [Nothing](#) to a structure variable, but the instance continues to be associated with the variable. You can still call its methods and access its data elements, although variable elements are reinitialized by the assignment.

In contrast, if you set an object variable to `Nothing`, you dissociate it from any class instance, and you cannot access any members through the variable until you assign another instance to it.

- **Multiple Instances.** An object variable can have different class instances assigned to it at different times, and several object variables can refer to the same class instance at the same time. Changes you make to the values of class members affect those members when accessed through another variable pointing to the same instance.

Structure elements, however, are isolated within their own instance. Changes to their values are not reflected in any other structure variables, even in other instances of the same `Structure` declaration.

- **Equality.** Equality testing of two structures must be performed with an element-by-element test. Two object variables can be compared using the [Equals](#) method. [Equals](#) indicates whether the two variables point to the same instance.

See also

- [Data Types](#)

- Composite Data Types
- Value Types and Reference Types
- Structures
- Troubleshooting Data Types
- Structures and Other Programming Elements
- Objects and Classes

Tuples (Visual Basic)

10/17/2019 • 11 minutes to read • [Edit Online](#)

Starting with Visual Basic 2017, the Visual Basic language offers built-in support for tuples that makes creating tuples and accessing the elements of tuples easier. A tuple is a lightweight data structure that has a specific number and sequence of values. When you instantiate the tuple, you define the number and the data type of each value (or element). For example, a 2-tuple (or pair) has two elements. The first might be a `Boolean` value, while the second is a `String`. Because tuples make it easy to store multiple values in a single object, they are often used as a lightweight way to return multiple values from a method.

IMPORTANT

Tuple support requires the `ValueTuple` type. If the .NET Framework 4.7 is not installed, you must add the NuGet package `System.ValueTuple`, which is available on the NuGet Gallery. Without this package, you may get a compilation error similar to, "Predefined type 'ValueTuple(Of,,)' is not defined or imported."

Instantiating and using a tuple

You instantiate a tuple by enclosing its comma-delimited values in parentheses. Each of those values then becomes a field of the tuple. For example, the following code defines a triple (or 3-tuple) with a `Date` as its first value, a `String` as its second, and a `Boolean` as its third.

```
Dim holiday = (#07/04/2017#, "Independence Day", True)
```

By default, the name of each field in a tuple consists of the string `Item` along with the field's one-based position in the tuple. For this 3-tuple, the `Date` field is `Item1`, the `String` field is `Item2`, and the `Boolean` field is `Item3`. The following example displays the values of fields of the tuple instantiated in the previous line of code

```
Console.WriteLine($"{holiday.Item1} is {holiday.Item2}" +
    $"{If(holiday.Item3, ", a national holiday", String.Empty)}")
' Output: 7/4/2017 12:00:00 AM Is Independence Day, a national holiday
```

The fields of a Visual Basic tuple are read-write; after you've instantiated a tuple, you can modify its values. The following example modifies two of the three fields of the tuple created in the previous example and displays the result.

```
holiday.Item1 = #01/01/2018#
holiday.Item2 = "New Year's Day"
Console.WriteLine($"{holiday.Item1} is {holiday.Item2}" +
    $"{If(holiday.Item3, ", a national holiday", String.Empty)}")
' Output: 1/1/2018 12:00:00 AM Is New Year's Day, a national holiday
```

Instantiating and using a named tuple

Rather than using default names for a tuple's fields, you can instantiate a *named tuple* by assigning your own names to the tuple's elements. The tuple's fields can then be accessed by their assigned names or by their default names. The following example instantiates the same 3-tuple as previously, except that it explicitly names the first field `EventDate`, the second `Name`, and the third `IsHoliday`. It then displays the field values, modifies them, and

displays the field values again.

```
Dim holiday = (EventDate:=#07/04/2017#, Name:="Independence Day", IsHoliday:=True)
Console.WriteLine($"{holiday.EventDate} Is {holiday.Name}" +
    $"{If(holiday.IsHoliday, ", a national holiday", String.Empty)}")
holiday.Item1 = #01/01/2018#
holiday.Item2 = "New Year's Day"
Console.WriteLine($"{holiday.Item1} is {holiday.Item2}" +
    $"{If(holiday.Item3, ", a national holiday", String.Empty)})")
' The example displays the following output:
'   7/4/2017 12:00:00 AM Is Independence Day, a national holiday
'   1/1/2018 12:00:00 AM Is New Year's Day, a national holiday
```

Inferred tuple element names

Starting with Visual Basic 15.3, Visual Basic can infer the names of tuple elements; you do not have to assign them explicitly. Inferred tuple names are useful when you initialize a tuple from a set of variables, and you want the tuple element name to be the same as the variable name.

The following example creates a `stateInfo` tuple that contains three explicitly named elements, `state`, `stateName`, and `capital`. Note that, in naming the elements, the tuple initialization statement simply assigns the named elements the values of the identically named variables.

```
Const state As String = "MI"
Const stateName As String = "Michigan"
Const capital As String = "Lansing"
Dim stateInfo = (state:=state, stateName:=stateName, capital:=capital)
Console.WriteLine($"{stateInfo.stateName}: 2-letter code: {stateInfo.state}, Capital {stateInfo.capital}")
' The example displays the following output:
'   Michigan: 2-letter code: MI, Capital Lansing
```

Because elements and variables have the same name, the Visual Basic compiler can infer the names of the fields, as the following example shows.

```
Const state As String = "MI"
Const stateName As String = "Michigan"
Const capital As String = "Lansing"
Dim stateInfo = ( state, stateName, capital )
Console.WriteLine($"{stateInfo.stateName}: 2-letter code: {stateInfo.State}, Capital {stateInfo.capital}")
' The example displays the following output:
'   Michigan: 2-letter code: MI, Capital Lansing
```

To enable inferred tuple element names, you must define the version of the Visual Basic compiler to use in your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.3</LangVersion>
</PropertyGroup>
```

The version number can be any version of the Visual Basic compiler starting with 15.3. Rather than hard-coding a specific compiler version, you can also specify "Latest" as the value of `LangVersion` to compile with the most recent version of the Visual Basic compiler installed on your system.

For more information, see [setting the Visual Basic language version](#).

In some cases, the Visual Basic compiler cannot infer the tuple element name from the candidate name, and the tuple field can only be referenced using its default name, such as `Item1`, `Item2`, etc. These include:

- The candidate name is the same as the name of a tuple member, such as `Item3`, `Rest`, or `ToString`.
- The candidate name is duplicated in the tuple.

When field name inference fails, Visual Basic does not generate a compiler error, nor is an exception thrown at runtime. Instead, tuple fields must be referenced by their predefined names, such as `Item1` and `Item2`.

Tuples versus structures

A Visual Basic tuple is a value type that is an instance of one of the **System.ValueTuple** generic types. For example, the `holiday` tuple defined in the previous example is an instance of the `ValueTuple<T1,T2,T3>` structure. It is designed to be a lightweight container for data. Since the tuple aims to make it easy to create an object with multiple data items, it lacks some of the features that a custom structure might have. These include:

- Custom members. You cannot define your own properties, methods, or events for a tuple.
- Validation. You cannot validate the data assigned to fields.
- Immutability. Visual Basic tuples are mutable. In contrast, a custom structure allows you to control whether an instance is mutable or immutable.

If custom members, property and field validation, or immutability are important, you should use the Visual Basic [Structure](#) statement to define a custom value type.

A Visual Basic tuple does inherit the members of its **ValueTuple** type. In addition to its fields, these include the following methods:

MEMBER	DESCRIPTION
<code>CompareTo</code>	Compares the current tuple to another tuple with the same number of elements.
<code>Equals</code>	Determines whether the current tuple is equal to another tuple or object.
<code>GetHashCode</code>	Calculates the hash code for the current instance.
<code>ToString</code>	Returns the string representation of this tuple, which takes the form <code>(Item1, Item2...)</code> , where <code>Item1</code> and <code>Item2</code> represent the values of the tuple's fields.

In addition, the **ValueTuple** types implement [IStructuralComparable](#) and [IStructuralEquatable](#) interfaces, which allow you to define customer comparers.

Assignment and tuples

Visual Basic supports assignment between tuple types that have the same number of fields. The field types can be converted if one of the following is true:

- The source and target field are of the same type.
- A widening (or implicit) conversion of the source type to the target type is defined.
- `Option Strict` is `on`, and a narrowing (or explicit) conversion of the source type to the target type is defined. This conversion can throw an exception if the source value is outside the range of the target type.

Other conversions are not considered for assignments. Let's look at the kinds of assignments that are allowed between tuple types.

Consider these variables used in the following examples:

```
' The number and field types of all these tuples are compatible.  
' The only difference is the field names being used.  
Dim unnamed = (42, "The meaning of life")  
Dim anonymous = (16, "a perfect square")  
Dim named = (Answer:=42, Message:="The meaning of life")  
Dim differentNamed = (SecretConstant:=42, Label:="The meaning of life")
```

The first two variables, `unnamed` and `anonymous`, do not have semantic names provided for the fields. Their field names are the default `Item1` and `Item2`. The last two variables, `named` and `differentNamed` have semantic field names. Note that these two tuples have different names for the fields.

All four of these tuples have the same number of fields (referred to as 'arity'), and the types of those fields are identical. Therefore, all of these assignments work:

```
' Assign named to unnamed.  
named = unnamed  
  
' Despite the assignment, named still has fields that can be referred to as 'answer' and 'message'.  
Console.WriteLine($"{named.Answer}, {named.Message}")  
' Output: 42, The meaning of life  
  
' Assign unnamed to anonymous.  
anonymous = unnamed  
' Because of the assignment, the value of the elements of anonymous changed.  
Console.WriteLine($"{anonymous.Item1}, {anonymous.Item2}")  
' Output: 42, The meaning of life  
  
' Assign one named tuple to the other.  
named = differentNamed  
' The field names are not assigned. 'named' still has 'answer' and 'message' fields.  
Console.WriteLine($"{named.Answer}, {named.Message}")  
' Output: 42, The meaning of life
```

Notice that the names of the tuples are not assigned. The values of the fields are assigned following the order of the fields in the tuple.

Finally, notice that we can assign the `named` tuple to the `conversion` tuple, even though the first field of `named` is an `Integer`, and the first field of `conversion` is a `Long`. This assignment succeeds because converting an `Integer` to a `Long` is a widening conversion.

```
' Assign an (Integer, String) tuple to a (Long, String) tuple (using implicit conversion).  
Dim conversion As (Long, String) = named  
Console.WriteLine($"{conversion.Item1} ({conversion.Item1.GetType().Name}), " +  
    $"{conversion.Item2} ({conversion.Item2.GetType().Name})")  
' Output: 42 (Int64), The meaning of life (String)
```

Tuples with different numbers of fields are not assignable:

```
' Does not compile.  
' VB30311: Value of type '(Integer, Integer, Integer)' cannot be converted  
'           to '(Answer As Integer, Message As String)'  
var differentShape = (1, 2, 3)  
named = differentShape
```

Tuples as method return values

A method can return only a single value. Frequently, though, you'd like a method call to return multiple values. There are several ways to work around this limitation:

- You can create a custom class or structure whose properties or fields represent values returned by the method. This is a heavyweight solution; it requires that you define a custom type whose only purpose is to retrieve values from a method call.
- You can return a single value from the method, and return the remaining values by passing them by reference to the method. This involves the overhead of instantiating a variable and risks inadvertently overwriting the value of the variable that you pass by reference.
- You can use a tuple, which provides a lightweight solution to retrieving multiple return values.

For example, the **TryParse** methods in .NET return a `Boolean` value that indicates whether the parsing operation succeeded. The result of the parsing operation is returned in a variable passed by reference to the method.

Normally, a call to the a parsing method such as `Int32.TryParse` looks like the following:

```
Dim numericString As String = "123456"
Dim number As Integer
Dim result = Int32.TryParse(numericString, number)
Console.WriteLine($"{If(result, $"Success: {number:N0}", "Failure"))")
'      Output: 123,456
```

We can return a tuple from the parsing operation if we wrap the call to the `Int32.TryParse` method in our own method. In the following example, `NumericLibrary.ParseInteger` calls the `Int32.TryParse` method and returns a named tuple with two elements.

```
Imports System.Globalization

Public Module NumericLibrary
    Public Function ParseInteger(value As String) As (Success As Boolean, Number As Int32)
        Dim number As Integer
        Return (Int32.TryParse(value, NumberStyles.Any, CultureInfo.InvariantCulture, number), number)
    End Function
End Module
```

You can then call the method with code like the following:

```
Dim numericString As String = "123,456"
Dim result = ParseInteger(numericString)
Console.WriteLine($"{If(result.Success, $"Success: {result.Number:N0}", "Failure"))")
Console.ReadLine()
'      Output: Success: 123,456
```

Visual Basic tuples and tuples in the .NET Framework

A Visual Basic tuple is an instance of one of the **System.ValueTuple** generic types, which were introduced in the .NET Framework 4.7. The .NET Framework also includes a set of generic **System.Tuple** classes. These classes, however, differ from Visual Basic tuples and the **System.ValueTuple** generic types in a number of ways:

- The elements of the **Tuple** classes are properties named `Item1`, `Item2`, and so on. In Visual Basic tuples and the **ValueTuple** types, tuple elements are fields.
- You cannot assign meaningful names to the elements of a **Tuple** instance or of a **ValueTuple** instance. Visual Basic allows you to assign names that communicate the meaning of the fields.
- The properties of a **Tuple** instance are read-only; the tuples are immutable. In Visual Basic tuples and the

ValueTuple types, tuple fields are read-write; the tuples are mutable.

- The generic **Tuple** types are reference types. Using these **Tuple** types means allocating objects. On hot paths, this can have a measurable impact on your application's performance. Visual Basic tuples and the **ValueTuple** types are value types.

Extension methods in the [TupleExtensions](#) class make it easy to convert between Visual Basic tuples and .NET **Tuple** objects. The **ToTuple** method converts a Visual Basic tuple to a .NET **Tuple** object, and the **ToValueTuple** method converts a .NET **Tuple** object to a Visual Basic tuple.

The following example creates a tuple, converts it to a .NET **Tuple** object, and converts it back to a Visual Basic tuple. The example then compares this tuple with the original one to ensure that they are equal.

```
Module Example
    Sub Main()
        Dim cityInfo = (name:="New York", area:=468.5, population:=8_550_405)
        Console.WriteLine($"{cityInfo}, type {cityInfo.GetType().Name}")

        ' Convert the Visual Basic tuple to a .NET tuple.
        Dim cityInfoT = TupleExtensions.ToTuple(cityInfo)
        Console.WriteLine($"{cityInfoT}, type {cityInfoT.GetType().Name}")

        ' Convert the .NET tuple back to a Visual Basic tuple and ensure they are the same.
        Dim cityInfo2 = TupleExtensions.ToValueTuple(cityInfoT)
        Console.WriteLine($"{cityInfo2}, type {cityInfo2.GetType().Name}")
        Console.WriteLine($"{NameOf(cityInfo)} = {NameOf(cityInfo2)}: {cityInfo.Equals(cityInfo2)}")
        Console.ReadLine()
    End Sub
End Module
' The example displays the following output:
' (New York, 468.5, 8550405), type ValueTuple`3
' (New York, 468.5, 8550405), type Tuple`3
' (New York, 468.5, 8550405), type ValueTuple`3
' cityInfo = cityInfo2 : True
```

See also

- [Visual Basic Language Reference](#)

Efficient Use of Data Types (Visual Basic)

7/30/2019 • 2 minutes to read • [Edit Online](#)

Undeclared variables and variables declared without a data type are assigned the `Object` data type. This makes it easy to write programs quickly, but it can cause them to execute more slowly.

Strong Typing

Specifying data types for all your variables is known as *strong typing*. Using strong typing has several advantages:

- It enables IntelliSense support for your variables. This allows you to see their properties and other members as you type in the code.
- It takes advantage of compiler type checking. This catches statements that can fail at run time due to errors such as overflow. It also catches calls to methods on objects that do not support them.
- It results in faster execution of your code.

Most Efficient Data Types

For variables that never contain fractions, the integral data types are more efficient than the nonintegral types. In Visual Basic, `Integer` and `UInteger` are the most efficient numeric types.

For fractional numbers, `Double` is the most efficient data type, because the processors on current platforms perform floating-point operations in double precision. However, operations with `Double` are not as fast as with the integral types such as `Integer`.

Specifying Data Type

Use the [Dim Statement](#) to declare a variable of a specific type. You can simultaneously specify its access level by using the [Public](#), [Protected](#), [Friend](#), or [Private](#) keyword, as in the following example.

```
Private x As Double  
Protected s As String
```

Character Conversion

The `AscW` and `ChrW` functions operate in Unicode. You should use them in preference to `Asc` and `Chr`, which must translate into and out of Unicode.

See also

- [Asc](#)
- [AscW](#)
- [Chr](#)
- [ChrW](#)
- [Data Types](#)
- [Numeric Data Types](#)

- [Variable Declaration](#)
- [Using IntelliSense](#)

Troubleshooting Data Types (Visual Basic)

10/18/2019 • 7 minutes to read • [Edit Online](#)

This page lists some common problems that can occur when you perform operations on intrinsic data types.

Floating-Point Expressions Do Not Compare as Equal

When you work with floating-point numbers ([Single Data Type](#) and [Double Data Type](#)), remember that they are stored as binary fractions. This means they cannot hold an exact representation of any quantity that is not a binary fraction (of the form $k / (2^n)$ where k and n are integers). For example, 0.5 (= 1/2) and 0.3125 (= 5/16) can be held as precise values, whereas 0.2 (= 1/5) and 0.3 (= 3/10) can be only approximations.

Because of this imprecision, you cannot rely on exact results when you operate on floating-point values. In particular, two values that are theoretically equal might have slightly different representations.

TO COMPARE FLOATING-POINT QUANTITIES

1. Calculate the absolute value of their difference by using the [Abs](#) method of the [Math](#) class in the [System](#) namespace.
2. Determine an acceptable maximum difference, such that you can consider the two quantities to be equal for practical purposes if their difference is no larger.
3. Compare the absolute value of the difference to the acceptable difference.

The following example demonstrates both incorrect and correct comparison of two `Double` values.

```
Dim oneThird As Double = 1.0 / 3.0
Dim pointThrees As Double = 0.333333333333333

' The following comparison does not indicate equality.
Dim exactlyEqual As Boolean = (oneThird = pointThrees)

' The following comparison indicates equality.
Dim closeEnough As Double = 0.00000000000001
Dim absoluteDifference As Double = Math.Abs(oneThird - pointThrees)
Dim practicallyEqual As Boolean = (absoluteDifference < closeEnough)

MsgBox("1.0 / 3.0 is represented as " & oneThird.ToString("G17") &
      vbCrLf & "0.333333333333333 is represented as " &
      pointThrees.ToString("G17") &
      vbCrLf & "Exact comparison generates " & CStr(exactlyEqual) &
      vbCrLf & "Acceptable difference comparison generates " &
      CStr(practicallyEqual))
```

The previous example uses the [ToString](#) method of the `Double` structure so that it can specify better precision than the `cstr` keyword uses. The default is 15 digits, but the "G17" format extends it to 17 digits.

Mod Operator Does Not Return Accurate Result

Because of the imprecision of floating-point storage, the [Mod Operator](#) can return an unexpected result when at least one of the operands is floating-point.

The [Decimal Data Type](#) does not use floating-point representation. Many numbers that are inexact in `Single` and `Double` are exact in `Decimal` (for example 0.2 and 0.3). Although arithmetic is slower in `Decimal` than in floating-point, it might be worth the performance decrease to achieve better precision.

TO FIND THE INTEGER REMAINDER OF FLOATING-POINT QUANTITIES

1. Declare variables as `Decimal`.
2. Use the literal type character `D` to force literals to `Decimal`, in case their values are too large for the `Long` data type.

The following example demonstrates the potential imprecision of floating-point operands.

```
Dim two As Double = 2.0
Dim zeroPointTwo As Double = 0.2
Dim quotient As Double = two / zeroPointTwo
Dim doubleRemainder As Double = two Mod zeroPointTwo

MsgBox("2.0 is represented as " & two.ToString("G17") &
vbCrLf & "0.2 is represented as " & zeroPointTwo.ToString("G17") &
vbCrLf & "2.0 / 0.2 generates " & quotient.ToString("G17") &
vbCrLf & "2.0 Mod 0.2 generates " &
doubleRemainder.ToString("G17"))

Dim decimalRemainder As Decimal = 2D Mod 0.2D
MsgBox("2.0D Mod 0.2D generates " & CStr(decimalRemainder))
```

The previous example uses the `ToString` method of the `Double` structure so that it can specify better precision than the `cstr` keyword uses. The default is 15 digits, but the "G17" format extends it to 17 digits.

Because `zeroPointTwo` is `Double`, its value for 0.2 is an infinitely repeating binary fraction with a stored value of 0.2000000000000001. Dividing 2.0 by this quantity yields 9.999999999999995 with a remainder of 0.1999999999999991.

In the expression for `decimalRemainder`, the literal type character `D` forces both operands to `Decimal`, and 0.2 has a precise representation. Therefore the `Mod` operator yields the expected remainder of 0.0.

Note that it is not sufficient to declare `decimalRemainder` as `Decimal`. You must also force the literals to `Decimal`, or they use `Double` by default and `decimalRemainder` receives the same inaccurate value as `doubleRemainder`.

Boolean Type Does Not Convert to Numeric Type Accurately

[Boolean Data Type](#) values are not stored as numbers, and the stored values are not intended to be equivalent to numbers. For compatibility with earlier versions, Visual Basic provides conversion keywords ([CType Function](#), `CBool`, `CInt`, and so on) to convert between `Boolean` and numeric types. However, other languages sometimes perform these conversions differently, as do the .NET Framework methods.

You should never write code that relies on equivalent numeric values for `True` and `False`. Whenever possible, you should restrict usage of `Boolean` variables to the logical values for which they are designed. If you must mix `Boolean` and numeric values, make sure that you understand the conversion method that you select.

Conversion in Visual Basic

When you use the `CType` or `CBool` conversion keywords to convert numeric data types to `Boolean`, 0 becomes `False` and all other values become `True`. When you convert `Boolean` values to numeric types by using the conversion keywords, `False` becomes 0 and `True` becomes -1.

Conversion in the Framework

The `ToInt32` method of the `Convert` class in the `System` namespace converts `True` to +1.

If you must convert a `Boolean` value to a numeric data type, be careful about which conversion method you

use.

Character Literal Generates Compiler Error

In the absence of any type characters, Visual Basic assumes default data types for literals. The default type for a character literal — enclosed in quotation marks (" ") — is `String`.

The `String` data type does not widen to the [Char Data Type](#). This means that if you want to assign a literal to a `Char` variable, you must either make a narrowing conversion or force the literal to the `Char` type.

TO CREATE A CHAR LITERAL TO ASSIGN TO A VARIABLE OR CONSTANT

1. Declare the variable or constant as `Char`.
2. Enclose the character value in quotation marks (" ").
3. Follow the closing double quotation mark with the literal type character `C` to force the literal to `char`. This is necessary if the type checking switch ([Option Strict Statement](#)) is `On`, and it is desirable in any case.

The following example demonstrates both unsuccessful and successful assignments of a literal to a `Char` variable.

```
Dim charVar As Char
' The following statement attempts to convert a String literal to Char.
' Because Option Strict is On, it generates a compiler error.
charVar = "Z"
' The following statement succeeds because it specifies a Char literal.
charVar = "Z"c
' The following statement succeeds because it converts String to Char.
charVar = CChar("Z")
```

There is always a risk in using narrowing conversions, because they can fail at run time. For example, a conversion from `String` to `Char` can fail if the `String` value contains more than one character. Therefore, it is better programming to use the `c` type character.

String Conversion Fails at Run Time

The [String Data Type](#) participates in very few widening conversions. `String` widens only to itself and `Object`, and only `Char` and `Char()` (a `char` array) widen to `String`. This is because `String` variables and constants can contain values that other data types cannot contain.

When the type checking switch ([Option Strict Statement](#)) is `On`, the compiler disallows all implicit narrowing conversions. This includes those involving `String`. Your code can still use conversion keywords such as `CStr` and [CType Function](#), which direct the .NET Framework to attempt the conversion.

NOTE

The narrowing-conversion error is suppressed for conversions from the elements in a `For Each...Next` collection to the loop control variable. For more information and examples, see the "Narrowing Conversions" section in [For Each...Next Statement](#).

Narrowing Conversion Protection

The disadvantage of narrowing conversions is that they can fail at run time. For example, if a `String` variable contains anything other than "True" or "False," it cannot be converted to `Boolean`. If it contains punctuation characters, conversion to any numeric type fails. Unless you know that your `String` variable always holds values that the destination type can accept, you should not try a conversion.

If you must convert from `String` to another data type, the safest procedure is to enclose the attempted conversion in the [Try...Catch...Finally Statement](#). This lets you deal with a run-time failure.

Character Arrays

A single `char` and an array of `char` elements both widen to `String`. However, `String` does not widen to `char()`. To convert a `String` value to a `char` array, you can use the [ToCharArray](#) method of the `System.String` class.

Meaningless Values

In general, `String` values are not meaningful in other data types, and conversion is highly artificial and dangerous. Whenever possible, you should restrict usage of `String` variables to the character sequences for which they are designed. You should never write code that relies on equivalent values in other types.

See also

- [Data Types](#)
- [Type Characters](#)
- [Value Types and Reference Types](#)
- [Type Conversions in Visual Basic](#)
- [Data Types](#)
- [Type Conversion Functions](#)
- [Efficient Use of Data Types](#)

Declared Elements in Visual Basic

4/28/2019 • 2 minutes to read • [Edit Online](#)

A *declared element* is a programming element that is defined in a declaration statement. Declared elements include variables, constants, enumerations, classes, structures, modules, interfaces, procedures, procedure parameters, function returns, external procedure references, operators, properties, events, and delegates.

Declaration statements include the following:

- [Dim Statement](#)
- [Const Statement](#)
- [Enum Statement](#)
- [Class Statement](#)
- [Structure Statement](#)
- [Module Statement](#)
- [Interface Statement](#)
- [Function Statement](#)
- [Sub Statement](#)
- [Declare Statement](#)
- [Operator Statement](#)
- [Property Statement](#)
- [Event Statement](#)
- [Delegate Statement](#)

In This Section

[Declared Element Names](#)

Describes how to name elements and use alphabetic case.

[Declared Element Characteristics](#)

Covers characteristics, such as scope, possessed by declared elements.

[References to Declared Elements](#)

Describes how the compiler matches a reference to a declaration and how to qualify a name.

Related Sections

[Program Structure and Code Conventions](#)

Presents guidelines for making your code easier to read, understand, and maintain.

[Statements](#)

Describes statements that name and define procedures, variables, arrays, and constants.

[Declaration Contexts and Default Access Levels](#)

Lists the types of declared elements and shows for each one its declaration statement, in what context you can declare it, and its default access level.

Delegates (Visual Basic)

3/8/2019 • 5 minutes to read • [Edit Online](#)

Delegates are objects that refer to methods. They are sometimes described as *type-safe function pointers* because they are similar to function pointers used in other programming languages. But unlike function pointers, Visual Basic delegates are a reference type based on the class [System.Delegate](#). Delegates can reference both shared methods — methods that can be called without a specific instance of a class — and instance methods.

Delegates and Events

Delegates are useful in situations where you need an intermediary between a calling procedure and the procedure being called. For example, you might want an object that raises events to be able to call different event handlers under different circumstances. Unfortunately, the object raising the events cannot know ahead of time which event handler is handling a specific event. Visual Basic lets you dynamically associate event handlers with events by creating a delegate for you when you use the `AddHandler` statement. At run time, the delegate forwards calls to the appropriate event handler.

Although you can create your own delegates, in most cases Visual Basic creates the delegate and takes care of the details for you. For example, an `Event` statement implicitly defines a delegate class named `<EventName>EventHandler` as a nested class of the class containing the `Event` statement, and with the same signature as the event. The `AddressOf` statement implicitly creates an instance of a delegate that refers to a specific procedure. The following two lines of code are equivalent. In the first line, you see the explicit creation of an instance of `EventHandler`, with a reference to method `Button1_Click` sent as the argument. The second line is a more convenient way to do the same thing.

```
AddHandler Button1.Click, New EventHandler(AddressOf Button1_Click)
' The following line of code is shorthand for the previous line.
AddHandler Button1.Click, AddressOf Me.Button1_Click
```

You can use the shorthand way of creating delegates anywhere the compiler can determine the delegate's type by the context.

Declaring Events that Use an Existing Delegate Type

In some situations, you may want to declare an event to use an existing delegate type as its underlying delegate. The following syntax demonstrates how:

```
Delegate Sub DelegateType()
Event AnEvent As DelegateType
```

This is useful when you want to route multiple events to the same handler.

Delegate Variables and Parameters

You can use delegates for other, non-event related tasks, such as free threading or with procedures that need to call different versions of functions at run time.

For example, suppose you have a classified-ad application that includes a list box with the names of cars. The ads are sorted by title, which is normally the make of the car. A problem you may face occurs when some cars include the year of the car before the make. The problem is that the built-in sort functionality of the list box sorts only by

character codes; it places all the ads starting with dates first, followed by the ads starting with the make.

To fix this, you can create a sort procedure in a class that uses the standard alphabetic sort on most list boxes, but is able to switch at run time to the custom sort procedure for car ads. To do this, you pass the custom sort procedure to the sort class at run time, using delegates.

AddressOf and Lambda Expressions

Each delegate class defines a constructor that is passed the specification of an object method. An argument to a delegate constructor must be a reference to a method, or a lambda expression.

To specify a reference to a method, use the following syntax:

```
AddressOf [ expression ] methodName
```

The compile-time type of the `expression` must be the name of a class or an interface that contains a method of the specified name whose signature matches the signature of the delegate class. The `methodName` can be either a shared method or an instance method. The `methodName` is not optional, even if you create a delegate for the default method of the class.

To specify a lambda expression, use the following syntax:

```
Function ([ parm As type , parm2 As type2 , ...]) expression
```

The following example shows both `AddressOf` and lambda expressions used to specify the reference for a delegate.

```

Module Module1

Sub Main()
    ' Create an instance of InOrderClass and assign values to the properties.
    ' InOrderClass method ShowInOrder displays the numbers in ascending
    ' or descending order, depending on the comparison method you specify.
    Dim inOrder As New InOrderClass
    inOrder.Num1 = 5
    inOrder.Num2 = 4

    ' Use AddressOf to send a reference to the comparison function you want
    ' to use.
    inOrder.ShowInOrder(AddressOf GreaterThan)
    inOrder.ShowInOrder(AddressOf LessThan)

    ' Use lambda expressions to do the same thing.
    inOrder.ShowInOrder(Function(m, n) m > n)
    inOrder.ShowInOrder(Function(m, n) m < n)
End Sub

Function GreaterThan(ByVal num1 As Integer, ByVal num2 As Integer) As Boolean
    Return num1 > num2
End Function

Function LessThan(ByVal num1 As Integer, ByVal num2 As Integer) As Boolean
    Return num1 < num2
End Function

Class InOrderClass
    ' Define the delegate function for the comparisons.
    Delegate Function CompareNumbers(ByVal num1 As Integer, ByVal num2 As Integer) As Boolean
    ' Display properties in ascending or descending order.
    Sub ShowInOrder(ByVal compare As CompareNumbers)
        If compare(_num1, _num2) Then
            Console.WriteLine(_num1 & " " & _num2)
        Else
            Console.WriteLine(_num2 & " " & _num1)
        End If
    End Sub

    Private _num1 As Integer
    Property Num1() As Integer
        Get
            Return _num1
        End Get
        Set(ByVal value As Integer)
            _num1 = value
        End Set
    End Property

    Private _num2 As Integer
    Property Num2() As Integer
        Get
            Return _num2
        End Get
        Set(ByVal value As Integer)
            _num2 = value
        End Set
    End Property
End Class
End Module

```

The signature of the function must match that of the delegate type. For more information about lambda expressions, see [Lambda Expressions](#). For more examples of lambda expression and `AddressOf` assignments to delegates, see [Relaxed Delegate Conversion](#).

Related Topics

TITLE	DESCRIPTION
How to: Invoke a Delegate Method	Provides an example that shows how to associate a method with a delegate and then invoke that method through the delegate.
How to: Pass Procedures to Another Procedure in Visual Basic	Demonstrates how to use delegates to pass one procedure to another procedure.
Relaxed Delegate Conversion	Describes how you can assign subs and functions to delegates or handlers even when their signatures are not identical
Events	Provides an overview of events in Visual Basic.

Early and Late Binding (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

The Visual Basic compiler performs a process called `binding` when an object is assigned to an object variable. An object is *early bound* when it is assigned to a variable declared to be of a specific object type. Early bound objects allow the compiler to allocate memory and perform other optimizations before an application executes. For example, the following code fragment declares a variable to be of type `FileStream`:

```
' Create a variable to hold a new object.  
Dim FS As System.IO.FileStream  
' Assign a new object to the variable.  
FS = New System.IO.FileStream("C:\tmp.txt",  
    System.IO.FileMode.Open)
```

Because `FileStream` is a specific object type, the instance assigned to `FS` is early bound.

By contrast, an object is *late bound* when it is assigned to a variable declared to be of type `Object`. Objects of this type can hold references to any object, but lack many of the advantages of early-bound objects. For example, the following code fragment declares an object variable to hold an object returned by the `CreateObject` function:

```
' To use this example, you must have Microsoft Excel installed on your computer.  
' Compile with Option Strict Off to allow late binding.  
Sub TestLateBinding()  
    Dim xlApp As Object  
    Dim xlBook As Object  
    Dim xlSheet As Object  
    xlApp = CreateObject("Excel.Application")  
    ' Late bind an instance of an Excel workbook.  
    xlBook = xlApp.Workbooks.Add  
    ' Late bind an instance of an Excel worksheet.  
    xlSheet = xlBook.Worksheets(1)  
    xlSheet.Activate()  
    ' Show the application.  
    xlSheet.Application.Visible = True  
    ' Place some text in the second row of the sheet.  
    xlSheet.Cells(2, 2) = "This is column B row 2"  
End Sub
```

Advantages of Early Binding

You should use early-bound objects whenever possible, because they allow the compiler to make important optimizations that yield more efficient applications. Early-bound objects are significantly faster than late-bound objects and make your code easier to read and maintain by stating exactly what kind of objects are being used. Another advantage to early binding is that it enables useful features such as automatic code completion and Dynamic Help because the Visual Studio integrated development environment (IDE) can determine exactly what type of object you are working with as you edit the code. Early binding reduces the number and severity of run-time errors because it allows the compiler to report errors when a program is compiled.

NOTE

Late binding can only be used to access type members that are declared as `Public`. Accessing members declared as `Friend` or `Protected Friend` results in a run-time error.

See also

- [CreateObject](#)
- [Object Lifetime: How Objects Are Created and Destroyed](#)
- [Object Data Type](#)

Error Types (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

In Visual Basic, errors fall into one of three categories: syntax errors, run-time errors, and logic errors.

Syntax Errors

Syntax errors are those that appear while you write code. If you're using Visual Studio, Visual Basic checks your code as you type it in the **Code Editor** window and alerts you if you make a mistake, such as misspelling a word or using a language element improperly. If you compile from the command line, Visual Basic displays a compiler error with information about the syntax error. Syntax errors are the most common type of errors. You can fix them easily in the coding environment as soon as they occur.

NOTE

The `Option Explicit` statement is one means of avoiding syntax errors. It forces you to declare, in advance, all the variables to be used in the application. Therefore, when those variables are used in the code, any typographic errors are caught immediately and can be fixed.

Run-Time Errors

Run-time errors are those that appear only after you compile and run your code. These involve code that may appear to be correct in that it has no syntax errors, but that will not execute. For example, you might correctly write a line of code to open a file. But if the file does not exist, the application cannot open the file, and it throws an exception. You can fix most run-time errors by rewriting the faulty code or by using [exception handling](#), and then recompiling and rerunning it.

Logic Errors

Logic errors are those that appear once the application is in use. They are most often faulty assumptions made by the developer, or unwanted or unexpected results in response to user actions. For example, a mistyped key might provide incorrect information to a method, or you may assume that a valid value is always supplied to a method when that is not the case. Although logic errors can be handled by using [exception handling](#) (for example, by testing whether an argument is `Nothing` and throwing an `ArgumentNullException`), most commonly they should be addressed by correcting the error in logic and recompiling the application.

See also

- [Try...Catch...Finally Statement](#)
- [Debugger Basics](#)

Events (Visual Basic)

8/22/2019 • 6 minutes to read • [Edit Online](#)

While you might visualize a Visual Studio project as a series of procedures that execute in a sequence, in reality, most programs are event driven—meaning the flow of execution is determined by external occurrences called *events*.

An event is a signal that informs an application that something important has occurred. For example, when a user clicks a control on a form, the form can raise a `Click` event and call a procedure that handles the event. Events also allow separate tasks to communicate. Say, for example, that your application performs a sort task separately from the main application. If a user cancels the sort, your application can send a cancel event instructing the sort process to stop.

Event Terms and Concepts

This section describes the terms and concepts used with events in Visual Basic.

Declaring Events

You declare events within classes, structures, modules, and interfaces using the `Event` keyword, as in the following example:

```
Event AnEvent(ByVal EventNumber As Integer)
```

Raising Events

An event is like a message announcing that something important has occurred. The act of broadcasting the message is called *raising* the event. In Visual Basic, you raise events with the `RaiseEvent` statement, as in the following example:

```
RaiseEvent AnEvent(EventNumber)
```

Events must be raised within the scope of the class, module, or structure where they are declared. For example, a derived class cannot raise events inherited from a base class.

Event Senders

Any object capable of raising an event is an *event sender*, also known as an *event source*. Forms, controls, and user-defined objects are examples of event senders.

Event Handlers

Event handlers are procedures that are called when a corresponding event occurs. You can use any valid subroutine with a matching signature as an event handler. You cannot use a function as an event handler, however, because it cannot return a value to the event source.

Visual Basic uses a standard naming convention for event handlers that combines the name of the event sender, an underscore, and the name of the event. For example, the `Click` event of a button named `button1` would be named `Sub button1_Click`.

NOTE

We recommend that you use this naming convention when defining event handlers for your own events, but it is not required; you can use any valid subroutine name.

Associating Events with Event Handlers

Before an event handler becomes usable, you must first associate it with an event by using either the `Handles` or `AddHandler` statement.

WithEvents and the Handles Clause

The `WithEvents` statement and `Handles` clause provide a declarative way of specifying event handlers. An event raised by an object declared with the `WithEvents` keyword can be handled by any procedure with a `Handles` statement for that event, as shown in the following example:

```
' Declare a WithEvents variable.  
Dim WithEvents EClass As New EventClass  
  
' Call the method that raises the object's events.  
Sub TestEvents()  
    EClass.RaiseEvents()  
End Sub  
  
' Declare an event handler that handles multiple events.  
Sub EClass_EventHandler() Handles EClass.XEvent, EClass.YEvent  
    MsgBox("Received Event.")  
End Sub  
  
Class EventClass  
    Public Event XEvent()  
    Public Event YEvent()  
    ' RaiseEvents raises both events.  
    Sub RaiseEvents()  
        RaiseEvent XEvent()  
        RaiseEvent YEvent()  
    End Sub  
End Class
```

The `WithEvents` statement and the `Handles` clause are often the best choice for event handlers because the declarative syntax they use makes event handling easier to code, read and debug. However, be aware of the following limitations on the use of `WithEvents` variables:

- You cannot use a `WithEvents` variable as an object variable. That is, you cannot declare it as `Object` —you must specify the class name when you declare the variable.
- Because shared events are not tied to class instances, you cannot use `WithEvents` to declaratively handle shared events. Similarly, you cannot use `WithEvents` or `Handles` to handle events from a `Structure`. In both cases, you can use the `AddHandler` statement to handle those events.
- You cannot create arrays of `WithEvents` variables.

`WithEvents` variables allow a single event handler to handle one or more kind of event, or one or more event handlers to handle the same kind of event.

Although the `Handles` clause is the standard way of associating an event with an event handler, it is limited to associating events with event handlers at compile time.

In some cases, such as with events associated with forms or controls, Visual Basic automatically stubs out an

empty event handler and associates it with an event. For example, when you double-click a command button on a form in design mode, Visual Basic creates an empty event handler and a `WithEvents` variable for the command button, as in the following code:

```
Friend WithEvents Button1 As System.Windows.Forms.Button
Protected Sub Button1_Click() Handles Button1.Click
End Sub
```

AddHandler and RemoveHandler

The `AddHandler` statement is similar to the `Handles` clause in that both allow you to specify an event handler. However, `AddHandler`, used with `RemoveHandler`, provides greater flexibility than the `Handles` clause, allowing you to dynamically add, remove, and change the event handler associated with an event. If you want to handle shared events or events from a structure, you must use `AddHandler`.

`AddHandler` takes two arguments: the name of an event from an event sender such as a control, and an expression that evaluates to a delegate. You do not need to explicitly specify the delegate class when using `AddHandler`, since the `AddressOf` statement always returns a reference to the delegate. The following example associates an event handler with an event raised by an object:

```
AddHandler Obj.XEvent, AddressOf Me.XEventHandler
```

`RemoveHandler`, which disconnects an event from an event handler, uses the same syntax as `AddHandler`. For example:

```
RemoveHandler Obj.XEvent, AddressOf Me.XEventHandler
```

In the following example, an event handler is associated with an event, and the event is raised. The event handler catches the event and displays a message.

Then the first event handler is removed and a different event handler is associated with the event. When the event is raised again, a different message is displayed.

Finally, the second event handler is removed and the event is raised for a third time. Because there is no longer an event handler associated with the event, no action is taken.

```

Module Module1

    Sub Main()
        Dim c1 As New Class1
        ' Associate an event handler with an event.
        AddHandler c1.AnEvent, AddressOf EventHandler1
        ' Call a method to raise the event.
        c1.CauseTheEvent()
        ' Stop handling the event.
        RemoveHandler c1.AnEvent, AddressOf EventHandler1
        ' Now associate a different event handler with the event.
        AddHandler c1.AnEvent, AddressOf EventHandler2
        ' Call a method to raise the event.
        c1.CauseTheEvent()
        ' Stop handling the event.
        RemoveHandler c1.AnEvent, AddressOf EventHandler2
        ' This event will not be handled.
        c1.CauseTheEvent()
    End Sub

    Sub EventHandler1()
        ' Handle the event.
        MsgBox("EventHandler1 caught event.")
    End Sub

    Sub EventHandler2()
        ' Handle the event.
        MsgBox("EventHandler2 caught event.")
    End Sub

    Public Class Class1
        ' Declare an event.
        Public Event AnEvent()
        Sub CauseTheEvent()
            ' Raise an event.
            RaiseEvent AnEvent()
        End Sub
    End Class

End Module

```

Handling Events Inherited from a Base Class

Derived classes—classes that inherit characteristics from a base class—can handle events raised by their base class using the `Handles MyBase` statement.

To handle events from a base class

- Declare an event handler in the derived class by adding a `Handles MyBase.eventname` statement to the declaration line of your event-handler procedure, where `eventname` is the name of the event in the base class you are handling. For example:

```

Public Class BaseClass
    Public Event BaseEvent(ByVal i As Integer)
    ' Place methods and properties here.
End Class

Public Class DerivedClass
    Inherits BaseClass
    Sub EventHandler(ByVal x As Integer) Handles MyBase.BaseEvent
        ' Place code to handle events from BaseClass here.
    End Sub
End Class

```

Related Sections

TITLE	DESCRIPTION
Walkthrough: Declaring and Raising Events	Provides a step-by-step description of how to declare and raise events for a class.
Walkthrough: Handling Events	Demonstrates how to write an event-handler procedure.
How to: Declare Custom Events To Avoid Blocking	Demonstrates how to define a custom event that allows its event handlers to be called asynchronously.
How to: Declare Custom Events To Conserve Memory	Demonstrates how to define a custom event that uses memory only when the event is handled.
Troubleshooting Inherited Event Handlers in Visual Basic	Lists common issues that arise with event handlers in inherited components.
Events	Provides an overview of the event model in the .NET Framework.
Creating Event Handlers in Windows Forms	Describes how to work with events associated with Windows Forms objects.
Delegates	Provides an overview of delegates in Visual Basic.

Interfaces (Visual Basic)

8/22/2019 • 5 minutes to read • [Edit Online](#)

Interfaces define the properties, methods, and events that classes can implement. Interfaces allow you to define features as small groups of closely related properties, methods, and events; this reduces compatibility problems because you can develop enhanced implementations for your interfaces without jeopardizing existing code. You can add new features at any time by developing additional interfaces and implementations.

There are several other reasons why you might want to use interfaces instead of class inheritance:

- Interfaces are better suited to situations in which your applications require many possibly unrelated object types to provide certain functionality.
- Interfaces are more flexible than base classes because you can define a single implementation that can implement multiple interfaces.
- Interfaces are better in situations in which you do not have to inherit implementation from a base class.
- Interfaces are useful when you cannot use class inheritance. For example, structures cannot inherit from classes, but they can implement interfaces.

Declaring Interfaces

Interface definitions are enclosed within the `Interface` and `End Interface` statements. Following the `Interface` statement, you can add an optional `Inherits` statement that lists one or more inherited interfaces. The `Inherits` statements must precede all other statements in the declaration except comments. The remaining statements in the interface definition should be `Event`, `Sub`, `Function`, `Property`, `Interface`, `Class`, `Structure`, and `Enum` statements. Interfaces cannot contain any implementation code or statements associated with implementation code, such as `End Sub` or `End Property`.

In a namespace, interface statements are `Friend` by default, but they can also be explicitly declared as `Public` or `Friend`. Interfaces defined within classes, modules, interfaces, and structures are `Public` by default, but they can also be explicitly declared as `Public`, `Friend`, `Protected`, or `Private`.

NOTE

The `Shadows` keyword can be applied to all interface members. The `overloads` keyword can be applied to `Sub`, `Function`, and `Property` statements declared in an interface definition. In addition, `Property` statements can have the `Default`, `ReadOnly`, or `WriteOnly` modifiers. None of the other modifiers—`Public`, `Private`, `Friend`, `Protected`, `Shared`, `Overrides`, `MustOverride`, or `Overridable`—are allowed. For more information, see [Declaration Contexts and Default Access Levels](#).

For example, the following code defines an interface with one function, one property, and one event.

```
Interface IAsset
    Event CommittedChange(ByVal Success As Boolean)
    Property Division() As String
    Function GetID() As Integer
End Interface
```

Implementing Interfaces

The Visual Basic reserved word `Implements` is used in two ways. The `Implements` statement signifies that a class or structure implements an interface. The `Implements` keyword signifies that a class member or structure member implements a specific interface member.

Implements Statement

If a class or structure implements one or more interfaces, it must include the `Implements` statement immediately after the `Class` or `Structure` statement. The `Implements` statement requires a comma-separated list of interfaces to be implemented by a class. The class or structure must implement all interface members using the `Implements` keyword.

Implements Keyword

The `Implements` keyword requires a comma-separated list of interface members to be implemented. Generally, only a single interface member is specified, but you can specify multiple members. The specification of an interface member consists of the interface name, which must be specified in an `Implements` statement within the class; a period; and the name of the member function, property, or event to be implemented. The name of a member that implements an interface member can use any legal identifier, and it is not limited to the `InterfaceName_MethodName` convention used in earlier versions of Visual Basic.

For example, the following code shows how to declare a subroutine named `Sub1` that implements a method of an interface:

```
Class Class1
    Implements interfaceclass.interface2

    Sub Sub1(ByVal i As Integer) Implements interfaceclass.interface2.Sub1
        End Sub
    End Class
```

The parameter types and return types of the implementing member must match the interface property or member declaration in the interface. The most common way to implement an element of an interface is with a member that has the same name as the interface, as shown in the previous example.

To declare the implementation of an interface method, you can use any attributes that are legal on instance method declarations, including `Overloads`, `Overrides`, `Overridable`, `Public`, `Private`, `Protected`, `Friend`, `Protected Friend`, `MustOverride`, `Default`, and `Static`. The `Shared` attribute is not legal since it defines a class rather than an instance method.

Using `Implements`, you can also write a single method that implements multiple methods defined in an interface, as in the following example:

```
Class Class2
    Implements I1, I2

    Protected Sub M1() Implements I1.M1, I1.M2, I2.M3, I2.M4
        End Sub
    End Class
```

You can use a private member to implement an interface member. When a private member implements a member of an interface, that member becomes available by way of the interface even though it is not available directly on object variables for the class.

Interface Implementation Examples

Classes that implement an interface must implement all its properties, methods, and events.

The following example defines two interfaces. The second interface, `Interface2`, inherits `Interface1` and defines an additional property and method.

```

Interface Interface1
    Sub sub1(ByVal i As Integer)
End Interface

' Demonstrates interface inheritance.
Interface Interface2
    Inherits Interface1
    Sub M1(ByVal y As Integer)
    ReadOnly Property Num() As Integer
End Interface

```

The next example implements `Interface1`, the interface defined in the previous example:

```

Public Class ImplementationClass1
    Implements Interface1
    Sub Sub1(ByVal i As Integer) Implements Interface1.sub1
        ' Insert code here to implement this method.
    End Sub
End Class

```

The final example implements `Interface2`, including a method inherited from `Interface1`:

```

Public Class ImplementationClass2
    Implements Interface2
    Dim INum As Integer = 0
    Sub sub1(ByVal i As Integer) Implements Interface2.sub1
        ' Insert code here that implements this method.
    End Sub
    Sub M1(ByVal x As Integer) Implements Interface2.M1
        ' Insert code here to implement this method.
    End Sub

    ReadOnly Property Num() As Integer Implements Interface2.Num
        Get
            Num = INum
        End Get
    End Property
End Class

```

You can implement a readonly property with a readwrite property (that is, you do not have to declare it readonly in the implementing class). Implementing an interface promises to implement at least the members that the interface declares, but you can offer more functionality, such as allowing your property to be writable.

Related Topics

TITLE	DESCRIPTION
Walkthrough: Creating and Implementing Interfaces	Provides a detailed procedure that takes you through the process of defining and implementing your own interface.
Variance in Generic Interfaces	Discusses covariance and contravariance in generic interfaces and provides a list of variant generic interfaces in the .NET Framework.

Walkthrough: Creating and Implementing Interfaces (Visual Basic)

8/22/2019 • 4 minutes to read • [Edit Online](#)

Interfaces describe the characteristics of properties, methods, and events, but leave the implementation details up to structures or classes.

This walkthrough demonstrates how to declare and implement an interface.

NOTE

This walkthrough doesn't provide information about how to create a user interface.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

To define an interface

1. Open a new Visual Basic Windows Application project.
2. Add a new module to the project by clicking **Add Module** on the **Project** menu.
3. Name the new module `Module1.vb` and click **Add**. The code for the new module is displayed.
4. Define an interface named `TestInterface` within `Module1` by typing `Interface TestInterface` between the `Module` and `End Module` statements, and then pressing ENTER. The **Code Editor** indents the `Interface` keyword and adds an `End Interface` statement to form a code block.
5. Define a property, method, and event for the interface by placing the following code between the `Interface` and `End Interface` statements:

```
Property Prop1() As Integer  
Sub Method1(ByVal X As Integer)  
Event Event1()
```

Implementation

You may notice that the syntax used to declare interface members is different from the syntax used to declare class members. This difference reflects the fact that interfaces cannot contain implementation code.

To implement the interface

1. Add a class named `ImplementationClass` by adding the following statement to `Module1`, after the `End Interface` statement but before the `End Module` statement, and then pressing ENTER:

```
Class ImplementationClass
```

If you are working within the integrated development environment, the **Code Editor** supplies a matching `End Class` statement when you press ENTER.

2. Add the following `Implements` statement to `ImplementationClass`, which names the interface the class implements:

```
Implements TestInterface
```

When listed separately from other items at the top of a class or structure, the `Implements` statement indicates that the class or structure implements an interface.

If you are working within the integrated development environment, the **Code Editor** implements the class members required by `TestInterface` when you press ENTER, and you can skip the next step.

3. If you are not working within the integrated development environment, you must implement all the members of the interface `MyInterface`. Add the following code to `ImplementationClass` to implement `Event1`, `Method1`, and `Prop1`:

```
Event Event1() Implements TestInterface.Event1

Public Sub Method1(ByVal X As Integer) Implements TestInterface.Method1
End Sub

Public Property Prop1() As Integer Implements TestInterface.Prop1
    Get
    End Get
    Set(ByVal value As Integer)
    End Set
End Property
```

The `Implements` statement names the interface and interface member being implemented.

4. Complete the definition of `Prop1` by adding a private field to the class that stored the property value:

```
' Holds the value of the property.
Private pval As Integer
```

Return the value of the `pval` from the property get accessor.

```
Return pval
```

Set the value of `pval` in the property set accessor.

```
pval = value
```

5. Complete the definition of `Method1` by adding the following code.

```
MsgBox("The X parameter for Method1 is " & X)
RaiseEvent Event1()
```

To test the implementation of the interface

1. Right-click the startup form for your project in the **Solution Explorer**, and click **View Code**. The editor displays the class for your startup form. By default, the startup form is called `Form1`.

2. Add the following `testInstance` field to the `Form1` class:

```
Dim WithEvents testInstance As TestInterface
```

By declaring `testInstance` as `WithEvents`, the `Form1` class can handle its events.

3. Add the following event handler to the `Form1` class to handle events raised by `testInstance`:

```
Sub EventHandler() Handles testInstance.Event1
    MsgBox("The event handler caught the event.")
End Sub
```

4. Add a subroutine named `Test` to the `Form1` class to test the implementation class:

```
Sub Test()
    ' Create an instance of the class.
    Dim T As New ImplementationClass
    ' Assign the class instance to the interface.
    ' Calls to the interface members are
    ' executed through the class instance.
    testInstance = T
    ' Set a property.
    testInstance.Prop1 = 9
    ' Read the property.
    MsgBox("Prop1 was set to " & testInstance.Prop1)
    ' Test the method and raise an event.
    testInstance.Method1(5)
End Sub
```

The `Test` procedure creates an instance of the class that implements `MyInterface`, assigns that instance to the `testInstance` field, sets a property, and runs a method through the interface.

5. Add code to call the `Test` procedure from the `Form1 Load` procedure of your startup form:

```
Private Sub Form1_Load(ByVal sender As System.Object,
                      ByVal e As System.EventArgs) Handles MyBase.Load
    Test() ' Test the class.
End Sub
```

6. Run the `Test` procedure by pressing F5. The message "Prop1 was set to 9" is displayed. After you click OK, the message "The X parameter for Method1 is 5" is displayed. Click OK, and the message "The event handler caught the event" is displayed.

See also

- [Implements Statement](#)
- [Interfaces](#)
- [Interface Statement](#)
- [Event Statement](#)

LINQ in Visual Basic

4/2/2019 • 2 minutes to read • [Edit Online](#)

This section contains overviews, examples, and background information that will help you understand and use Visual Basic and Language-Integrated Query (LINQ).

In This Section

[Introduction to LINQ in Visual Basic](#)

Provides an introduction to LINQ providers, operators, query structure, and language features.

[How to: Query a Database](#)

Provides an example of how to connect to a SQL Server database and execute a query by using LINQ.

[How to: Call a Stored Procedure](#)

Provides an example of how to connect to a SQL Server database and call a stored procedure by using LINQ.

[How to: Modify Data in a Database](#)

Provides an example of how to connect to a SQL Server database and retrieve and modify data by using LINQ.

[How to: Combine Data with Joins](#)

Provides examples of how to join data in a manner similar to database joins by using LINQ.

[How to: Sort Query Results](#)

Provides an example of how to order the results of a query by using LINQ.

[How to: Filter Query Results](#)

Provides an example of how to include search criteria in a query by using LINQ.

[How to: Count, Sum, or Average Data](#)

Provides examples of how to include aggregate functions to Count, Sum, or Average data returned from a query by using LINQ.

[How to: Find the Minimum or Maximum Value in a Query Result](#)

Provides examples of how to include aggregate functions to determine the minimum and maximum values of data returned from a query by using LINQ.

[How to: Return a LINQ Query Result as a Specific Type](#)

Provides an example of how to return the results of a LINQ query as a specific type instead of as an anonymous type.

See also

- [LINQ \(Language-Integrated Query\)](#)
- [Overview of LINQ to XML in Visual Basic](#)
- [LINQ to DataSet Overview](#)
- [LINQ to SQL](#)

Objects and classes in Visual Basic

7/10/2019 • 11 minutes to read • [Edit Online](#)

An *object* is a combination of code and data that can be treated as a unit. An object can be a piece of an application, like a control or a form. An entire application can also be an object.

When you create an application in Visual Basic, you constantly work with objects. You can use objects provided by Visual Basic, such as controls, forms, and data access objects. You can also use objects from other applications within your Visual Basic application. You can even create your own objects and define additional properties and methods for them. Objects act like prefabricated building blocks for programs — they let you write a piece of code once and reuse it over and over.

This topic discusses objects in detail.

Objects and classes

Each object in Visual Basic is defined by a *class*. A class describes the variables, properties, procedures, and events of an object. Objects are instances of classes; you can create as many objects you need once you have defined a class.

To understand the relationship between an object and its class, think of cookie cutters and cookies. The cookie cutter is the class. It defines the characteristics of each cookie, for example size and shape. The class is used to create objects. The objects are the cookies.

You must create an object before you can access its members.

To create an object from a class

1. Determine from which class you want to create an object.
2. Write a [Dim Statement](#) to create a variable to which you can assign a class instance. The variable should be of the type of the desired class.

```
Dim nextCustomer As customer
```

3. Add the [New Operator](#) keyword to initialize the variable to a new instance of the class.

```
Dim nextCustomer As New customer
```

4. You can now access the members of the class through the object variable.

```
nextCustomer.accountNumber = lastAccountNumber + 1
```

NOTE

Whenever possible, you should declare the variable to be of the class type you intend to assign to it. This is called *early binding*. If you don't know the class type at compile time, you can invoke *late binding* by declaring the variable to be of the [Object Data Type](#). However, late binding can make performance slower and limit access to the run-time object's members. For more information, see [Object Variable Declaration](#).

Multiple instances

Objects newly created from a class are often identical to each other. Once they exist as individual objects, however, their variables and properties can be changed independently of the other instances. For example, if you add three check boxes to a form, each check box object is an instance of the [CheckBox](#) class. The individual [CheckBox](#) objects share a common set of characteristics and capabilities (properties, variables, procedures, and events) defined by the class. However, each has its own name, can be separately enabled and disabled, and can be placed in a different location on the form.

Object members

An object is an element of an application, representing an *instance* of a class. Fields, properties, methods, and events are the building blocks of objects and constitute their *members*.

Member Access

You access a member of an object by specifying, in order, the name of the object variable, a period (.), and the name of the member. The following example sets the [Text](#) property of a [Label](#) object.

```
warningLabel.Text = "Data not saved"
```

IntelliSense listing of members

IntelliSense lists members of a class when you invoke its List Members option, for example when you type a period (.) as a member-access operator. If you type the period following the name of a variable declared as an instance of that class, IntelliSense lists all the instance members and none of the shared members. If you type the period following the class name itself, IntelliSense lists all the shared members and none of the instance members. For more information, see [Using IntelliSense](#).

Fields and properties

Fields and *properties* represent information stored in an object. You retrieve and set their values with assignment statements the same way you retrieve and set local variables in a procedure. The following example retrieves the [Width](#) property and sets the [ForeColor](#) property of a [Label](#) object.

```
Dim warningWidth As Integer = warningLabel.Width  
warningLabel.ForeColor = System.Drawing.Color.Red
```

Note that a field is also called a *member variable*.

Use property procedures when:

- You need to control when and how a value is set or retrieved.
- The property has a well-defined set of values that need to be validated.
- Setting the value causes some perceptible change in the object's state, such as an [IsVisible](#) property.
- Setting the property causes changes to other internal variables or to the values of other properties.
- A set of steps must be performed before the property can be set or retrieved.

Use fields when:

- The value is of a self-validating type. For example, an error or automatic data conversion occurs if a value other than [True](#) or [False](#) is assigned to a [Boolean](#) variable.
- Any value in the range supported by the data type is valid. This is true of many properties of type [Single](#) or [Double](#).

- The property is a `String` data type, and there is no constraint on the size or value of the string.
- For more information, see [Property Procedures](#).

Methods

A *method* is an action that an object can perform. For example, `Add` is a method of the `ComboBox` object that adds a new entry to a combo box.

The following example demonstrates the `Start` method of a `Timer` object.

```
Dim safetyTimer As New System.Windows.Forms.Timer
safetyTimer.Start()
```

Note that a method is simply a *procedure* that is exposed by an object.

For more information, see [Procedures](#).

Events

An event is an action recognized by an object, such as clicking the mouse or pressing a key, and for which you can write code to respond. Events can occur as a result of a user action or program code, or they can be caused by the system. Code that signals an event is said to *raise* the event, and code that responds to it is said to *handle* it.

You can also develop your own custom events to be raised by your objects and handled by other objects. For more information, see [Events](#).

Instance members and shared members

When you create an object from a class, the result is an instance of that class. Members that are not declared with the `Shared` keyword are *instance members*, which belong strictly to that particular instance. An instance member in one instance is independent of the same member in another instance of the same class. An instance member variable, for example, can have different values in different instances.

Members declared with the `Shared` keyword are *shared members*, which belong to the class as a whole and not to any particular instance. A shared member exists only once, no matter how many instances of its class you create, or even if you create no instances. A shared member variable, for example, has only one value, which is available to all code that can access the class.

Accessing nonshared members

To access a nonshared member of an object

1. Make sure the object has been created from its class and assigned to an object variable.

```
Dim secondForm As New System.Windows.Forms.Form
```

2. In the statement that accesses the member, follow the object variable name with the *member-access operator* (`.`) and then the member name.

```
secondForm.Show()
```

Accessing shared members

To access a shared member of an object

- Follow the class name with the *member-access operator* (`.`) and then the member name. You should always access a `Shared` member of the object directly through the class name.

```
MsgBox("This computer is called " & Environment.MachineName)
```

- If you have already created an object from the class, you can alternatively access a `Shared` member through the object's variable.

Differences between classes and modules

The main difference between classes and modules is that classes can be instantiated as objects while standard modules cannot. Because there is only one copy of a standard module's data, when one part of your program changes a public variable in a standard module, any other part of the program gets the same value if it then reads that variable. In contrast, object data exists separately for each instantiated object. Another difference is that unlike standard modules, classes can implement interfaces.

NOTE

When the `Shared` modifier is applied to a class member, it is associated with the class itself instead of a particular instance of the class. The member is accessed directly by using the class name, the same way module members are accessed.

Classes and modules also use different scopes for their members. Members defined within a class are scoped within a specific instance of the class and exist only for the lifetime of the object. To access class members from outside a class, you must use fully qualified names in the format of *Object.Member*.

On the other hand, members declared within a module are publicly accessible by default, and can be accessed by any code that can access the module. This means that variables in a standard module are effectively global variables because they are visible from anywhere in your project, and they exist for the life of the program.

Reusing classes and objects

Objects let you declare variables and procedures once and then reuse them whenever needed. For example, if you want to add a spelling checker to an application you could define all the variables and support functions to provide spell-checking functionality. If you create your spelling checker as a class, you can then reuse it in other applications by adding a reference to the compiled assembly. Better yet, you may be able to save yourself some work by using a spelling checker class that someone else has already developed.

The .NET Framework provides many examples of components that are available for use. The following example uses the `TimeZone` class in the `System` namespace. `TimeZone` provides members that allow you to retrieve information about the time zone of the current computer system.

```
Public Sub examineTimeZone()
    Dim tz As System.TimeZone = System.TimeZone.CurrentTimeZone
    Dim s As String = "Current time zone is "
    s &= CStr(tz.GetUtcOffset(Now).Hours) & " hours and "
    s &= CStr(tz.GetUtcOffset(Now).Minutes) & " minutes "
    s &= "different from UTC (coordinated universal time)"
    s &= vbCrLf & "and is currently "
    If tz.IsDaylightSavingTime(Now) = False Then s &= "not "
    s &= "on ""summer time""."
    MsgBox(s)
End Sub
```

In the preceding example, the first `Dim Statement` declares an object variable of type `TimeZone` and assigns to it a `TimeZone` object returned by the `CurrentTimeZone` property.

Relationships among objects

Objects can be related to each other in several ways. The principal kinds of relationship are *hierarchical* and *containment*.

Hierarchical relationship

When classes are derived from more fundamental classes, they are said to have a *hierarchical relationship*. Class hierarchies are useful when describing items that are a subtype of a more general class.

In the following example, suppose you want to define a special kind of [Button](#) that acts like a normal [Button](#) but also exposes a method that reverses the foreground and background colors.

To define a class is derived from an already existing class

1. Use a [Class Statement](#) to define a class from which to create the object you need.

```
Public Class reversibleButton
```

Be sure an [End Class](#) statement follows the last line of code in your class. By default, the integrated development environment (IDE) automatically generates an [End Class](#) when you enter a [Class](#) statement.

2. Follow the [Class](#) statement immediately with an [Inherits Statement](#). Specify the class from which your new class derives.

```
Inherits System.Windows.Forms.Button
```

Your new class inherits all the members defined by the base class.

3. Add the code for the additional members your derived class exposes. For example, you might add a [reverseColors](#) method, and your derived class might look as follows:

```
Public Class reversibleButton
    Inherits System.Windows.Forms.Button
    Public Sub reverseColors()
        Dim saveColor As System.Drawing.Color = Me.BackColor
        Me.BackColor = Me.ForeColor
        Me.ForeColor = saveColor
    End Sub
End Class
```

If you create an object from the [reversibleButton](#) class, it can access all the members of the [Button](#) class, as well as the [reverseColors](#) method and any other new members you define on [reversibleButton](#).

Derived classes inherit members from the class they are based on, allowing you to add complexity as you progress in a class hierarchy. For more information, see [Inheritance Basics](#).

Compiling the code

Be sure the compiler can access the class from which you intend to derive your new class. This might mean fully qualifying its name, as in the preceding example, or identifying its namespace in an [Imports Statement \(.NET Namespace and Type\)](#). If the class is in a different project, you might need to add a reference to that project. For more information, see [Managing references in a project](#).

Containment relationship

Another way that objects can be related is a *containment relationship*. Container objects logically encapsulate other objects. For example, the [OperatingSystem](#) object logically contains a [Version](#) object, which it returns through its [Version](#) property. Note that the container object does not physically contain any other object.

Collections

One particular type of object containment is represented by *collections*. Collections are groups of similar objects that can be enumerated. Visual Basic supports a specific syntax in the [For Each...Next Statement](#) that allows you

to iterate through the items of a collection. Additionally, collections often allow you to use an `Item[String]` to retrieve elements by their index or by associating them with a unique string. Collections can be easier to use than arrays because they allow you to add or remove items without using indexes. Because of their ease of use, collections are often used to store forms and controls.

Related topics

[Walkthrough: Defining Classes](#)

Provides a step-by-step description of how to create a class.

[Overloaded Properties and Methods](#)

Overloaded Properties and Methods

[Inheritance Basics](#)

Covers inheritance modifiers, overriding methods and properties, `MyClass`, and `MyBase`.

[Object Lifetime: How Objects Are Created and Destroyed](#)

Discusses creating and disposing of class instances.

[Anonymous Types](#)

Describes how to create and use anonymous types, which allow you to create objects without writing a class definition for the data type.

[Object Initializers: Named and Anonymous Types](#)

Discusses object initializers, which are used to create instances of named and anonymous types by using a single expression.

[How to: Infer Property Names and Types in Anonymous Type Declarations](#)

Explains how to infer property names and types in anonymous type declarations. Provides examples of successful and unsuccessful inference.

Operators and Expressions in Visual Basic

4/28/2019 • 2 minutes to read • [Edit Online](#)

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](#).

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.

```
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.

See also

- [Operators](#)
- [Efficient Combination of Operators](#)
- [Statements](#)

Procedures in Visual Basic

4/28/2019 • 3 minutes to read • [Edit Online](#)

A *procedure* is a block of Visual Basic statements enclosed by a declaration statement (`Function`, `Sub`, `Operator`, `Get`, `Set`) and a matching `End` declaration. All executable statements in Visual Basic must be within some procedure.

Calling a Procedure

You invoke a procedure from some other place in the code. This is known as a *procedure call*. When the procedure is finished running, it returns control to the code that invoked it, which is known as the *calling code*. The calling code is a statement, or an expression within a statement, that specifies the procedure by name and transfers control to it.

Returning from a Procedure

A procedure returns control to the calling code when it has finished running. To do this, it can use a [Return Statement](#), the appropriate [Exit Statement](#) statement for the procedure, or the procedure's [End <keyword> Statement](#). Control then passes to the calling code following the point of the procedure call.

- With a `Return` statement, control returns immediately to the calling code. Statements following the `Return` statement do not run. You can have more than one `Return` statement in the same procedure.
- With an `Exit Sub` or `Exit Function` statement, control returns immediately to the calling code. Statements following the `Exit` statement do not run. You can have more than one `Exit` statement in the same procedure, and you can mix `Return` and `Exit` statements in the same procedure.
- If a procedure has no `Return` or `Exit` statements, it concludes with an `End Sub` or `End Function`, `End Get`, or `End Set` statement following the last statement of the procedure body. The `End` statement returns control immediately to the calling code. You can have only one `End` statement in a procedure.

Parameters and Arguments

In most cases, a procedure needs to operate on different data each time you call it. You can pass this information to the procedure as part of the procedure call. The procedure defines zero or more *parameters*, each of which represents a value it expects you to pass to it. Corresponding to each parameter in the procedure definition is an *argument* in the procedure call. An argument represents the value you pass to the corresponding parameter in a given procedure call.

Types of Procedures

Visual Basic uses several types of procedures:

- [Sub Procedures](#) perform actions but do not return a value to the calling code.
- Event-handling procedures are `Sub` procedures that execute in response to an event raised by user action or by an occurrence in a program.
- [Function Procedures](#) return a value to the calling code. They can perform other actions before returning.

Some functions written in C# return a *reference return value*. Function callers can modify the return value, and this modification is reflected in the state of the called object. Starting with Visual Basic 2017, Visual

Basic code can consume reference return values, although it cannot return a value by reference. For more information, see [Reference return values](#).

- [Property Procedures](#) return and assign values of properties on objects or modules.
- [Operator Procedures](#) define the behavior of a standard operator when one or both of the operands is a newly-defined class or structure.
- [Generic Procedures in Visual Basic](#) define one or more *type parameters* in addition to their normal parameters, so the calling code can pass specific data types each time it makes a call.

Procedures and Structured Code

Every line of executable code in your application must be inside some procedure, such as `Main`, `calculate`, or `Button1_Click`. If you subdivide large procedures into smaller ones, your application is more readable.

Procedures are useful for performing repeated or shared tasks, such as frequently used calculations, text and control manipulation, and database operations. You can call a procedure from many different places in your code, so you can use procedures as building blocks for your application.

Structuring your code with procedures gives you the following benefits:

- Procedures allow you to break your programs into discrete logical units. You can debug separate units more easily than you can debug an entire program without procedures.
- After you develop procedures for use in one program, you can use them in other programs, often with little or no modification. This helps you avoid code duplication.

See also

- [How to: Create a Procedure](#)
- [Sub Procedures](#)
- [Function Procedures](#)
- [Property Procedures](#)
- [Operator Procedures](#)
- [Procedure Parameters and Arguments](#)
- [Recursive Procedures](#)
- [Procedure Overloading](#)
- [Generic Procedures in Visual Basic](#)
- [Objects and Classes](#)

Statements in Visual Basic

8/24/2018 • 12 minutes to read • [Edit Online](#)

A statement in Visual Basic is a complete instruction. It can contain keywords, operators, variables, constants, and expressions. Each statement belongs to one of the following categories:

- **Declaration Statements**, which name a variable, constant, or procedure, and can also specify a data type.
- **Executable Statements**, which initiate actions. These statements can call a method or function, and they can loop or branch through blocks of code. Executable statements include **Assignment Statements**, which assign a value or expression to a variable or constant.

This topic describes each category. Also, this topic describes how to combine multiple statements on a single line and how to continue a statement over multiple lines.

Declaration statements

You use declaration statements to name and define procedures, variables, properties, arrays, and constants. When you declare a programming element, you can also define its data type, access level, and scope. For more information, see [Declared Element Characteristics](#).

The following example contains three declarations.

```
Public Sub ApplyFormat()
    Const limit As Integer = 33
    Dim thisWidget As New widget
    ' Insert code to implement the procedure.
End Sub
```

The first declaration is the `Sub` statement. Together with its matching `End Sub` statement, it declares a procedure named `applyFormat`. It also specifies that `applyFormat` is `Public`, which means that any code that can refer to it can call it.

The second declaration is the `Const` statement, which declares the constant `limit`, specifying the `Integer` data type and a value of 33.

The third declaration is the `Dim` statement, which declares the variable `thisWidget`. The data type is a specific object, namely an object created from the `widget` class. You can declare a variable to be of any elementary data type or of any object type that is exposed in the application you are using.

Initial Values

When the code containing a declaration statement runs, Visual Basic reserves the memory required for the declared element. If the element holds a value, Visual Basic initializes it to the default value for its data type. For more information, see "Behavior" in [Dim Statement](#).

You can assign an initial value to a variable as part of its declaration, as the following example illustrates.

```
Dim m As Integer = 45
' The preceding declaration creates m and assigns the value 45 to it.
```

If a variable is an object variable, you can explicitly create an instance of its class when you declare it by using

the [New Operator](#) keyword, as the following example illustrates.

```
Dim f As New System.Windows.Forms.Form()
```

Note that the initial value you specify in a declaration statement is not assigned to a variable until execution reaches its declaration statement. Until that time, the variable contains the default value for its data type.

Executable statements

An executable statement performs an action. It can call a procedure, branch to another place in the code, loop through several statements, or evaluate an expression. An assignment statement is a special case of an executable statement.

The following example uses an `If...Then...Else` control structure to run different blocks of code based on the value of a variable. Within each block of code, a `For...Next` loop runs a specified number of times.

```
Public Sub StartWidget(ByVal aWidget As widget,
    ByVal clockwise As Boolean, ByVal revolutions As Integer)
    Dim counter As Integer
    If clockwise = True Then
        For counter = 1 To revolutions
            aWidget.SpinClockwise()
        Next counter
    Else
        For counter = 1 To revolutions
            aWidget.SpinCounterClockwise()
        Next counter
    End If
End Sub
```

The `If` statement in the preceding example checks the value of the parameter `clockwise`. If the value is `True`, it calls the `spinClockwise` method of `aWidget`. If the value is `False`, it calls the `spinCounterClockwise` method of `aWidget`. The `If...Then...Else` control structure ends with `End If`.

The `For...Next` loop within each block calls the appropriate method a number of times equal to the value of the `revolutions` parameter.

Assignment statements

Assignment statements carry out assignment operations, which consist of taking the value on the right side of the assignment operator (`=`) and storing it in the element on the left, as in the following example.

```
v = 42
```

In the preceding example, the assignment statement stores the literal value 42 in the variable `v`.

Eligible programming elements

The programming element on the left side of the assignment operator must be able to accept and store a value. This means it must be a variable or property that is not [ReadOnly](#), or it must be an array element. In the context of an assignment statement, such an element is sometimes called an *lvalue*, for "left value."

The value on the right side of the assignment operator is generated by an expression, which can consist of any combination of literals, constants, variables, properties, array elements, other expressions, or function calls. The following example illustrates this.

```
x = y + z + FindResult(3)
```

The preceding example adds the value held in variable `y` to the value held in variable `z`, and then adds the value returned by the call to function `FindResult`. The total value of this expression is then stored in variable `x`.

Data types in assignment statements

In addition to numeric values, the assignment operator can also assign `String` values, as the following example illustrates.

```
Dim a, b As String
a = "String variable assignment"
b = "Con" & "cat" & "enation"
' The preceding statement assigns the value "Concatenation" to b.
```

You can also assign `Boolean` values, using either a `Boolean` literal or a `Boolean` expression, as the following example illustrates.

```
Dim r, s, t As Boolean
r = True
s = 45 > 1003
t = 45 > 1003 Or 45 > 17
' The preceding statements assign False to s and True to t.
```

Similarly, you can assign appropriate values to programming elements of the `Char`, `Date`, or `Object` data type. You can also assign an object instance to an element declared to be of the class from which that instance is created.

Compound assignment statements

Compound assignment statements first perform an operation on an expression before assigning it to a programming element. The following example illustrates one of these operators, `+=`, which increments the value of the variable on the left side of the operator by the value of the expression on the right.

```
n += 1
```

The preceding example adds 1 to the value of `n`, and then stores that new value in `n`. It is a shorthand equivalent of the following statement:

```
n = n + 1
```

A variety of compound assignment operations can be performed using operators of this type. For a list of these operators and more information about them, see [Assignment Operators](#).

The concatenation assignment operator (`&=`) is useful for adding a string to the end of already existing strings, as the following example illustrates.

```
Dim q As String = "Sample "
q &= "String"
' q now contains "Sample String".
```

Type Conversions in Assignment Statements

The value you assign to a variable, property, or array element must be of a data type appropriate to that destination element. In general, you should try to generate a value of the same data type as that of the

destination element. However, some types can be converted to other types during assignment.

For information on converting between data types, see [Type Conversions in Visual Basic](#). In brief, Visual Basic automatically converts a value of a given type to any other type to which it widens. A *widening conversion* is one in that always succeeds at run time and does not lose any data. For example, Visual Basic converts an `Integer` value to `Double` when appropriate, because `Integer` widens to `Double`. For more information, see [Widening and Narrowing Conversions](#).

Narrowing conversions (those that are not widening) carry a risk of failure at run time, or of data loss. You can perform a narrowing conversion explicitly by using a type conversion function, or you can direct the compiler to perform all conversions implicitly by setting `Option Strict off`. For more information, see [Implicit and Explicit Conversions](#).

Putting multiple statements on one line

You can have multiple statements on a single line separated by the colon (`:`) character. The following example illustrates this.

```
Dim sampleString As String = "Hello World" : MsgBox(sampleString)
```

Though occasionally convenient, this form of syntax makes your code hard to read and maintain. Thus, it is recommended that you keep one statement to a line.

Continuing a statement over multiple lines

A statement usually fits on one line, but when it is too long, you can continue it onto the next line using a line-continuation sequence, which consists of a space followed by an underscore character (`_`) followed by a carriage return. In the following example, the `MsgBox` executable statement is continued over two lines.

```
Public Sub DemoBox()
    Dim nameVar As String
    nameVar = "John"
    MsgBox("Hello " & nameVar _
        & ". How are you?")
End Sub
```

Implicit line continuation

In many cases, you can continue a statement on the next consecutive line without using the underscore character (`_`). The following syntax elements implicitly continue the statement on the next line of code.

- After a comma (`,`). For example:

```
Public Function GetUsername(ByVal username As String,
                           ByVal delimiter As Char,
                           ByVal position As Integer) As String

    Return username.Split(delimiter)(position)
End Function
```

- After an open parenthesis (`(`) or before a closing parenthesis (`)`). For example:

```
Dim username = GetUsername()  
    Security.Principal.WindowsIdentity.GetCurrent().Name,  
    CChar("\\"),  
    1  
)
```

- After an open curly brace (`{`) or before a closing curly brace (`}`). For example:

```
Dim customer = New Customer With {  
    .Name = "Terry Adams",  
    .Company = "Adventure Works",  
    .Email = "terry@www.adventure-works.com"  
}
```

For more information, see [Object Initializers: Named and Anonymous Types](#) or [Collection Initializers](#).

- After an open embedded expression (`<%=`) or before the close of an embedded expression (`%>`) within an XML literal. For example:

```
Dim customerXml = <Customer>  
    <Name>  
        <%=  
            customer.Name  
        %>  
    </Name>  
    <Email>  
        <%=  
            customer.Email  
        %>  
    </Email>  
</Customer>
```

For more information, see [Embedded Expressions in XML](#).

- After the concatenation operator (`&`). For example:

```
cmd.CommandText =  
    "SELECT * FROM Titles JOIN Publishers " &  
    "ON Publishers.PubId = Titles.PubID " &  
    "WHERE Publishers.State = 'CA'"
```

For more information, see [Operators Listed by Functionality](#).

- After assignment operators (`=`, `&=`, `:=`, `+=`, `-=`, `*=`, `/=`, `\=`, `^=`, `<<=`, `>>=`). For example:

```
Dim fileStream =  
    My.Computer.FileSystem.  
    OpenTextFileReader(filePath)
```

For more information, see [Operators Listed by Functionality](#).

- After binary operators (`+`, `-`, `/`, `*`, `Mod`, `<>`, `<`, `>`, `<=`, `>=`, `^`, `>>`, `<<`, `And`, `AndAlso`, `Or`, `OrElse`, `Like`, `Xor`) within an expression. For example:

```
Dim memoryInUse =
    My.Computer.Info.TotalPhysicalMemory +
    My.Computer.Info.TotalVirtualMemory -
    My.Computer.Info.AvailablePhysicalMemory -
    My.Computer.Info.AvailableVirtualMemory
```

For more information, see [Operators Listed by Functionality](#).

- After the `Is` and `IsNot` operators. For example:

```
If TypeOf inStream Is
    IO.FileStream AndAlso
    inStream IsNot
    Nothing Then

    ReadFile(inStream)

End If
```

For more information, see [Operators Listed by Functionality](#).

- After a member qualifier character (`.`) and before the member name. For example:

```
Dim fileStream =
    My.Computer.FileSystem.
        OpenTextFileReader(filePath)
```

However, you must include a line-continuation character (`_`) following a member qualifier character when you are using the `With` statement or supplying values in the initialization list for a type. Consider breaking the line after the assignment operator (for example, `=`) when you are using `With` statements or object initialization lists. For example:

```
' Not allowed:
' Dim aType = New With { .
'     PropertyName = "Value"

' Allowed:
Dim aType = New With {.PropertyName =
    "Value"}
```

```
Dim log As New EventLog()
```

```
' Not allowed:
' With log
'
'     Source = "Application"
' End With

' Allowed:
With log
    .Source =
        "Application"
End With
```

For more information, see [With...End With Statement](#) or [Object Initializers: Named and Anonymous Types](#).

- After an XML axis property qualifier (`.` or `@` or `...`). However, you must include a line-continuation character (`_`) when you specify a member qualifier when you are using the `With` keyword. For example:

```
Dim customerName = customerXml.  
    <Name>.Value  
  
Dim customerEmail = customerXml...  
    <Email>.Value
```

For more information, see [XML Axis Properties](#).

- After a less-than sign (`<`) or before a greater-than sign (`>`) when you specify an attribute. Also after a greater-than sign (`>`) when you specify an attribute. However, you must include a line-continuation character (`_`) when you specify assembly-level or module-level attributes. For example:

```
<  
Serializable()  
>  
Public Class Customer  
    Public Property Name As String  
    Public Property Company As String  
    Public Property Email As String  
End Class
```

For more information, see [Attributes overview](#).

- Before and after query operators (`Aggregate` , `Distinct` , `From` , `Group By` , `Group Join` , `Join` , `Let` , `Order By` , `Select` , `Skip` , `Skip While` , `Take` , `Take While` , `Where` , `In` , `Into` , `On` , `Ascending` , and `Descending`). You cannot break a line between the keywords of query operators that are made up of multiple keywords (`order By` , `Group Join` , `Take While` , and `Skip While`). For example:

```
Dim vsProcesses = From proc In  
    Process.GetProcesses  
    Where proc.MainWindowTitle.Contains("Visual Studio")  
    Select proc.ProcessName, proc.Id,  
        proc.MainWindowTitle
```

For more information, see [Queries](#).

- After the `In` keyword in a `For Each` statement. For example:

```
For Each p In  
    vsProcesses  
  
    Console.WriteLine("{0}" & vbTab & "{1}" & vbTab & "{2}",  
        p.ProcessName,  
        p.Id,  
        p.MainWindowTitle)  
Next
```

For more information, see [For Each...Next Statement](#).

- After the `From` keyword in a collection initializer. For example:

```
Dim days = New List(Of String) From
{
    "Mo", "Tu", "We", "Th", "F", "Sa", "Su"
}
```

For more information, see [Collection Initializers](#).

Adding comments

Source code is not always self-explanatory, even to the programmer who wrote it. To help document their code, therefore, most programmers make liberal use of embedded comments. Comments in code can explain a procedure or a particular instruction to anyone reading or working with it later. Visual Basic ignores comments during compilation, and they do not affect the compiled code.

Comment lines begin with an apostrophe (') or REM followed by a space. They can be added anywhere in code, except within a string. To append a comment to a statement, insert an apostrophe or REM after the statement, followed by the comment. Comments can also go on their own separate line. The following example demonstrates these possibilities.

```
' This is a comment on a separate code line.
REM This is another comment on a separate code line.
x += a(i) * b(i) ' Add this amount to total.
MsgBox(statusMessage) REM Inform operator of status.
```

Checking compilation errors

If, after you type a line of code, the line is displayed with a wavy blue underline (an error message may appear as well), there is a syntax error in the statement. You must find out what is wrong with the statement (by looking in the task list, or hovering over the error with the mouse pointer and reading the error message) and correct it. Until you have fixed all syntax errors in your code, your program will fail to compile correctly.

Related sections

TERM	DEFINITION
Assignment Operators	Provides links to language reference pages covering assignment operators such as =, *=, and &=.
Operators and Expressions	Shows how to combine elements with operators to yield new values.
How to: Break and Combine Statements in Code	Shows how to break a single statement into multiple lines and how to place multiple statements on the same line.
How to: Label Statements	Shows how to label a line of code.

Strings in Visual Basic

4/2/2019 • 2 minutes to read • [Edit Online](#)

This section describes the basic concepts behind using strings in Visual Basic.

In This Section

[Introduction to Strings in Visual Basic](#)

Lists topics that describe the basic concepts behind using strings in Visual Basic.

[How to: Create Strings Using a StringBuilder in Visual Basic](#)

Demonstrates how to efficiently create a long string from many smaller strings.

[How to: Search Within a String](#)

Demonstrates how to determine the index of the first occurrence of a substring.

[Converting Between Strings and Other Data Types in Visual Basic](#)

Lists topics that describe how to convert strings into other data types.

[Validating Strings in Visual Basic](#)

Lists topics that discuss how to validate strings.

[Walkthrough: Encrypting and Decrypting Strings in Visual Basic](#)

Demonstrates how to encrypt and decrypt strings by using the cryptographic service provider version of the Triple Data Encryption Standard algorithm.

See also

- [Visual Basic Language Features](#)

Variables in Visual Basic

8/22/2019 • 2 minutes to read • [Edit Online](#)

You often have to store values when you perform calculations with Visual Basic. For example, you might want to calculate several values, compare them, and perform different operations on them, depending on the result of the comparison. You have to retain the values if you want to compare them.

Usage

Visual Basic, just like most programming languages, uses variables for storing values. A *variable* has a name (the word that you use to refer to the value that the variable contains). A variable also has a data type (which determines the kind of data that the variable can store). A variable can represent an array if it has to store an indexed set of closely related data items.

Local type inference enables you to declare variables without explicitly stating a data type. Instead, the compiler infers the type of the variable from the type of the initialization expression. For more information, see [Local Type Inference](#) and [Option Infer Statement](#).

Assigning Values

You use assignment statements to perform calculations and assign the result to a variable, as the following example shows.

```
' The following statement assigns the value 10 to the variable.  
applesSold = 10  
' The following statement increments the variable.  
applesSold = applesSold + 1  
' The variable now holds the value 11.
```

NOTE

The equal sign (`=`) in this example is an assignment operator, not an equality operator. The value is being assigned to the variable `applesSold`.

For more information, see [How to: Move Data Into and Out of a Variable](#).

Variables and Properties

Like a variable, a *property* represents a value that you can access. However, it is more complex than a variable. A property uses code blocks that control how to set and retrieve its value. For more information, see [Differences Between Properties and Variables in Visual Basic](#).

See also

- [Variable Declaration](#)
- [Object Variables](#)
- [Troubleshooting Variables](#)
- [How to: Move Data Into and Out of a Variable](#)
- [Differences Between Properties and Variables in Visual Basic](#)

- Local Type Inference

XML in Visual Basic

1/23/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic provides integrated language support that enables it to interact with LINQ to XML.

In this section

The topics in this section introduce using LINQ to XML with Visual Basic.

TOPIC	DESCRIPTION
Overview of LINQ to XML in Visual Basic	Describes how Visual Basic supports LINQ to XML.
Creating XML in Visual Basic	Describes how to create XML literal objects by using LINQ to XML.
Manipulating XML in Visual Basic	Describes how to load and parse XML by using Visual Basic.
Accessing XML in Visual Basic	Describes the XML axis properties and LINQ to XML methods for accessing XML elements and attributes.

See also

- [System.Xml.Linq](#)
- [XML Literals](#)
- [XML Axis Properties](#)
- [LINQ to XML](#)

COM Interop (Visual Basic)

7/30/2019 • 2 minutes to read • [Edit Online](#)

The Component Object Model (COM) allows an object to expose its functionality to other components and to host applications. Most of today's software includes COM objects. Although .NET assemblies are the best choice for new applications, you may at times need to employ COM objects. This section covers some of the issues associated with creating and using COM objects with Visual Basic.

In This Section

[Introduction to COM Interop](#)

Provides an overview of COM interoperability.

[How to: Reference COM Objects from Visual Basic](#)

Covers how to add references to COM objects that have type libraries.

[How to: Work with ActiveX Controls](#)

Demonstrates how to use existing ActiveX controls to add features to the Visual Studio Toolbox.

[Walkthrough: Calling Windows APIs](#)

Steps you through the process of calling the APIs that are part of the Windows operating system.

[How to: Call Windows APIs](#)

Demonstrates how to define and call the `MessageBox` function in User32.dll.

[How to: Call a Windows Function that Takes Unsigned Types](#)

Demonstrates how to call a Windows function that has a parameter of an unsigned type.

[Walkthrough: Creating COM Objects with Visual Basic](#)

Steps you through the process of creating COM objects with and without the COM class template.

[Troubleshooting Interoperability](#)

Covers some of the problems you may encounter when using COM.

[COM Interoperability in .NET Framework Applications](#)

Provides an overview of how to use COM objects and .NET Framework objects in the same application.

[Walkthrough: Implementing Inheritance with COM Objects](#)

Describes using existing COM objects as the basis for new objects.

Related Sections

[Interoperating with Unmanaged Code](#)

Describes interoperability services provided by the common language runtime.

[Exposing COM Components to the .NET Framework](#)

Describes the process of calling COM types through COM interop.

[Exposing .NET Framework Components to COM](#)

Describes the preparation and use of managed types from COM.

[Applying Interop Attributes](#)

Covers attributes you can use when working with unmanaged code.

Introduction to COM Interop (Visual Basic)

5/14/2019 • 2 minutes to read • [Edit Online](#)

The Component Object Model (COM) lets an object expose its functionality to other components and to host applications. While COM objects have been fundamental to Windows programming for many years, applications designed for the common language runtime (CLR) offer many advantages.

.NET Framework applications will eventually replace those developed with COM. Until then, you may have to use or create COM objects by using Visual Studio. Interoperability with COM, or *COM interop*, enables you to use existing COM objects while transitioning to the .NET Framework at your own pace.

By using the .NET Framework to create COM components, you can use registration-free COM interop. This lets you control which DLL version is enabled when more than one version is installed on a computer, and lets end users use XCOPY or FTP to copy your application to an appropriate directory on their computer where it can be run. For more information, see [Registration-Free COM Interop](#).

Managed Code and Data

Code developed for the .NET Framework is referred to as *managed code*, and contains metadata that is used by the CLR. Data used by .NET Framework applications is called *managed data* because the runtime manages data-related tasks such as allocating and reclaiming memory and performing type checking. By default, Visual Basic .NET uses managed code and data, but you can access the unmanaged code and data of COM objects using interop assemblies (described later on this page).

Assemblies

An assembly is the primary building block of a .NET Framework application. It is a collection of functionality that is built, versioned, and deployed as a single implementation unit containing one or more files. Each assembly contains an assembly manifest.

Type Libraries and Assembly Manifests

Type libraries describe characteristics of COM objects, such as member names and data types. Assembly manifests perform the same function for .NET Framework applications. They include information about the following:

- Assembly identity, version, culture, and digital signature.
- Files that make up the assembly implementation.
- Types and resources that make up the assembly. This includes those that are exported from it.
- Compile-time dependencies on other assemblies.
- Permissions required for the assembly to run correctly.

For more information about assemblies and assembly manifests, see [Assemblies in .NET](#).

Importing and Exporting Type Libraries

Visual Studio contains a utility, Tlbimp, that lets you import information from a type library into a .NET Framework application. You can generate type libraries from assemblies by using the Tlbexp utility.

For information about Tlbimp and Tlbexp, see [Tlbimp.exe \(Type Library Importer\)](#) and [Tlbexp.exe \(Type Library Exporter\)](#).

Interop Assemblies

Interop assemblies are .NET Framework assemblies that bridge between managed and unmanaged code, mapping COM object members to equivalent .NET Framework managed members. Interop assemblies created by Visual Basic .NET handle many of the details of working with COM objects, such as interoperability marshaling.

Interoperability Marshaling

All .NET Framework applications share a set of common types that enable interoperability of objects, regardless of the programming language that is used. The parameters and return values of COM objects sometimes use data types that differ from those used in managed code. *Interoperability marshaling* is the process of packaging parameters and return values into equivalent data types as they move to and from COM objects. For more information, see [Interop Marshaling](#).

See also

- [COM Interop](#)
- [Walkthrough: Implementing Inheritance with COM Objects](#)
- [Interoperating with Unmanaged Code](#)
- [Troubleshooting Interoperability](#)
- [Assemblies in .NET](#)
- [Tlbimp.exe \(Type Library Importer\)](#)
- [Tlbexp.exe \(Type Library Exporter\)](#)
- [Interop Marshaling](#)
- [Registration-Free COM Interop](#)

How to: Reference COM Objects from Visual Basic

10/17/2019 • 2 minutes to read • [Edit Online](#)

In Visual Basic, adding references to COM objects that have type libraries requires the creation of an interop assembly for the COM library. References to the members of the COM object are routed to the interop assembly and then forwarded to the actual COM object. Responses from the COM object are routed to the interop assembly and forwarded to your .NET Framework application.

You can reference a COM object without using an interop assembly by embedding the type information for the COM object in a .NET assembly. To embed type information, set the `Embed Interop Types` property to `True` for the reference to the COM object. If you are compiling by using the command-line compiler, use the `/link` option to reference the COM library. For more information, see [-link \(Visual Basic\)](#).

Visual Basic automatically creates interop assemblies when you add a reference to a type library from the integrated development environment (IDE). When working from the command line, you can use the Tlbimp utility to manually create interop assemblies.

To add references to COM objects

1. On the **Project** menu, choose **Add Reference** and then click the **COM** tab in the dialog box.
2. Select the component you want to use from the list of COM objects.
3. To simplify access to the interop assembly, add an `Imports` statement to the top of the class or module in which you will use the COM object. For example, the following code example imports the namespace `INKEDLib` for objects referenced in the `Microsoft InkEdit Control 1.0` library.

```
Imports INKEDLib

Class Sample
    Private s As IInkCursor

End Class
```

To create an interop assembly using Tlbimp

1. Add the location of Tlbimp to the search path, if it is not already part of the search path and you are not currently in the directory where it is located.
2. Call Tlbimp from a command prompt, providing the following information:
 - Name and location of the DLL that contains the type library
 - Name and location of the namespace where the information should be placed
 - Name and location of the target interop assembly

The following code provides an example:

```
Tlbimp test3.dll /out:NameSpace1 /out:Interop1.dll
```

You can use Tlbimp to create interop assemblies for type libraries, even for unregistered COM objects. However, the COM objects referred to by interop assemblies must be properly registered on the computer where they are to be used. You can register a COM object by using the Regsvr32 utility included with the Windows operating system.

See also

- [COM Interop](#)
- [Tlbimp.exe \(Type Library Importer\)](#)
- [Tlbexp.exe \(Type Library Exporter\)](#)
- [Walkthrough: Implementing Inheritance with COM Objects](#)
- [Troubleshooting Interoperability](#)
- [Imports Statement \(.NET Namespace and Type\)](#)

How to: Work with ActiveX Controls (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

ActiveX controls are COM components or objects you can insert into a Web page or other application to reuse packaged functionality someone else has programmed. You can use ActiveX controls developed for Visual Basic 6.0 and earlier versions to add features to the **Toolbox** of Visual Studio.

To add ActiveX controls to the toolbox

1. On the **Tools** menu, click **Choose Toolbox Items**.

The **Choose Toolbox** dialog box appears.

2. Click the **COM Components** tab.
3. Select the check box next to the ActiveX control you want to use, and then click **OK**.

The new control appears with the other tools in the **Toolbox**.

NOTE

You can use the Aximp utility to manually create an interop assembly for ActiveX controls. For more information, see [Aximp.exe \(Windows Forms ActiveX Control Importer\)](#).

See also

- [COM Interop](#)
- [How to: Add ActiveX Controls to Windows Forms](#)
- [Aximp.exe \(Windows Forms ActiveX Control Importer\)](#)
- [Considerations When Hosting an ActiveX Control on a Windows Form](#)
- [Troubleshooting Interoperability](#)

Walkthrough: Calling Windows APIs (Visual Basic)

8/22/2019 • 9 minutes to read • [Edit Online](#)

Windows APIs are dynamic-link libraries (DLLs) that are part of the Windows operating system. You use them to perform tasks when it is difficult to write equivalent procedures of your own. For example, Windows provides a function named `FlashWindowEx` that lets you make the title bar for an application alternate between light and dark shades.

The advantage of using Windows APIs in your code is that they can save development time because they contain dozens of useful functions that are already written and waiting to be used. The disadvantage is that Windows APIs can be difficult to work with and unforgiving when things go wrong.

Windows APIs represent a special category of interoperability. Windows APIs do not use managed code, do not have built-in type libraries, and use data types that are different than those used with Visual Studio. Because of these differences, and because Windows APIs are not COM objects, interoperability with Windows APIs and the .NET Framework is performed using platform invoke, or PInvoke. Platform invoke is a service that enables managed code to call unmanaged functions implemented in DLLs. For more information, see [Consuming Unmanaged DLL Functions](#). You can use PInvoke in Visual Basic by using either the `Declare` statement or applying the `DllImport` attribute to an empty procedure.

Windows API calls were an important part of Visual Basic programming in the past, but are seldom necessary with Visual Basic .NET. Whenever possible, you should use managed code from the .NET Framework to perform tasks, instead of Windows API calls. This walkthrough provides information for those situations in which using Windows APIs is necessary.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

API Calls Using Declare

The most common way to call Windows APIs is by using the `Declare` statement.

To declare a DLL procedure

1. Determine the name of the function you want to call, plus its arguments, argument types, and return value, as well as the name and location of the DLL that contains it.

NOTE

For complete information about the Windows APIs, see the Win32 SDK documentation in the Platform SDK Windows API. For more information about the constants that Windows APIs use, examine the header files such as Windows.h included with the Platform SDK.

2. Open a new Windows Application project by clicking **New** on the **File** menu, and then clicking **Project**.

The **New Project** dialog box appears.

3. Select **Windows Application** from the list of Visual Basic project templates. The new project is displayed.

4. Add the following `Declare` function either to the class or module in which you want to use the DLL:

```
Declare Auto Function MBox Lib "user32.dll" Alias "MessageBox" (
    ByVal hWnd As Integer,
    ByVal txt As String,
    ByVal caption As String,
    ByVal Typ As Integer) As Integer
```

Parts of the Declare Statement

The `Declare` statement includes the following elements.

Auto modifier

The `Auto` modifier instructs the runtime to convert the string based on the method name according to common language runtime rules (or alias name if specified).

Lib and Alias keywords

The name following the `Function` keyword is the name your program uses to access the imported function. It can be the same as the real name of the function you are calling, or you can use any valid procedure name and then employ the `Alias` keyword to specify the real name of the function you are calling.

Specify the `Lib` keyword, followed by the name and location of the DLL that contains the function you are calling. You do not need to specify the path for files located in the Windows system directories.

Use the `Alias` keyword if the name of the function you are calling is not a valid Visual Basic procedure name, or conflicts with the name of other items in your application. `Alias` indicates the true name of the function being called.

Argument and Data Type Declarations

Declare the arguments and their data types. This part can be challenging because the data types that Windows uses do not correspond to Visual Studio data types. Visual Basic does a lot of the work for you by converting arguments to compatible data types, a process called *marshaling*. You can explicitly control how arguments are marshaled by using the `MarshalAsAttribute` attribute defined in the `System.Runtime.InteropServices` namespace.

NOTE

Previous versions of Visual Basic allowed you to declare parameters `As Any`, meaning that data of any data type could be used. Visual Basic requires that you use a specific data type for all `Declare` statements.

Windows API Constants

Some arguments are combinations of constants. For example, the `MessageBox` API shown in this walkthrough accepts an integer argument called `Typ` that controls how the message box is displayed. You can determine the numeric value of these constants by examining the `#define` statements in the file WinUser.h. The numeric values are generally shown in hexadecimal, so you may want to use a calculator to add them and convert to decimal. For example, if you want to combine the constants for the exclamation style `MB_ICONEXCLAMATION` 0x00000030 and the Yes/No style `MB_YESNO` 0x00000004, you can add the numbers and get a result of 0x00000034, or 52 decimal. Although you can use the decimal result directly, it is better to declare these values as constants in your application and combine them using the `or` operator.

To declare constants for Windows API calls

1. Consult the documentation for the Windows function you are calling. Determine the name of the constants it uses and the name of the .h file that contains the numeric values for these constants.
2. Use a text editor, such as Notepad, to view the contents of the header (.h) file, and find the values associated with the constants you are using. For example, the `MessageBox` API uses the constant `MB_ICONQUESTION` to show a question mark in the message box. The definition for `MB_ICONQUESTION` is in WinUser.h and appears

as follows:

```
#define MB_ICONQUESTION 0x00000020L
```

3. Add equivalent `Const` statements to your class or module to make these constants available to your application. For example:

```
Const MB_ICONQUESTION As Integer = &H20
Const MB_YESNO As Integer = &H4
Const IDYES As Integer = 6
Const IDNO As Integer = 7
```

To call the DLL procedure

1. Add a button named `Button1` to the startup form for your project, and then double-click it to view its code. The event handler for the button is displayed.
2. Add code to the `Click` event handler for the button you added, to call the procedure and provide the appropriate arguments:

```
Private Sub Button1_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles Button1.Click

    ' Stores the return value.
    Dim RetVal As Integer
    RetVal = MBox(0, "Declare DLL Test", "Windows API MessageBox",
        MB_ICONQUESTION Or MB_YESNO)

    ' Check the return value.
    If RetVal = IDYES Then
        MsgBox("You chose Yes")
    Else
        MsgBox("You chose No")
    End If
End Sub
```

3. Run the project by pressing F5. The message box is displayed with both **Yes** and **No** response buttons. Click either one.

Data Marshaling

Visual Basic automatically converts the data types of parameters and return values for Windows API calls, but you can use the `MarshalAs` attribute to explicitly specify unmanaged data types that an API expects. For more information about interop marshaling, see [Interop Marshaling](#).

To use `Declare` and `MarshalAs` in an API call

1. Determine the name of the function you want to call, plus its arguments, data types, and return value.
2. To simplify access to the `MarshalAs` attribute, add an `Imports` statement to the top of the code for the class or module, as in the following example:

```
Imports System.Runtime.InteropServices
```

3. Add a function prototype for the imported function to the class or module you are using, and apply the `MarshalAs` attribute to the parameters or return value. In the following example, an API call that expects the type `void*` is marshaled as `AsAny`:

```
Declare Sub SetData Lib "..\LIB\UnmgdLib.dll" (
    ByVal x As Short,
    <MarshalAsAttribute(UnmanagedType.AsAny)>
    ByVal o As Object)
```

API Calls Using DllImport

The `DllImport` attribute provides a second way to call functions in DLLs without type libraries. `DllImport` is roughly equivalent to using a `Declare` statement but provides more control over how functions are called.

You can use `DllImport` with most Windows API calls as long as the call refers to a shared (sometimes called *static*) method. You cannot use methods that require an instance of a class. Unlike `Declare` statements, `DllImport` calls cannot use the `MarshalAs` attribute.

To call a Windows API using the `DllImport` attribute

1. Open a new Windows Application project by clicking **New** on the **File** menu, and then clicking **Project**.
The **New Project** dialog box appears.
2. Select **Windows Application** from the list of Visual Basic project templates. The new project is displayed.
3. Add a button named `Button2` to the startup form.
4. Double-click `Button2` to open the code view for the form.
5. To simplify access to `DllImport`, add an `Imports` statement to the top of the code for the startup form class:

```
Imports System.Runtime.InteropServices
```

6. Declare an empty function preceding the `End Class` statement for the form, and name the function `MoveFile`.

7. Apply the `Public` and `Shared` modifiers to the function declaration and set parameters for `MoveFile` based on the arguments the Windows API function uses:

```
Public Shared Function MoveFile(
    ByVal src As String,
    ByVal dst As String) As Boolean
    ' Leave the body of the function empty.
End Function
```

Your function can have any valid procedure name; the `DllImport` attribute specifies the name in the DLL. It also handles interoperability marshaling for the parameters and return values, so you can choose Visual Studio data types that are similar to the data types the API uses.

8. Apply the `DllImport` attribute to the empty function. The first parameter is the name and location of the DLL containing the function you are calling. You do not need to specify the path for files located in the Windows system directories. The second parameter is a named argument that specifies the name of the function in the Windows API. In this example, the `DllImport` attribute forces calls to `MoveFile` to be forwarded to `MoveFileW` in KERNEL32.DLL. The `MoveFileW` method copies a file from the path `src` to the path `dst`.

```
<DllImport("KERNEL32.DLL", EntryPoint:="MoveFileW", SetLastError:=True,
    CharSet:=CharSet.Unicode, ExactSpelling:=True,
    CallingConvention:=CallingConvention.StdCall)>
Public Shared Function MoveFile(
    ByVal src As String,
    ByVal dst As String) As Boolean
    ' Leave the body of the function empty.
End Function
```

9. Add code to the `Button2_Click` event handler to call the function:

```
Private Sub Button2_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles Button2.Click

    Dim RetVal As Boolean = MoveFile("c:\tmp\Test.txt", "c:\Test.txt")
    If RetVal = True Then
        MsgBox("The file was moved successfully.")
    Else
        MsgBox("The file could not be moved.")
    End If
End Sub
```

10. Create a file named Test.txt and place it in the C:\Tmp directory on your hard drive. Create the Tmp directory if necessary.
11. Press F5 to start the application. The main form appears.
12. Click **Button2**. The message "The file was moved successfully" is displayed if the file can be moved.

See also

- [DllImportAttribute](#)
- [MarshalAsAttribute](#)
- [Declare Statement](#)
- [Auto](#)
- [Alias](#)
- [COM Interop](#)
- [Creating Prototypes in Managed Code](#)
- [Marshaling a Delegate as a Callback Method](#)

How to: Call Windows APIs (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

This example defines and calls the `MessageBox` function in user32.dll and then passes a string to it.

Example

```
' Defines the MessageBox function.  
Public Class Win32  
    Declare Auto Function MessageBox Lib "user32.dll" (  
        ByVal hWnd As Integer, ByVal txt As String,  
        ByVal caption As String, ByVal Type As Integer  
    ) As Integer  
End Class  
  
' Calls the MessageBox function.  
Public Class DemoMessageBox  
    Public Shared Sub Main()  
        Win32.MessageBox(0, "Here's a MessageBox", "Platform Invoke Sample", 0)  
    End Sub  
End Class
```

Compiling the Code

This example requires:

- A reference to the `System` namespace.

Robust Programming

The following conditions may cause an exception:

- The method is not static, is abstract, or has been previously defined. The parent type is an interface, or the length of `name` or `dllName` is zero. ([ArgumentException](#))
- The `name` or `dllName` is `Nothing`. ([ArgumentNullException](#))
- The containing type has been previously created using `CreateType`. ([InvalidOperationException](#))

See also

- [A Closer Look at Platform Invoke](#)
- [Platform Invoke Examples](#)
- [Consuming Unmanaged DLL Functions](#)
- [Defining a Method with Reflection Emit](#)
- [Walkthrough: Calling Windows APIs](#)
- [COM Interop](#)

How to: Call a Windows Function that Takes Unsigned Types (Visual Basic)

9/17/2019 • 2 minutes to read • [Edit Online](#)

If you are consuming a class, module, or structure that has members of unsigned integer types, you can access these members with Visual Basic.

To call a Windows function that takes an unsigned type

1. Use a [Declare Statement](#) to tell Visual Basic which library holds the function, what its name is in that library, what its calling sequence is, and how to convert strings when calling it.
2. In the `Declare` statement, use `UInteger`, `ULong`, `UShort`, or `Byte` as appropriate for each parameter with an unsigned type.
3. Consult the documentation for the Windows function you are calling to find the names and values of the constants it uses. Many of these are defined in the WinUser.h file.
4. Declare the necessary constants in your code. Many Windows constants are 32-bit unsigned values, and you should declare these `As UInteger`.
5. Call the function in the normal way. The following example calls the Windows function `MessageBox`, which takes an unsigned integer argument.

```
Public Class windowsMessage
    Private Declare Auto Function mb Lib "user32.dll" Alias "MessageBox" (
        ByVal hWnd As Integer,
        ByVal lpText As String,
        ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer
    Private Const MB_OK As UInteger = 0
    Private Const MB_ICONEXCLAMATION As UInteger = &H30
    Private Const IDOK As UInteger = 1
    Private Const IDCLOSE As UInteger = 8
    Private Const c As UInteger = MB_OK Or MB_ICONEXCLAMATION
    Public Function messageThroughWindows() As String
        Dim r As Integer = mb(0, "Click OK if you see this!",
            "Windows API call", c)
        Dim s As String = "Windows API MessageBox returned " &
            CStr(r)& vbCrLf & "(IDOK = " & CStr(IDOK) &
            ", IDCLOSE = " & CStr(IDCLOSE) & ")"
        Return s
    End Function
End Class
```

You can test the function `messageThroughWindows` with the following code.

```
Public Sub consumeWindowsMessage()
    Dim w As New windowsMessage
    w.messageThroughWindows()
End Sub
```

Caution

The `UInteger`, `ULong`, `UShort`, and `SByte` data types are not part of the [Language Independence and](#)

Language-Independent Components (CLS), so CLS-compliant code cannot consume a component that uses them.

IMPORTANT

Making a call to unmanaged code, such as the Windows application programming interface (API), exposes your code to potential security risks.

IMPORTANT

Calling the Windows API requires unmanaged code permission, which might affect its execution in partial-trust situations. For more information, see [SecurityPermission](#) and [Code Access Permissions](#).

See also

- [Data Types](#)
- [Integer Data Type](#)
- [UInteger Data Type](#)
- [Declare Statement](#)
- [Walkthrough: Calling Windows APIs](#)

Walkthrough: Creating COM Objects with Visual Basic

8/22/2019 • 4 minutes to read • [Edit Online](#)

When creating new applications or components, it is best to create .NET Framework assemblies. However, Visual Basic also makes it easy to expose a .NET Framework component to COM. This enables you to provide new components for earlier application suites that require COM components. This walkthrough demonstrates how to use Visual Basic to expose .NET Framework objects as COM objects, both with and without the COM class template.

The easiest way to expose COM objects is by using the COM class template. The COM class template creates a new class, and then configures your project to generate the class and interoperability layer as a COM object and register it with the operating system.

NOTE

Although you can also expose a class created in Visual Basic as a COM object for unmanaged code to use, it is not a true COM object and cannot be used by Visual Basic. For more information, see [COM Interoperability in .NET Framework Applications](#).

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

To create a COM object by using the COM class template

1. Open a new Windows Application project from the **File** menu by clicking **New Project**.
2. In the **New Project** dialog box under the **Project Types** field, check that Windows is selected. Select **Class Library** from the **Templates** list, and then click **OK**. The new project is displayed.
3. Select **Add New Item** from the **Project** menu. The **Add New Item** dialog box is displayed.
4. Select **COM Class** from the **Templates** list, and then click **Add**. Visual Basic adds a new class and configures the new project for COM interop.
5. Add code such as properties, methods, and events to the COM class.
6. Select **Build ClassLibrary1** from the **Build** menu. Visual Basic builds the assembly and registers the COM object with the operating system.

Creating COM Objects without the COM Class Template

You can also create a COM class manually instead of using the COM class template. This procedure is helpful when you are working from the command line or when you want more control over how COM objects are defined.

To set up your project to generate a COM object

1. Open a new Windows Application project from the **File** menu by clicking **NewProject**.
2. In the **New Project** dialog box under the **Project Types** field, check that Windows is selected. Select **Class**

Library from the **Templates** list, and then click **OK**. The new project is displayed.

3. In **Solution Explorer**, right-click your project, and then click **Properties**. The **Project Designer** is displayed.
4. Click the **Compile** tab.
5. Select the **Register for COM Interop** check box.

To set up the code in your class to create a COM object

1. In **Solution Explorer**, double-click **Class1.vb** to display its code.
2. Rename the class to `ComClass1`.
3. Add the following constants to `ComClass1`. They will store the Globally Unique Identifier (GUID) constants that the COM objects are required to have.

```
Public Const ClassId As String = ""  
Public Const InterfaceId As String = ""  
Public Const EventsId As String = ""
```

4. On the **Tools** menu, click **Create Guid**. In the **Create GUID** dialog box, click **Registry Format** and then click **Copy**. Click **Exit**.
5. Replace the empty string for the `classId` with the GUID, removing the leading and trailing braces. For example, if the GUID provided by Guidgen is `"{2C8B0AEE-02C9-486e-B809-C780A11530FE}"` then your code should appear as follows.

```
Public Const ClassId As String = "2C8B0AEE-02C9-486e-B809-C780A11530FE"
```

6. Repeat the previous steps for the `InterfaceId` and `EventsId` constants, as in the following example.

```
Public Const InterfaceId As String = "3D8B5BA4-FB8C-5ff8-8468-11BF6BD5CF91"  
Public Const EventsId As String = "2B691787-6ED7-401e-90A4-B3B9C0360E31"
```

NOTE

Make sure that the GUIDs are new and unique; otherwise, your COM component could conflict with other COM components.

7. Add the `ComClass` attribute to `ComClass1`, specifying the GUIDs for the Class ID, Interface ID, and Events ID as in the following example:

```
<ComClass(ComClass1.ClassId, ComClass1.InterfaceId, ComClass1.EventsId)>  
Public Class ComClass1
```

8. COM classes must have a parameterless `Public Sub New()` constructor, or the class will not register correctly. Add a parameterless constructor to the class:

```
Public Sub New()  
    MyBase.New()  
End Sub
```

9. Add properties, methods, and events to the class, ending it with an `End Class` statement. Select **Build**

Solution from the **Build** menu. Visual Basic builds the assembly and registers the COM object with the operating system.

NOTE

The COM objects you generate with Visual Basic cannot be used by other Visual Basic applications because they are not true COM objects. Attempts to add references to such COM objects will raise an error. For details, see [COM Interoperability in .NET Framework Applications](#).

See also

- [ComClassAttribute](#)
- [COM Interop](#)
- [Walkthrough: Implementing Inheritance with COM Objects](#)
- [#Region Directive](#)
- [COM Interoperability in .NET Framework Applications](#)
- [Troubleshooting Interoperability](#)

Troubleshooting Interoperability (Visual Basic)

8/22/2019 • 9 minutes to read • [Edit Online](#)

When you interoperate between COM and the managed code of the .NET Framework, you may encounter one or more of the following common issues.

Interop Marshaling

At times, you may have to use data types that are not part of the .NET Framework. Interop assemblies handle most of the work for COM objects, but you may have to control the data types that are used when managed objects are exposed to COM. For example, structures in class libraries must specify the `BStr` unmanaged type on strings sent to COM objects created by Visual Basic 6.0 and earlier versions. In such cases, you can use the [MarshalAsAttribute](#) attribute to cause managed types to be exposed as unmanaged types.

Exporting Fixed-Length Strings to Unmanaged Code

In Visual Basic 6.0 and earlier versions, strings are exported to COM objects as sequences of bytes without a null termination character. For compatibility with other languages, Visual Basic .NET includes a termination character when exporting strings. The best way to address this incompatibility is to export strings that lack the termination character as arrays of `Byte` or `Char`.

Exporting Inheritance Hierarchies

Managed class hierarchies flatten out when exposed as COM objects. For example, if you define a base class with a member, and then inherit the base class in a derived class that is exposed as a COM object, clients that use the derived class in the COM object will not be able to use the inherited members. Base class members can be accessed from COM objects only as instances of a base class, and then only if the base class is also created as a COM object.

Overloaded Methods

Although you can create overloaded methods with Visual Basic, they are not supported by COM. When a class that contains overloaded methods is exposed as a COM object, new method names are generated for the overloaded methods.

For example, consider a class that has two overloads of the `Synch` method. When the class is exposed as a COM object, the new generated method names could be `Synch` and `Synch_2`.

The renaming can cause two problems for consumers of the COM object.

1. Clients might not expect the generated method names.
2. The generated method names in the class exposed as a COM object can change when new overloads are added to the class or its base class. This can cause versioning problems.

To solve both problems, give each method a unique name, instead of using overloading, when you develop objects that will be exposed as COM objects.

Use of COM Objects Through Interop Assemblies

You use interop assemblies almost as if they are managed code replacements for the COM objects they represent.

However, because they are wrappers and not actual COM objects, there are some differences between using interop assemblies and standard assemblies. These areas of difference include the exposure of classes, and data types for parameters and return values.

Classes Exposed as Both Interfaces and Classes

Unlike classes in standard assemblies, COM classes are exposed in interop assemblies as both an interface and a class that represents the COM class. The interface's name is identical to that of the COM class. The name of the interop class is the same as that of the original COM class, but with the word "Class" appended. For example, suppose you have a project with a reference to an interop assembly for a COM object. If the COM class is named `MyComClass`, IntelliSense and the Object Browser show an interface named `MyComClass` and a class named `MyComClassClass`.

Creating Instances of a .NET Framework Class

Generally, you create an instance of a .NET Framework class using the `New` statement with a class name. Having a COM class represented by an interop assembly is the one case in which you can use the `New` statement with an interface. Unless you are using the COM class with an `Inherits` statement, you can use the interface just as you would a class. The following code demonstrates how to create a `Command` object in a project that has a reference to the Microsoft ActiveX Data Objects 2.8 Library COM object:

```
Dim cmd As New ADODB.Command
```

However, if you are using the COM class as the base for a derived class, you must use the interop class that represents the COM class, as in the following code:

```
Class DerivedCommand  
    Inherits ADODB.CommandClass  
End Class
```

NOTE

Interop assemblies implicitly implement interfaces that represent COM classes. You should not try to use the `Implements` statement to implement these interfaces or an error will result.

Data Types for Parameters and Return Values

Unlike members of standard assemblies, interop assembly members may have data types that differ from those used in the original object declaration. Although interop assemblies implicitly convert COM types to compatible common language runtime types, you should pay attention to the data types that are used by both sides to prevent runtime errors. For example, in COM objects created in Visual Basic 6.0 and earlier versions, values of type `Integer` assume the .NET Framework equivalent type, `Short`. It is recommended that you use the Object Browser to examine the characteristics of imported members before you use them.

Module level COM methods

Most COM objects are used by creating an instance of a COM class using the `New` keyword and then calling methods of the object. One exception to this rule involves COM objects that contain `AppObj` or `GlobalMultiUse` COM classes. Such classes resemble module level methods in Visual Basic .NET classes. Visual Basic 6.0 and earlier versions implicitly create instances of such objects for you the first time that you call one of their methods. For example, in Visual Basic 6.0 you can add a reference to the Microsoft DAO 3.6 Object Library and call the

`DBEngine` method without first creating an instance:

```
Dim db As DAO.Database
' Open the database.
Set db = DBEngine.OpenDatabase("C:\nwind.mdb")
' Use the database object.
```

Visual Basic .NET requires that you always create instances of COM objects before you can use their methods. To use these methods in Visual Basic, declare a variable of the desired class and use the new keyword to assign the object to the object variable. The `Shared` keyword can be used when you want to make sure that only one instance of the class is created.

```
' Class level variable.
Shared DBEngine As New DAO.DBEngine

Sub DAOOpenRecordset()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim fld As DAO.Field
    ' Open the database.
    db = DBEngine.OpenDatabase("C:\nwind.mdb")

    ' Open the Recordset.
    rst = db.OpenRecordset(
        "SELECT * FROM Customers WHERE Region = 'WA'", 
        DAO.RecordsetTypeEnum.dbOpenForwardOnly,
        DAO.RecordsetOptionEnum.dbReadOnly)
    ' Print the values for the fields in the debug window.
    For Each fld In rst.Fields
        Debug.WriteLine(fld.Value.ToString & ";")
    Next
    Debug.WriteLine("")
    ' Close the Recordset.
    rst.Close()
End Sub
```

Unhandled Errors in Event Handlers

One common interop problem involves errors in event handlers that handle events raised by COM objects. Such errors are ignored unless you specifically check for errors using `On Error` or `Try...Catch...Finally` statements.

For example, the following example is from a Visual Basic .NET project that has a reference to the Microsoft ActiveX Data Objects 2.8 Library COM object.

```

' To use this example, add a reference to the
'   Microsoft ActiveX Data Objects 2.8 Library
' from the COM tab of the project references page.
Dim WithEvents cn As New ADODB.Connection
Sub ADODBConnect()
    cn.ConnectionString =
        "Provider=Microsoft.Jet.OLEDB.4.0;" &
        "Data Source=C:\NWIND.MDB"
    cn.Open()
    MsgBox(cn.ConnectionString)
End Sub

Private Sub Form1_Load(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles MyBase.Load

    ADODBConnect()
End Sub

Private Sub cn_ConnectComplete(
    ByVal pError As ADODB.Error,
    ByRef adStatus As ADODB.EventStatusEnum,
    ByVal pConnection As ADODB.Connection) Handles cn.ConnectComplete

    ' This is the event handler for the cn_ConnectComplete event raised
    ' by the ADODB.Connection object when a database is opened.
    Dim x As Integer = 6
    Dim y As Integer = 0
    Try
        x = CInt(x / y) ' Attempt to divide by zero.
        ' This procedure would fail silently without exception handling.
    Catch ex As Exception
        MsgBox("There was an error: " & ex.Message)
    End Try
End Sub

```

This example raises an error as expected. However, if you try the same example without the `Try...Catch...Finally` block, the error is ignored as if you used the `On Error Resume Next` statement. Without error handling, the division by zero silently fails. Because such errors never raise unhandled exception errors, it is important that you use some form of exception handling in event handlers that handle events from COM objects.

Understanding COM interop errors

Without error handling, interop calls often generate errors that provide little information. Whenever possible, use structured error handling to provide more information about problems when they occur. This can be especially helpful when you debug applications. For example:

```

Try
    ' Place call to COM object here.
Catch ex As Exception
    ' Display information about the failed call.
End Try

```

You can find information such as the error description, HRESULT, and the source of COM errors by examining the contents of the exception object.

ActiveX Control Issues

Most ActiveX controls that work with Visual Basic 6.0 work with Visual Basic .NET without trouble. The main exceptions are container controls, or controls that visually contain other controls. Some examples of older controls that do not work correctly with Visual Studio are as follows:

- Microsoft Forms 2.0 Frame control
- Up-Down control, also known as the spin control
- Sheridan Tab Control

There are only a few workarounds for unsupported ActiveX control problems. You can migrate existing controls to Visual Studio if you own the original source code. Otherwise, you can check with software vendors for updated .NET-compatible versions of controls to replace unsupported ActiveX controls.

Passing ReadOnly Properties of Controls ByRef

Visual Basic .NET sometimes raises COM errors such as, "Error 0x800A017F CTL_E_SETNOTSUPPORTED", when you pass `ReadOnly` properties of some older ActiveX controls as `ByRef` parameters to other procedures. Similar procedure calls from Visual Basic 6.0 do not raise an error, and the parameters are treated as if you passed them by value. The Visual Basic .NET error message indicates that you are trying to change a property that does not have a property `Set` procedure.

If you have access to the procedure being called, you can prevent this error by using the `ByVal` keyword to declare parameters that accept `ReadOnly` properties. For example:

```
Sub ProcessParams(ByVal c As Object)
    'Use the arguments here.
End Sub
```

If you do not have access to the source code for the procedure being called, you can force the property to be passed by value by adding an extra set of brackets around the calling procedure. For example, in a project that has a reference to the Microsoft ActiveX Data Objects 2.8 Library COM object, you can use:

```
Sub PassByVal(ByVal pError As ADODB.Error)
    ' The extra set of parentheses around the arguments
    ' forces them to be passed by value.
    ProcessParams((pError.Description))
End Sub
```

Deploying Assemblies That Expose Interop

Deploying assemblies that expose COM interfaces presents some unique challenges. For example, a potential problem occurs when separate applications reference the same COM assembly. This situation is common when a new version of an assembly is installed and another application is still using the old version of the assembly. If you uninstall an assembly that shares a DLL, you can unintentionally make it unavailable to the other assemblies.

To avoid this problem, you should install shared assemblies to the Global Assembly Cache (GAC) and use a MergeModule for the component. If you cannot install the application in the GAC, it should be installed to CommonFilesFolder in a version-specific subdirectory.

Assemblies that are not shared should be located side by side in the directory with the calling application.

See also

- [MarshalAsAttribute](#)
- [COM Interop](#)
- [Tlbimp.exe \(Type Library Importer\)](#)
- [Tlbexp.exe \(Type Library Exporter\)](#)
- [Walkthrough: Implementing Inheritance with COM Objects](#)

- [Inherits Statement](#)
- [Global Assembly Cache](#)

COM Interoperability in .NET Framework Applications (Visual Basic)

7/30/2019 • 2 minutes to read • [Edit Online](#)

When you want to use COM objects and .NET Framework objects in the same application, you need to address the differences in how the objects exist in memory. A .NET Framework object is located in managed memory—the memory controlled by the common language runtime—and may be moved by the runtime as needed. A COM object is located in unmanaged memory and is not expected to move to another memory location. Visual Studio and the .NET Framework provide tools to control the interaction of these managed and unmanaged components. For more information about managed code, see [Common Language Runtime](#).

In addition to using COM objects in .NET applications, you may also want to use Visual Basic to develop objects accessible from unmanaged code through COM.

The links on this page provide details on the interactions between COM and .NET Framework objects.

Related sections

COM Interop	Provides links to topics covering COM interoperability in Visual Basic, including COM objects, ActiveX controls, Win32 DLLs, managed objects, and inheritance of COM objects.
Interoperating with Unmanaged Code	Briefly describes some of the interaction issues between managed and unmanaged code, and provides links for further study.
COM Wrappers	Discusses runtime callable wrappers, which allow managed code to call COM methods, and COM callable wrappers, which allow COM clients to call .NET object methods.
Advanced COM Interoperability	Provides links to topics covering COM interoperability with respect to wrappers, exceptions, inheritance, threading, events, conversions, and marshaling.
Tlbimp.exe (Type Library Importer)	Discusses the tool you can use to convert the type definitions found within a COM type library into equivalent definitions in a common language runtime assembly.

Walkthrough: Implementing Inheritance with COM Objects (Visual Basic)

9/17/2019 • 5 minutes to read • [Edit Online](#)

You can derive Visual Basic classes from `Public` classes in COM objects, even those created in earlier versions of Visual Basic. The properties and methods of classes inherited from COM objects can be overridden or overloaded just as properties and methods of any other base class can be overridden or overloaded. Inheritance from COM objects is useful when you have an existing class library that you do not want to recompile.

The following procedure shows how to use Visual Basic 6.0 to create a COM object that contains a class, and then use it as a base class.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

To build the COM object that is used in this walkthrough

1. In Visual Basic 6.0, open a new ActiveX DLL project. A project named `Project1` is created. It has a class named `Class1`.
2. In the **Project Explorer**, right-click **Project1**, and then click **Project1 Properties**. The **Project Properties** dialog box is displayed.
3. On the **General** tab of the **Project Properties** dialog box, change the project name by typing `ComObject1` in the **Project Name** field.
4. In the **Project Explorer**, right-click `Class1`, and then click **Properties**. The **Properties** window for the class is displayed.
5. Change the `Name` property to `MathFunctions`.
6. In the **Project Explorer**, right-click `MathFunctions`, and then click **View Code**. The **Code Editor** is displayed.
7. Add a local variable to hold the property value:

```
' Local variable to hold property value  
Private mvarProp1 As Integer
```

8. Add Property `Let` and Property `Get` property procedures:

```

Public Property Let Prop1(ByVal vData As Integer)
    'Used when assigning a value to the property.
    mvarProp1 = vData
End Property
Public Property Get Prop1() As Integer
    'Used when retrieving a property's value.
    Prop1 = mvarProp1
End Property

```

9. Add a function:

```

Function AddNumbers(
    ByVal SomeNumber As Integer,
    ByVal AnotherNumber As Integer) As Integer

    AddNumbers = SomeNumber + AnotherNumber
End Function

```

10. Create and register the COM object by clicking **Make ComObject1.dll** on the **File** menu.

NOTE

Although you can also expose a class created with Visual Basic as a COM object, it is not a true COM object and cannot be used in this walkthrough. For details, see [COM Interoperability in .NET Framework Applications](#).

Interop Assemblies

In the following procedure, you will create an interop assembly, which acts as a bridge between unmanaged code (such as a COM object) and the managed code Visual Studio uses. The interop assembly that Visual Basic creates handles many of the details of working with COM objects, such as *interop marshaling*, the process of packaging parameters and return values into equivalent data types as they move to and from COM objects. The reference in the Visual Basic application points to the interop assembly, not the actual COM object.

To use a COM object with Visual Basic 2005 and later versions

1. Open a new Visual Basic Windows Application project.

2. On the **Project** menu, click **Add Reference**.

The **Add Reference** dialog box is displayed.

3. On the **COM** tab, double-click **ComObject1** in the **Component Name** list and click **OK**.

4. On the **Project** menu, click **Add New Item**.

The **Add New Item** dialog box is displayed.

5. In the **Templates** pane, click **Class**.

The default file name, **Class1.vb**, appears in the **Name** field. Change this field to **MathClass.vb** and click **Add**. This creates a class named **MathClass**, and displays its code.

6. Add the following code to the top of **MathClass** to inherit from the COM class.

```

' The inherited class is called MathFunctions in the base class,
' but the interop assembly appends the word Class to the name.
Inherits ComObject1.MathFunctionsClass

```

7. Overload the public method of the base class by adding the following code to `MathClass` :

```
' This method overloads the method AddNumbers from the base class.  
Overloads Function AddNumbers(  
    ByVal SomeNumber As Integer,  
    ByVal AnotherNumber As Integer) As Integer  
  
    Return SomeNumber + AnotherNumber  
End Function
```

8. Extend the inherited class by adding the following code to `MathClass` :

```
' The following function extends the inherited class.  
Function SubtractNumbers(  
    ByVal SomeNumber As Integer,  
    ByVal AnotherNumber As Integer) As Integer  
  
    Return AnotherNumber - SomeNumber  
End Function
```

The new class inherits the properties of the base class in the COM object, overloads a method, and defines a new method to extend the class.

To test the inherited class

1. Add a button to your startup form, and then double-click it to view its code.

2. In the button's `Click` event handler procedure, add the following code to create an instance of `MathClass` and call the overloaded methods:

```
Dim Result1 As Short  
Dim Result2 As Integer  
Dim Result3 As Integer  
Dim MathObject As New MathClass  
Result1 = MathObject.AddNumbers(4S, 2S) ' Add two Shorts.  
Result2 = MathObject.AddNumbers(4, 2) 'Add two Integers.  
Result3 = MathObject.SubtractNumbers(2, 4) ' Subtract 2 from 4.  
MathObject.Prop1 = 6 ' Set an inherited property.  
  
MsgBox("Calling the AddNumbers method in the base class " &  
    "using Short type numbers 4 and 2 = " & Result1)  
MsgBox("Calling the overloaded AddNumbers method using " &  
    "Integer type numbers 4 and 2 = " & Result2)  
MsgBox("Calling the SubtractNumbers method "  
    "subtracting 2 from 4 = " & Result3)  
MsgBox("The value of the inherited property is " &  
    MathObject.Prop1)
```

3. Run the project by pressing F5.

When you click the button on the form, the `AddNumbers` method is first called with `Short` data type numbers, and Visual Basic chooses the appropriate method from the base class. The second call to `AddNumbers` is directed to the overload method from `MathClass`. The third call calls the `SubtractNumbers` method, which extends the class. The property in the base class is set, and the value is displayed.

Next Steps

You may have noticed that the overloaded `AddNumbers` function appears to have the same data type as the method inherited from the base class of the COM object. This is because the arguments and parameters of the

base class method are defined as 16-bit integers in Visual Basic 6.0, but they are exposed as 16-bit integers of type `Short` in later versions of Visual Basic. The new function accepts 32-bit integers, and overloads the base class function.

When working with COM objects, make sure that you verify the size and data types of parameters. For example, when you are using a COM object that accepts a Visual Basic 6.0 collection object as an argument, you cannot provide a collection from a later version of Visual Basic.

Properties and methods inherited from COM classes can be overridden, meaning that you can declare a local property or method that replaces a property or method inherited from a base COM class. The rules for overriding inherited COM properties are similar to the rules for overriding other properties and methods with the following exceptions:

- If you override any property or method inherited from a COM class, you must override all the other inherited properties and methods.
- Properties that use `ByRef` parameters cannot be overridden.

See also

- [COM Interoperability in .NET Framework Applications](#)
- [Inherits Statement](#)
- [Short Data Type](#)

Visual Basic Language Reference

8/24/2018 • 2 minutes to read • [Edit Online](#)

This section provides reference information for various aspects of the Visual Basic language.

In This Section

[Typographic and Code Conventions](#)

Summarizes the way that keywords, placeholders, and other elements of the language are formatted in the Visual Basic documentation.

[Visual Basic Runtime Library Members](#)

Lists the classes and modules of the [Microsoft.VisualBasic](#) namespace, with links to their member functions, methods, properties, constants, and enumerations.

[Keywords](#)

Lists all Visual Basic keywords and provides links to more information.

[Attributes \(Visual Basic\)](#)

Documents the attributes available in Visual Basic.

[Constants and Enumerations](#)

Documents the constants and enumerations available in Visual Basic.

[Data Types](#)

Documents the data types available in Visual Basic.

[Directives](#)

Documents the compiler directives available in Visual Basic.

[Functions](#)

Documents the run-time functions available in Visual Basic.

[Modifiers](#)

Lists the Visual Basic run-time modifiers and provides links to more information.

[Modules](#)

Documents the modules available in Visual Basic and their members.

[Nothing](#)

Describes the default value of any data type.

[Objects](#)

Documents the objects available in Visual Basic and their members.

[Operators](#)

Documents the operators available in Visual Basic.

[Properties](#)

Documents the properties available in Visual Basic.

[Queries](#)

Provides reference information about using Language-Integrated Query (LINQ) expressions in your code.

[Statements](#)

Documents the declaration and executable statements available in Visual Basic.

[XML Comment Tags](#)

Describes the documentation comments for which IntelliSense is provided in the Visual Basic Code Editor.

[XML Axis Properties](#)

Provides links to information about using XML axis properties to access XML directly in your code.

[XML Literals](#)

Provides links to information about using XML literals to incorporate XML directly in your code.

[Error Messages](#)

Provides a listing of Visual Basic compiler and run-time error messages and help on how to handle them.

Related Sections

[Visual Basic](#)

Provides comprehensive help on all areas of the Visual Basic language.

[Visual Basic Command-Line Compiler](#)

Describes how to use the command-line compiler as an alternative to compiling programs from within the Visual Studio integrated development environment (IDE).

Select the Visual Basic language version

6/20/2019 • 2 minutes to read • [Edit Online](#)

The Visual Basic compiler defaults to the latest major version of the language that has been released. You may choose to compile any project using a new point release of the language. Choosing a newer version of the language enables your project to make use of the latest language features. In other scenarios, you may need to validate that a project compiles cleanly when using an older version of the language.

This capability decouples the decision to install new versions of the SDK and tools in your development environment from the decision to incorporate new language features in a project. You can install the latest SDK and tools on your build machine. Each project can be configured to use a specific version of the language for its build.

There are three ways to set the language version:

- Manually edit your [.vbproj](#) file
- Set the language version [for multiple projects in a subdirectory](#)
- Configure the [-langversion](#) compiler option

Edit the vbproj file

You can set the language version in your [.vbproj](#) file. Add the following element:

```
<PropertyGroup>
  <LangVersion>latest</LangVersion>
</PropertyGroup>
```

The value `latest` uses the latest minor version of the Visual Basic language. Valid values are:

VALUE	MEANING
default	The compiler accepts all valid language syntax from the latest major version that it can support.
9	The compiler accepts only syntax that is included in Visual Basic 9.0 or lower.
10	The compiler accepts only syntax that is included in Visual Basic 10.0 or lower.
11	The compiler accepts only syntax that is included in Visual Basic 11.0 or lower.
12	The compiler accepts only syntax that is included in Visual Basic 12.0 or lower.
14	The compiler accepts only syntax that is included in Visual Basic 14.0 or lower.
15	The compiler accepts only syntax that is included in Visual Basic 15.0 or lower.

VALUE	MEANING
15.3	The compiler accepts only syntax that is included in Visual Basic 15.3 or lower.
15.5	The compiler accepts only syntax that is included in Visual Basic 15.5 or lower.
15.8	The compiler accepts only syntax that is included in Visual Basic 15.8 or lower.
latest	The compiler accepts all valid language syntax that it can support.

The special strings `default` and `latest` resolve to the latest major and minor language versions installed on the build machine, respectively.

Configure multiple projects

You can create a **Directory.build.props** file that contains the `<LangVersion>` element to configure multiple directories. You typically do that in your solution directory. Add the following to a **Directory.build.props** file in your solution directory:

```
<Project>
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
</Project>
```

Now, builds in every subdirectory of the directory containing that file will use Visual Basic version 15.5 syntax. For more information, see the article on [Customize your build](#).

Set the langversion compiler option

You can use the `-langversion` command-line option. For more information, see the article on the [-langversion compiler option](#). You can see a list of the valid values by typing `vbc -langversion:?`.

Typographic and Code Conventions (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic documentation uses the following typographic and code conventions.

Typographic Conventions

EXAMPLE	DESCRIPTION
<code>Sub</code> , <code>If</code> , <code>ChDir</code> , <code>Print</code> , <code>True</code> , <code>Debug</code>	Language-specific keywords and runtime members have initial uppercase letters and are formatted as shown in this example.
SmallProject , ButtonCollection	Words and phrases you are instructed to type are formatted as shown in this example.
Module Statement	Links you can click to go to another Help page are formatted as shown in this example.
<i>object</i> , <i>variableName</i> , <code>argumentList</code>	Placeholders for information that you supply are formatted as shown in this example.
[<i>Shadows</i>], [<i>expressionList</i>]	In syntax, optional items are enclosed in brackets.
{ <code>Public</code> <code>Friend</code> <code>Private</code> }	In syntax, when you must make a choice between two or more items, the items are enclosed in braces and separated by vertical bars. You must select one, and only one, of the items.
[<code>Protected</code> <code>Friend</code>]	In syntax, when you have the option of selecting between two or more items, the items are enclosed in square brackets and separated by vertical bars. You can select any combination of the items, or no item.
{[<code>ByVal</code> <code>ByRef</code>]}	In syntax, when you can select no more than one item, but you can also omit the items completely, the items are enclosed in square brackets surrounded by braces and separated by vertical bars.
<i>memberName1</i> , <i>memberName2</i> , <i>memberName3</i>	Multiple instances of the same placeholder are differentiated by subscripts, as shown in the example.
<i>memberName1</i> ... <i>memberNameN</i>	In syntax, an ellipsis (...) is used to indicate an indefinite number of items of the kind immediately in front of the ellipsis. In code, ellipses signify code omitted for the sake of clarity.
<code>ESC</code> , <code>ENTER</code>	Key names and key sequences on the keyboard appear in all uppercase letters.

EXAMPLE	DESCRIPTION
ALT+F1	When plus signs (+) appear between key names, you must hold down one key while pressing the other. For example, ALT+F1 means hold down the ALT key while pressing the F1 key.

Code Conventions

EXAMPLE	DESCRIPTION
<code>sampleString = "Hello, world!"</code>	Code samples appear in a fixed-pitch font and are formatted as shown in this example.
The previous statement sets the value of <code>sampleString</code> to "Hello, world!"	Code elements in explanatory text appear in a fixed-pitch font, as shown in this example.
<code>' This is a comment.</code> <code>REM This is also a comment.</code>	Code comments are introduced by an apostrophe ('') or the REM keyword.
<code>sampleVar = "This is an " _</code> <code>& "example" _</code> <code>& " of how to continue code."</code>	A space followed by an underscore (_) at the end of a line indicates that the statement continues on the following line.

See also

- [Visual Basic Language Reference](#)
- [Keywords](#)
- [Visual Basic Runtime Library Members](#)
- [Visual Basic Naming Conventions](#)
- [How to: Break and Combine Statements in Code](#)
- [Comments in Code](#)

Visual Basic Runtime Library Members

4/2/2019 • 2 minutes to read • [Edit Online](#)

The `Microsoft.VisualBasic` namespace contains the classes, modules, constants, and enumerations that constitute the Visual Basic runtime library. These library members provide procedures, properties, and constant values you can use in your code. Each module and class represents a particular category of functionality.

Microsoft.VisualBasic.Collection Class

Add	Clear	Contains	Count
GetEnumerator	Item[String]	Remove	

Microsoft.VisualBasic.ComClassAttribute Class

ClassID	EventID	InterfaceID	InterfaceShadows
---------	---------	-------------	------------------

Microsoft.VisualBasic.ControlChars Class

Back	Cr	CrLf	FormFeed
Lf	NewLine	NullChar	Quote
Tab	VerticalTab		

Microsoft.VisualBasic.Constants Class

vbAbort	vbAbortRetryIgnore	vbApplicationModal	vbArchive
vbArray	vbBack	vbBinaryCompare	vbBoolean
vbByte	vbCancel	vbCr	vbCritical
vbCrLf	vbCurrency	vbDate	vbDecimal
vbDefaultButton1	vbDefaultButton2	vbDefaultButton3	vbDirectory
vbDouble	vbEmpty	vbExclamation	vbFalse
vbFirstFourDays	vbFirstFullWeek	vbFirstJan1	vbFormFeed

vbFriday	vbGeneralDate	vbGet	vbHidden
vbHide	vbHiragana	vblgnore	vblnformation
vblnteger	vbKatakana	vbLet	vblf
vblnguisticCasing	vbLong	vbLongDate	vbLongTime
vbLowerCase	vbMaximizedFocus	vbMethod	vbMinimizedFocus
vbmimizedNoFocus	vbMonday	vbmMsgBoxHelp	vbmMsgBoxRight
vbmMsgBoxRtlReading	vbmMsgBoxSetForeground	vbNarrow	vbNewLine
vbNo	vbNormal	vbNormalFocus	vbNormalNoFocus
vbnll	vbnNullChar	vbnNullString	vboObject
vboObjectError	vbOK	vbOKCancel	vbOKOnly
vboProperCase	vbQuestion	vbReadOnly	vbRetry
vboRetryCancel	vbSaturday	vbSet	vbShortDate
vboShortTime	vboSimplifiedChinese	vbSingle	vboString
vboSunday	vboSystem	vboSystemModal	vboTab
vboTextCompare	vbThursday	vboTraditionalChinese	vboTrue
vbtuesday	vbUpperCase	vbUseDefault	vbuUserDefinedType
vbuUseSystem	vbuUseSystemDayOfWeek	vbuVariant	vbuVerticalTab
vbuVolume	vbuWednesday	vbuWide	vbuYes
vbuYesNo	vbuYesNoCancel		

Microsoft.VisualBasic.Conversion Module

ErrorToString	Fix	Hex	Int
Oct	Str	Val	

Microsoft.VisualBasic.DateAndTime Module

DateAdd	DateDiff	DatePart	DateSerial

DateString	DateValue	Day	Hour
Minute	Month	MonthName	Now
Second	TimeOfDay	Timer	TimeSerial
TimeString	TimeValue	Today	Weekday
WeekdayName	Year		

Microsoft.VisualBasic.ErrObject Class

Clear	Description	Erl	GetException
HelpContext	HelpFile	LastDllError	Number
Raise	Raise		

Microsoft.VisualBasic.FileSystem Module

ChDir	ChDrive	CurDir	Dir
EOF	FileAttr	FileClose	FileCopy
FileDateTime	FileGet	FileGetObject	FileLen
FileOpen	FilePut	FilePutObject	FileWidth
FreeFile	GetAttr	Input	InputString
Kill	LineInput	Loc	Lock
LOF	MkDir	Print	PrintLine
Rename	Reset	RmDir	Seek
SetAttr	SPC	TAB	Unlock
Write	WriteLine		

Microsoft.VisualBasic.Financial Module

DDB	FV	IPmt	IRR
MIRR	NPer	NPV	Pmt

PPmt	PV	Rate	SLN
SYD			

Microsoft.VisualBasic.Globals Module

ScriptEngine	ScriptEngineBuildVersion	ScriptEngineMajorVersion	ScriptEngineMinorVersion
--------------	--------------------------	--------------------------	--------------------------

Microsoft.VisualBasic.HideModuleNameAttribute Class

HideModuleNameAttribute			
-------------------------	--	--	--

Microsoft.VisualBasic.Information Module

Erl	Err	IsArray	IsDate
IsDBNull	IsError	IsNothing	IsNumeric
IsReference	LBound	QBColor	RGB
SystemTypeName	TypeName	UBound	VarType
VbTypeName			

Microsoft.VisualBasic.Interaction Module

AppActivate	Beep	CallByName	Choose
Command	CreateObject	DeleteSetting	Environ
GetAllSettings	GetObject	GetSetting	IIf
InputBox	MsgBox	Partition	SaveSetting
Shell	Switch		

Microsoft.VisualBasic.MyGroupCollectionAttribute Class

CreateMethod	DefaultInstanceAlias	DisposeMethod	MyGroupName
--------------	----------------------	---------------	-------------

Microsoft.VisualBasic.Strings Module

Asc	Asc	Chr	ChrW
Filter	Format	FormatCurrency	FormatDateTime
FormatNumber	FormatPercent	GetChar	InStr
InStrRev	Join	LCase	Left
Len	LSet	LTrim	Mid
Replace	Right	RSet	RTrim
Space	Split	StrComp	StrConv
StrDup	StrReverse	Trim	UCase

Microsoft.VisualBasic.VBFixedArrayAttribute Class

Bounds	Length
--------	--------

Microsoft.VisualBasic.VBFixedStringAttribute Class

Length	
--------	--

Microsoft.VisualBasic.VbMath Module

Randomize	Rnd
-----------	-----

Microsoft.VisualBasic Constants and Enumerations

The `Microsoft.VisualBasic` namespace provides constants and enumerations as part of the Visual Basic run-time library. You can use these constant values in your code. Each enumeration represents a particular category of functionality. For more information, see [Constants and Enumerations](#).

See also

- [Constants and Enumerations](#)
- [Keywords](#)

Keywords (Visual Basic)

8/22/2019 • 3 minutes to read • [Edit Online](#)

The following tables list all Visual Basic language keywords.

Reserved Keywords

The following keywords are *reserved*, which means that you cannot use them as names for programming elements such as variables or procedures. You can bypass this restriction by enclosing the name in brackets (`[]`). For more information, see "Escaped Names" in [Declared Element Names](#).

NOTE

We do not recommend that you use escaped names, because it can make your code hard to read, and it can lead to subtle errors that can be difficult to find.

AddHandler	AddressOf	Alias	And
AndAlso	As	Boolean	ByRef
Byte	ByVal	Call	Case
Catch	CBool	CByte	CChar
CDate	CDbl	CDec	Char
CInt	Class Constraint	Class Statement	CLng
CObj	Const	Continue	CSByte
CShort	CSng	CStr	CType
CUInt	CULng	CUShort	Date
Decimal	Declare	Default	Delegate
Dim	DirectCast	Do	Double
Each	Else	ElseIf	End Statement
End <keyword>	EndIf	Enum	Erase
Error	Event	Exit	False
Finally	For (in For...Next)	For Each...Next	Friend
Function	Get	GetType	GetXMLNamespace

Global	GoSub	GoTo	Handles
If	If()	Implements	Implements Statement
Imports (.NET Namespace and Type)	Imports (XML Namespace)	In	In (Generic Modifier)
Inherits	Integer	Interface	Is
IsNot	Let	Lib	Like
Long	Loop	Me	Mod
Module	Module Statement	MustInherit	MustOverride
MyBase	MyClass	Namespace	Narrowing
New Constraint	New Operator	Next	Next (in Resume)
Not	Nothing	NotInheritable	NotOverridable
Object	Of	On	Operator
Option	Optional	Or	OrElse
Out (Generic Modifier)	Overloads	Overridable	Overrides
ParamArray	Partial	Private	Property
Protected	Public	RaiseEvent	ReadOnly
ReDim	REM	RemoveHandler	Resume
Return	SByte	Select	Set
Shadows	Shared	Short	Single
Static	Step	Stop	String
Structure Constraint	Structure Statement	Sub	SyncLock
Then	Throw	To	True
Try	TryCast	TypeOf...Is	UInteger
ULong	UShort	Using	Variant
Wend	When	While	Widening
With	WithEvents	WriteOnly	Xor

#Const	#Else	#ElseIf	#End
#If	=	&	&=
*	*=	/	/=
\	\=	^	^=
+	+ =	-	- =
>> Operator	>> = Operator	<<	<< =

NOTE

`EndIf`, `GoSub`, `Variant`, and `Wend` are retained as reserved keywords, although they are no longer used in Visual Basic. The meaning of the `Let` keyword has changed. `Let` is now used in LINQ queries. For more information, see [Let Clause](#).

Unreserved Keywords

The following keywords are not reserved, which means you can use them as names for your programming elements. However, doing this is not recommended, because it can make your code hard to read and can lead to subtle errors that can be difficult to find.

Aggregate	Ansi	Assembly	Async
Auto	Await	Binary	Compare
Custom	Distinct	Equals	Explicit
From	Group By	Group Join	Into
IsFalse	IsTrue	Iterator	Join
Key	Mid	Off	Order By
Preserve	Skip	Skip While	Strict
Take	Take While	Text	Unicode
Until	Where	Yield	#ExternalSource
#Region			

Related Topics

TITLE	DESCRIPTION
Arrays Summary	Lists language elements that are used to create, define, and use arrays.
Collection Object Summary	Lists language elements that are used for collections.
Control Flow Summary	Lists statements that are used for looping and controlling procedure flow.
Conversion Summary	Lists functions that are used to convert numbers, dates, times, and strings.
Data Types Summary	Lists data types. Also lists functions that are used to convert between data types and verify data types.
Dates and Times Summary	Lists language elements that are used for dates and times.
Declarations and Constants Summary	Lists statements that are used to declare variables, constants, classes, modules, and other programming elements. Also lists language elements that are used to obtain object information, handle events, and implement inheritance.
Directories and Files Summary	Lists functions that are used to control the file system and to process files.
Errors Summary	Lists language elements that are used to catch and return run-time error values.
Financial Summary	Lists functions that are used to perform financial calculations.
Input and Output Summary	Lists functions that are used to read from and write to files, manage files, and print output.
Information and Interaction Summary	Lists functions that are used to run other programs, obtain command-line arguments, manipulate COM objects, retrieve color information, and use control dialog boxes.
Math Summary	Lists functions that are used to perform trigonometric and other mathematical calculations.
My Reference	Lists the objects contained in <code>My</code> , a feature that provides access to frequently used methods, properties, and events of the computer on which the application is running, the current application, the application's resources, the application's settings, and so on.
Operators Summary	Lists assignment and comparison expressions and other operators.
Registry Summary	Lists functions that are used to read, save, and delete program settings.
String Manipulation Summary	Lists functions that are used to manipulate strings.

See also

- [Visual Basic Runtime Library Members](#)

Arrays Summary (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Verify an array.	IsArray
Declare and initialize an array.	Dim , Private , Public , ReDim
Find the limits of an array.	LBound , UBound
Reinitialize an array	Erase , ReDim

See also

- [Keywords](#)
- [Visual Basic Runtime Library Members](#)

Collection Object Summary (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Create a <code>Collection</code> object.	<code>Collection</code>
Add an item to a collection.	<code>Add</code>
Remove an object from a collection.	<code>Remove</code>
Reference an item in a collection.	<code>Item[String]</code>
Return a reference to an <code>IEnumerable</code> interface.	<code>IEnumerable.GetEnumerator</code>

See also

- [Keywords](#)
- [Visual Basic Runtime Library Members](#)

Control Flow Summary (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Branch.	GoTo , On Error
Exit or pause the program.	End , Exit , Stop
Loop.	Do...Loop , For...Next , For Each...Next , While...End While , With
Make decisions.	Choose , If...Then...Else , Select Case , Switch
Use procedures.	Call , Function , Property , Sub

See also

- [Keywords](#)
- [Visual Basic Runtime Library Members](#)

Conversion Summary (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Convert ANSI value to string.	Chr , ChrW
Convert string to lowercase or uppercase.	Format , LCase , UCase
Convert date to serial number.	DateSerial , DateValue
Convert decimal number to other bases.	Hex , Oct
Convert number to string.	Format , Str
Convert one data type to another.	CBool , CByte , CDate , CDbl , CDec , CInt , CLng , CSng , CShort , CStr , CType , Fix , Int
Convert date to day, month, weekday, or year.	Day , Month , Weekday , Year
Convert time to hour, minute, or second.	Hour , Minute , Second
Convert string to ASCII value.	Asc , AscW
Convert string to number.	Val
Convert time to serial number.	TimeSerial , TimeValue

See also

- [Keywords](#)
- [Visual Basic Runtime Library Members](#)

Data Types Summary (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Convert between data types	CBool, CByte, CChar, CDate, CDbl, CDec, CInt, CLng, CObj, CShort, CSng, CStr, Fix, Int
Set intrinsic data types	Boolean, Byte, Char, Date, Decimal, Double, Integer, Long, Object, Short, Single, String
Verify data types	IsArray, IsDate, IsDBNull, IsError, IsNothing, IsNumeric, IsReference

See also

- [Keywords](#)
- [Visual Basic Runtime Library Members](#)

Dates and Times Summary (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Get the current date or time.	Now , Today , TimeOfDay
Perform date calculations.	DateAdd , DateDiff , DatePart
Return a date.	DateSerial , DateValue , MonthName , WeekdayName
Return a time.	TimeSerial , TimeValue
Set the date or time.	DateString , TimeOfDay , TimeString , Today
Time a process.	Timer

See also

- [Keywords](#)
- [Visual Basic Runtime Library Members](#)

Declarations and Constants Summary (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Assign a value.	Get , Property
Declare variables or constants.	Const , Dim , Private , Protected , Public , Shadows , Shared , Static
Declare a class, delegate, enumeration, module, namespace, or structure.	Class , Delegate , Enum , Module , Namespace , Structure
Create objects.	CreateObject , GetObject , New
Get information about an object.	GetType , IsArray , IsDate , IsDBNull , IsError , IsNothing , IsNumeric , IsReference , SystemTypeName , TypeName , VarType , VbTypeName
Refer to the current object.	Me
Require explicit variable declarations.	Option Explicit , Option Strict
Handle events.	AddHandler , Event , RaiseEvent , RemoveHandler
Implement inheritance.	Inherits , MustInherit , MustOverride , MyBase , MyClass , New , NotInheritable , NotOverridable , Overloads , Overridable , Overrides

See also

- [Keywords](#)
- [Visual Basic Runtime Library Members](#)

Directories and Files Summary (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic language keywords and run-time library members are organized by purpose and use.

Using the `My` feature gives you greater productivity and performance in file I/O operations than using these members. For more information, see [FileSystem](#).

ACTION	LANGUAGE ELEMENT
Change a directory or folder.	ChDir
Change the drive.	ChDrive
Copy a file.	FileCopy
Make a directory or folder.	MkDir
Remove a directory or folder.	RmDir
Rename a file, directory, or folder.	Rename
Return the current path.	CurDir
Return a file's date/time stamp.	FileDateTime
Return file, directory, or label attributes.	GetAttr
Return a file's length.	FileLen
Return a file's name or volume label.	Dir
Set attribute information for a file.	SetAttr

See also

- [Keywords](#)
- [Visual Basic Runtime Library Members](#)
- [Reading from Files](#)
- [Writing to Files](#)
- [Creating, Deleting, and Moving Files and Directories](#)
- [Parsing Text Files with the TextFieldParser Object](#)

Errors Summary (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Generate run-time errors.	Clear , Error , Raise
Get exceptions.	GetException
Provide error information.	Err
Trap errors during run time.	On Error , Resume , Try...Catch...Finally
Provide line number of error.	Erl
Provide system error code.	LastDllError

See also

- [Keywords](#)
- [Visual Basic Runtime Library Members](#)

Financial Summary (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Calculate depreciation.	DDB, SLN, SYD
Calculate future value.	FV
Calculate interest rate.	Rate
Calculate internal rate of return.	IRR, MIRR
Calculate number of periods.	NPer
Calculate payments.	IPmt, Pmt, PPmt
Calculate present value.	NPV, PV

See also

- [Keywords](#)
- [Visual Basic Runtime Library Members](#)

Information and Interaction Summary (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Run other programs.	AppActivate , Shell
Call a method or property.	CallByName
Sound a beep from computer.	Beep
Provide a command-line string.	Command
Manipulate COM objects.	CreateObject , GetObject
Retrieve color information.	QBColor , RGB
Control dialog boxes	InputBox , MsgBox

See also

- [Keywords](#)
- [Visual Basic Runtime Library Members](#)

Input and Output Summary (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Access or create a file.	FileOpen
Close files.	FileClose , Reset
Control output appearance.	Format , Print , SPC , TAB , FileWidth
Copy a file.	FileCopy
Get information about a file.	EOF , FileAttr , FileDateTime , FileLen , FreeFile , GetAttr , Loc , LOF , Seek
Get or provide information from/to the user by means of a control dialog box.	InputBox , MsgBox
Manage files.	Dir , Kill , Lock , Unlock
Read from a file.	FileGet , FileGetObject , Input , InputString , LineInput
Return length of a file.	FileLen
Set or get file attributes.	FileAttr , GetAttr , SetAttr
Set read-write position in a file.	Seek
Write to a file.	FilePut , FilePutObject , Print , Write , WriteLine

See also

- [Keywords](#)
- [Visual Basic Runtime Library Members](#)

Math Summary (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Derive trigonometric functions.	Atan , Cos , Sin , Tan
General calculations.	Exp , Log , Sqrt
Generate random numbers.	Randomize , Rnd
Get absolute value.	Abs
Get the sign of an expression.	Sign
Perform numeric conversions.	Fix , Int

See also

- [Derived Math Functions](#)
- [Keywords](#)
- [Visual Basic Runtime Library Members](#)

Derived Math Functions (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

The following table shows non-intrinsic math functions that can be derived from the intrinsic math functions of the `System.Math` object. You can access the intrinsic math functions by adding `Imports System.Math` to your file or project.

FUNCTION	DERIVED EQUIVALENTS
Secant (Sec(x))	$1 / \text{Cos}(x)$
Cosecant (Csc(x))	$1 / \text{Sin}(x)$
Cotangent (Ctan(x))	$1 / \text{Tan}(x)$
Inverse sine (Asin(x))	$\text{Atan}(x / \text{Sqrt}(-x * x + 1))$
Inverse cosine (Acos(x))	$\text{Atan}(-x / \text{Sqrt}(-x * x + 1)) + 2 * \text{Atan}(1)$
Inverse secant (Asec(x))	$2 * \text{Atan}(1) - \text{Atan}(\text{Sign}(x) / \text{Sqrt}(x * x - 1))$
Inverse cosecant (Acsc(x))	$\text{Atan}(\text{Sign}(x) / \text{Sqrt}(x * x - 1))$
Inverse cotangent (Acot(x))	$2 * \text{Atan}(1) - \text{Atan}(x)$
Hyperbolic sine (Sinh(x))	$(\text{Exp}(x) - \text{Exp}(-x)) / 2$
Hyperbolic cosine (Cosh(x))	$(\text{Exp}(x) + \text{Exp}(-x)) / 2$
Hyperbolic tangent (Tanh(x))	$(\text{Exp}(x) - \text{Exp}(-x)) / (\text{Exp}(x) + \text{Exp}(-x))$
Hyperbolic secant (Sech(x))	$2 / (\text{Exp}(x) + \text{Exp}(-x))$
Hyperbolic cosecant (Csch(x))	$2 / (\text{Exp}(x) - \text{Exp}(-x))$
Hyperbolic cotangent (Coth(x))	$(\text{Exp}(x) + \text{Exp}(-x)) / (\text{Exp}(x) - \text{Exp}(-x))$
Inverse hyperbolic sine (Asinh(x))	$\text{Log}(x + \text{Sqrt}(x * x + 1))$
Inverse hyperbolic cosine (Acosh(x))	$\text{Log}(x + \text{Sqrt}(x * x - 1))$
Inverse hyperbolic tangent (Atanh(x))	$\text{Log}((1 + x) / (1 - x)) / 2$
Inverse hyperbolic secant (AsecH(x))	$\text{Log}((\text{Sqrt}(-x * x + 1) + 1) / x)$
Inverse hyperbolic cosecant (Acsch(x))	$\text{Log}((\text{Sign}(x) * \text{Sqrt}(x * x + 1) + 1) / x)$
Inverse hyperbolic cotangent (Acoth(x))	$\text{Log}((x + 1) / (x - 1)) / 2$

See also

- [Math Functions](#)

My Reference (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

The `My` feature makes programming faster and easier by giving you intuitive access to commonly used methods, properties, and events. This table lists the objects contained in `My`, and the actions that can be performed with each.

ACTION	OBJECT
Accessing application information and services.	<p>The <code>My.Application</code> object consists of the following classes:</p> <p><code>ApplicationBase</code> provides members that are available in all projects.</p> <p><code>WindowsFormsApplicationBase</code> provides members available in Windows Forms applications.</p> <p><code>ConsoleApplicationBase</code> provides members available in console applications.</p>
Accessing the host computer and its resources, services, and data.	<code>My.Computer</code> (Computer)
Accessing the forms in the current project.	<code>My.Forms</code> Object
Accessing the application log.	<code>My.Application.Log</code> (Log)
Accessing the current web request.	<code>My.Request</code> Object
Accessing resource elements.	<code>My.Resources</code> Object
Accessing the current web response.	<code>My.Response</code> Object
Accessing user and application level settings.	<code>My.Settings</code> Object
Accessing the current user's security context.	<code>My.User</code> (User)
Accessing XML Web services referenced by the current project.	<code>My.WebServices</code> Object

See also

- [Overview of the Visual Basic Application Model](#)
- [Development with My](#)

Operators Summary (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Arithmetic	<code>^, -, *, /, \, Mod, +, =</code>
Assignment	<code>=, ^=, *=, /=, \=, +=, -=, &=</code>
Comparison	<code>=, <>, <, >, <=, >=, Like, Is</code>
Concatenation	<code>&, +</code>
Logical/bitwise operations	<code>Not, And, Or, Xor, AndAlso, OrElse</code>
Miscellaneous operations	<code>AddressOf, Await, GetType</code>

See also

- [Keywords](#)
- [Visual Basic Runtime Library Members](#)

Registry Summary (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Studio language keywords and run-time library members are organized by purpose and use.

Using the `My` feature provides you with greater productivity and performance in registry operations than these elements. For more information, see [RegistryProxy](#).

ACTION	LANGUAGE ELEMENT
Delete program settings.	DeleteSetting
Read program settings.	GetSetting , GetAllSettings
Save program settings.	SaveSetting

See also

- [Keywords](#)
- [Visual Basic Runtime Library Members](#)
- [Reading from and Writing to the Registry](#)

String Manipulation Summary (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Compare two strings.	StrComp
Convert strings.	StrConv
Reverse a string.	InStrRev , StrReverse
Convert to lowercase or uppercase.	Format , LCase , UCase
Create a string of repeating characters.	Space , StrDup
Find the length of a string.	Len
Format a string.	Format , FormatCurrency , FormatDateTime , FormatNumber , FormatPercent
Manipulate strings.	InStr , Left , LTrim , Mid , Right , RTrim , Trim
Set string comparison rules.	Option Compare
Work with ASCII and ANSI values.	Asc , AscW , Chr , ChrW
Replace a specified substring.	Replace
Return a filter-based string array.	Filter
Return a specified number of substrings.	Split , Join

See also

- [Keywords](#)
- [Visual Basic Runtime Library Members](#)

Attributes (Visual Basic)

8/8/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic provides several attributes that allow objects to interoperate with unmanaged code and one attribute that enables module members to be accessed without the module name. The following table lists the attributes used by Visual Basic.

ComClassAttribute	Instructs the compiler to add metadata that allows a class to be exposed as a COM object.
HideModuleNameAttribute	Allows the module members to be accessed using only the qualification needed for the module.
VBFixedArrayAttribute	Indicates that an array in a structure or non-local variable should be treated as a fixed-length array.
VBFixedStringAttribute	Indicates that a string should be treated as if it were fixed length.

See also

- [Attributes overview](#)

Constants and Enumerations (Visual Basic)

5/9/2019 • 3 minutes to read • [Edit Online](#)

Visual Basic supplies a number of predefined constants and enumerations for developers. Constants store values that remain constant throughout the execution of an application. Enumerations provide a convenient way to work with sets of related constants, and to associate constant values with names.

Constants

Conditional Compilation Constants

The following table lists the predefined constants available for conditional compilation.

CONSTANT	DESCRIPTION
CONFIG	A string that corresponds to the current setting of the Active Solution Configuration box in the Configuration Manager .
DEBUG	A <code>Boolean</code> value that can be set in the Project Properties dialog box. By default, the Debug configuration for a project defines <code>DEBUG</code> . When <code>DEBUG</code> is defined, <code>Debug</code> class methods generate output to the Output window. When it is not defined, <code>Debug</code> class methods are not compiled and no Debug output is generated.
TARGET	A string representing the output type for the project or the setting of the command-line <code>/target</code> option. The possible values of <code>TARGET</code> are: <ul style="list-style-type: none">- "winexe" for a Windows application.- "exe" for a console application.- "library" for a class library.- "module" for a module.- The <code>/target</code> option may be set in the Visual Studio integrated development environment. For more information, see /target (Visual Basic).
TRACE	A <code>Boolean</code> value that can be set in the Project Properties dialog box. By default, all configurations for a project define <code>TRACE</code> . When <code>TRACE</code> is defined, <code>Trace</code> class methods generate output to the Output window. When it is not defined, <code>Trace</code> class methods are not compiled and no <code>Trace</code> output is generated.
VBC_VER	A number representing the Visual Basic version, in <i>major.minor</i> format.

Print and Display Constants

When you call print and display functions, you can use the following constants in your code in place of the actual values.

CONSTANT	DESCRIPTION
<code>vbCrLf</code>	Carriage return/linefeed character combination.
<code>vbCr</code>	Carriage return character.
<code>vbLf</code>	Linefeed character.
<code>vbNewLine</code>	Newline character.
<code>vbNullChar</code>	Null character.
<code>vbNullString</code>	Not the same as a zero-length string (""); used for calling external procedures.
<code>vbObjectError</code>	Error number. User-defined error numbers should be greater than this value. For example: <code>Err.Raise(Number) = vbObjectError + 1000</code>
<code>vbTab</code>	Tab character.
<code>vbBack</code>	Backspace character.
<code>vbFormFeed</code>	Not used in Microsoft Windows.
<code>vbVerticalTab</code>	Not useful in Microsoft Windows.

Enumerations

The following table lists and describes the enumerations provided by Visual Basic.

ENUMERATION	DESCRIPTION
<code>AppWinStyle</code>	Indicates the window style to use for the invoked program when calling the <code>Shell</code> function.
<code>AudioPlayMode</code>	Indicates how to play sounds when calling audio methods.
<code>BuiltInRole</code>	Indicates the type of role to check when calling the <code>IsInRole</code> method.
<code>CallType</code>	Indicates the type of procedure being invoked when calling the <code>CallByName</code> function.
<code>CompareMethod</code>	Indicates how to compare strings when calling comparison functions.
<code>DateFormat</code>	Indicates how to display dates when calling the <code>FormatDateTime</code> function.
<code>DateInterval</code>	Indicates how to determine and format date intervals when calling date-related functions.

ENUMERATION	DESCRIPTION
DeleteDirectoryOption	Specifies what should be done when a directory that is to be deleted contains files or directories.
DueDate	Indicates when payments are due when calling financial methods.
FieldType	Indicates whether text fields are delimited or fixed-width.
FileAttribute	Indicates the file attributes to use when calling file-access functions.
FirstDayOfWeek	Indicates the first day of the week to use when calling date-related functions.
FirstWeekOfYear	Indicates the first week of the year to use when calling date-related functions.
MsgBoxResult	Indicates which button was pressed on a message box, returned by the MsgBox function.
MsgBoxStyle	Indicates which buttons to display when calling the MsgBox function.
OpenAccess	Indicates how to open a file when calling file-access functions.
OpenMode	Indicates how to open a file when calling file-access functions.
OpenShare	Indicates how to open a file when calling file-access functions.
RecycleOption	Specifies whether a file should be deleted permanently or placed in the Recycle Bin.
SearchOption	Specifies whether to search all or only top-level directories.
TriState	Indicates a Boolean value or whether the default should be used when calling number-formatting functions.
UICancelOption	Specifies what should be done if the user clicks Cancel during an operation.
UIOption	Specifies whether or not to show a progress dialog when copying, deleting, or moving files or directories.
VariantType	Indicates the type of a variant object, returned by the VarType function.
VbStrConv	Indicates which type of conversion to perform when calling the StrConv function.

See also

- [Visual Basic Language Reference](#)

- [Visual Basic](#)
- [Constants Overview](#)
- [Enumerations Overview](#)

Data Type Summary (Visual Basic)

8/22/2019 • 3 minutes to read • [Edit Online](#)

The following table shows the Visual Basic data types, their supporting common language runtime types, their nominal storage allocation, and their value ranges.

VISUAL BASIC TYPE	COMMON LANGUAGE RUNTIME TYPE STRUCTURE	NOMINAL STORAGE ALLOCATION	VALUE RANGE
Boolean	Boolean	Depends on implementing platform	<code>True</code> or <code>False</code>
Byte	Byte	1 byte	0 through 255 (unsigned)
Char (single character)	Char	2 bytes	0 through 65535 (unsigned)
Date	DateTime	8 bytes	0:00:00 (midnight) on January 1, 0001 through 11:59:59 PM on December 31, 9999
Decimal	Decimal	16 bytes	0 through +/- 79,228,162,514,264,337,593,543,950,335 (+/- 7.9...E+28) [†] with no decimal point; 0 through +/- 7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest nonzero number is +/- 0.00000000000000000000000000000001 (+/-1E-28) [†]
Double (double-precision floating-point)	Double	8 bytes	-1.79769313486231570E+308 through -4.94065645841246544E-324 [†] for negative values; 4.94065645841246544E-324 through 1.79769313486231570E+308 [†] for positive values
Integer	Int32	4 bytes	-2,147,483,648 through 2,147,483,647 (signed)

VISUAL BASIC TYPE	COMMON LANGUAGE RUNTIME TYPE STRUCTURE	NOMINAL STORAGE ALLOCATION	VALUE RANGE
Long (long integer)	Int64	8 bytes	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807 (9.2...E+18 [†]) (signed)
Object	Object (class)	4 bytes on 32-bit platform 8 bytes on 64-bit platform	Any type can be stored in a variable of type Object
SByte	SByte	1 byte	-128 through 127 (signed)
Short (short integer)	Int16	2 bytes	-32,768 through 32,767 (signed)
Single (single-precision floating-point)	Single	4 bytes	-3.4028235E+38 through -1.401298E-45 [†] for negative values; 1.401298E-45 through 3.4028235E+38 [†] for positive values
String (variable-length)	String (class)	Depends on implementing platform	0 to approximately 2 billion Unicode characters
UInteger	UInt32	4 bytes	0 through 4,294,967,295 (unsigned)
ULong	UInt64	8 bytes	0 through 18,446,744,073,709,551,615 (1.8...E+19 [†]) (unsigned)
User-Defined (structure)	(inherits from ValueType)	Depends on implementing platform	Each member of the structure has a range determined by its data type and independent of the ranges of the other members
UShort	UInt16	2 bytes	0 through 65,535 (unsigned)

[†] In *scientific notation*, "E" refers to a power of 10. So 3.56E+2 signifies 3.56×10^2 or 356, and 3.56E-2 signifies $3.56 / 10^2$ or 0.0356.

NOTE

For strings containing text, use the [StrConv](#) function to convert from one text format to another.

In addition to specifying a data type in a declaration statement, you can force the data type of some programming elements by using a type character. See [Type Characters](#).

Memory Consumption

When you declare an elementary data type, it is not safe to assume that its memory consumption is the same as its nominal storage allocation. This is due to the following considerations:

- **Storage Assignment.** The common language runtime can assign storage based on the current characteristics of the platform on which your application is executing. If memory is nearly full, it might pack your declared elements as closely together as possible. In other cases it might align their memory addresses to natural hardware boundaries to optimize performance.
- **Platform Width.** Storage assignment on a 64-bit platform is different from assignment on a 32-bit platform.

Composite Data Types

The same considerations apply to each member of a composite data type, such as a structure or an array. You cannot rely on simply adding together the nominal storage allocations of the type's members. Furthermore, there are other considerations, such as the following:

- **Overhead.** Some composite types have additional memory requirements. For example, an array uses extra memory for the array itself and also for each dimension. On a 32-bit platform, this overhead is currently 12 bytes plus 8 bytes for each dimension. On a 64-bit platform this requirement is doubled.
- **Storage Layout.** You cannot safely assume that the order of storage in memory is the same as your order of declaration. You cannot even make assumptions about byte alignment, such as a 2-byte or 4-byte boundary. If you are defining a class or structure and you need to control the storage layout of its members, you can apply the [StructLayoutAttribute](#) attribute to the class or structure.

Object Overhead

An `Object` referring to any elementary or composite data type uses 4 bytes in addition to the data contained in the data type.

See also

- [StrConv](#)
- [StructLayoutAttribute](#)
- [Type Conversion Functions](#)
- [Conversion Summary](#)
- [Type Characters](#)
- [Efficient Use of Data Types](#)

Boolean Data Type (Visual Basic)

6/20/2019 • 2 minutes to read • [Edit Online](#)

Holds values that can be only `True` or `False`. The keywords `True` and `False` correspond to the two states of `Boolean` variables.

Remarks

Use the [Boolean Data Type \(Visual Basic\)](#) to contain two-state values such as true/false, yes/no, or on/off.

The default value of `Boolean` is `False`.

`Boolean` values are not stored as numbers, and the stored values are not intended to be equivalent to numbers. You should never write code that relies on equivalent numeric values for `True` and `False`. Whenever possible, you should restrict usage of `Boolean` variables to the logical values for which they are designed.

Type Conversions

When Visual Basic converts numeric data type values to `Boolean`, 0 becomes `False` and all other values become `True`. When Visual Basic converts `Boolean` values to numeric types, `False` becomes 0 and `True` becomes -1.

When you convert between `Boolean` values and numeric data types, keep in mind that the .NET Framework conversion methods do not always produce the same results as the Visual Basic conversion keywords. This is because the Visual Basic conversion retains behavior compatible with previous versions. For more information, see "Boolean Type Does Not Convert to Numeric Type Accurately" in [Troubleshooting Data Types](#).

Programming Tips

- **Negative Numbers.** `Boolean` is not a numeric type and cannot represent a negative value. In any case, you should not use `Boolean` to hold numeric values.
- **Type Characters.** `Boolean` has no literal type character or identifier type character.
- **Framework Type.** The corresponding type in the .NET Framework is the [System.Boolean](#) structure.

Example

In the following example, `runningVB` is a `Boolean` variable, which stores a simple yes/no setting.

```
Dim runningVB As Boolean
' Check to see if program is running on Visual Basic engine.
If scriptEngine = "VB" Then
    runningVB = True
End If
```

See also

- [System.Boolean](#)
- [Data Types](#)

- Type Conversion Functions
- Conversion Summary
- Efficient Use of Data Types
- Troubleshooting Data Types
- CType Function

Byte data type (Visual Basic)

4/28/2019 • 3 minutes to read • [Edit Online](#)

Holds unsigned 8-bit (1-byte) integers that range in value from 0 through 255.

Remarks

Use the `Byte` data type to contain binary data.

The default value of `Byte` is 0.

Literal assignments

You can declare and initialize a `Byte` variable by assigning it a decimal literal, a hexadecimal literal, an octal literal, or (starting with Visual Basic 2017) a binary literal. If the integral literal is outside the range of a `Byte` (that is, if it is less than `Byte.MinValue` or greater than `Byte.MaxValue`), a compilation error occurs.

In the following example, integers equal to 201 that are represented as decimal, hexadecimal, and binary literals are implicitly converted from `Integer` to `byte` values.

```
Dim byteValue1 As Byte = 201
Console.WriteLine(byteValue1)

Dim byteValue2 As Byte = &H00C9
Console.WriteLine(byteValue2)

Dim byteValue3 As Byte = &B1100_1001
Console.WriteLine(byteValue3)
' The example displays the following output:
'      201
'      201
'      201
```

NOTE

You use the prefix `&h` or `&H` to denote a hexadecimal literal, the prefix `&b` or `&B` to denote a binary literal, and the prefix `&o` or `&O` to denote an octal literal. Decimal literals have no prefix.

Starting with Visual Basic 2017, you can also use the underscore character, `_`, as a digit separator to enhance readability, as the following example shows.

```
Dim byteValue3 As Byte = &B1100_1001
Console.WriteLine(byteValue3)
' The example displays the following output:
'      201
```

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As Byte = &H_6A
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see [setting the Visual Basic language version](#).

Programming tips

- **Negative Numbers.** Because `Byte` is an unsigned type, it cannot represent a negative number. If you use the unary minus (`-`) operator on an expression that evaluates to type `Byte`, Visual Basic converts the expression to `Short` first.
- **Format Conversions.** When Visual Basic reads or writes files, or when it calls DLLs, methods, and properties, it can automatically convert between data formats. Binary data stored in `Byte` variables and arrays is preserved during such format conversions. You should not use a `String` variable for binary data, because its contents can be corrupted during conversion between ANSI and Unicode formats.
- **Widening.** The `Byte` data type widens to `Short`, `UShort`, `Integer`, `UInteger`, `Long`, `ULong`, `Decimal`, `Single`, or `Double`. This means you can convert `Byte` to any of these types without encountering a `System.OverflowException` error.
- **Type Characters.** `Byte` has no literal type character or identifier type character.
- **Framework Type.** The corresponding type in the .NET Framework is the `System.Byte` structure.

Example

In the following example, `b` is a `Byte` variable. The statements demonstrate the range of the variable and the application of bit-shift operators to it.

```
' The valid range of a Byte variable is 0 through 255.
Dim b As Byte
b = 30
' The following statement causes an error because the value is too large.
'b = 256
' The following statement causes an error because the value is negative.
'b = -5
' The following statement sets b to 6.
b = CByte(5.7)

' The following statements apply bit-shift operators to b.
' The initial value of b is 6.
Console.WriteLine(b)
' Bit shift to the right divides the number in half. In this
' example, binary 110 becomes 11.
b >>= 1
' The following statement displays 3.
Console.WriteLine(b)
' Now shift back to the original position, and then one more bit
' to the left. Each shift to the left doubles the value. In this
' example, binary 11 becomes 1100.
b <<= 2
' The following statement displays 12.
Console.WriteLine(b)
```

See also

- [System.Byte](#)
- [Data Types](#)
- [Type Conversion Functions](#)
- [Conversion Summary](#)
- [Efficient Use of Data Types](#)

Char Data Type (Visual Basic)

7/30/2019 • 2 minutes to read • [Edit Online](#)

Holds unsigned 16-bit (2-byte) code points ranging in value from 0 through 65535. Each *code point*, or character code, represents a single Unicode character.

Remarks

Use the `Char` data type when you need to hold only a single character and do not need the overhead of `String`. In some cases you can use `Char()`, an array of `Char` elements, to hold multiple characters.

The default value of `Char` is the character with a code point of 0.

Unicode Characters

The first 128 code points (0–127) of Unicode correspond to the letters and symbols on a standard U.S. keyboard. These first 128 code points are the same as those the ASCII character set defines. The second 128 code points (128–255) represent special characters, such as Latin-based alphabet letters, accents, currency symbols, and fractions. Unicode uses the remaining code points (256–65535) for a wide variety of symbols, including worldwide textual characters, diacritics, and mathematical and technical symbols.

You can use methods like `IsDigit` and `IsPunctuation` on a `Char` variable to determine its Unicode classification.

Type Conversions

Visual Basic does not convert directly between `Char` and the numeric types. You can use the `Asc` or `AscW` function to convert a `Char` value to an `Integer` that represents its code point. You can use the `Chr` or `ChrW` function to convert an `Integer` value to a `Char` that has that code point.

If the type checking switch (the [Option Strict Statement](#)) is on, you must append the literal type character to a single-character string literal to identify it as the `Char` data type. The following example illustrates this. The first assignment to the `charVar` variable generates compiler error `BC30512` because `Option Strict` is on. The second compiles successfully because the `c` literal type character identifies the literal as a `Char` value.

```
Option Strict On

Module CharType
    Public Sub Main()
        Dim charVar As Char

        ' This statement generates compiler error BC30512 because Option Strict is On.
        charVar = "Z"

        ' The following statement succeeds because it specifies a Char literal.
        charVar = "Z"c
    End Sub
End Module
```

Programming Tips

- **Negative Numbers.** `Char` is an unsigned type and cannot represent a negative value. In any case, you should not use `Char` to hold numeric values.

- **Interop Considerations.** If you interface with components not written for the .NET Framework, for example Automation or COM objects, remember that character types have a different data width (8 bits) in other environments. If you pass an 8-bit argument to such a component, declare it as `Byte` instead of `char` in your new Visual Basic code.
- **Widening.** The `Char` data type widens to `String`. This means you can convert `char` to `String` and will not encounter a `System.OverflowException`.
- **Type Characters.** Appending the literal type character `c` to a single-character string literal forces it to the `Char` data type. `char` has no identifier type character.
- **Framework Type.** The corresponding type in the .NET Framework is the `System.Char` structure.

See also

- [System.Char](#)
- [Asc](#)
- [AscW](#)
- [Chr](#)
- [ChrW](#)
- [Data Types](#)
- [String Data Type](#)
- [Type Conversion Functions](#)
- [Conversion Summary](#)
- [How to: Call a Windows Function that Takes Unsigned Types](#)
- [Efficient Use of Data Types](#)

Date Data Type (Visual Basic)

7/30/2019 • 3 minutes to read • [Edit Online](#)

Holds IEEE 64-bit (8-byte) values that represent dates ranging from January 1 of the year 0001 through December 31 of the year 9999, and times from 12:00:00 AM (midnight) through 11:59:59.9999999 PM. Each increment represents 100 nanoseconds of elapsed time since the beginning of January 1 of the year 1 in the Gregorian calendar. The maximum value represents 100 nanoseconds before the beginning of January 1 of the year 10000.

Remarks

Use the `Date` data type to contain date values, time values, or date and time values.

The default value of `Date` is 0:00:00 (midnight) on January 1, 0001.

You can get the current date and time from the [DateAndTime](#) class.

Format Requirements

You must enclose a `Date` literal within number signs (# #). You must specify the date value in the format M/d/yyyy, for example `#5/31/1993#`, or yyyy-MM-dd, for example `#1993-5-31#`. You can use slashes when specifying the year first. This requirement is independent of your locale and your computer's date and time format settings.

The reason for this restriction is that the meaning of your code should never change depending on the locale in which your application is running. Suppose you hard-code a `Date` literal of `#3/4/1998#` and intend it to mean March 4, 1998. In a locale that uses mm/dd/yyyy, 3/4/1998 compiles as you intend. But suppose you deploy your application in many countries/regions. In a locale that uses dd/mm/yyyy, your hard-coded literal would compile to April 3, 1998. In a locale that uses yyyy/mm/dd, the literal would be invalid (April 1998, 0003) and cause a compiler error.

Workarounds

To convert a `Date` literal to the format of your locale, or to a custom format, supply the literal to the [Format](#) function, specifying either a predefined or user-defined date format. The following example demonstrates this.

```
MsgBox("The formatted date is " & Format(#5/31/1993#, "dddd, d MMM yyyy"))
```

Alternatively, you can use one of the overloaded constructors of the [DateTime](#) structure to assemble a date and time value. The following example creates a value to represent May 31, 1993 at 12:14 in the afternoon.

```
Dim dateInMay As New System.DateTime(1993, 5, 31, 12, 14, 0)
```

Hour Format

You can specify the time value in either 12-hour or 24-hour format, for example `#1:15:30 PM#` or `#13:15:30#`. However, if you do not specify either the minutes or the seconds, you must specify AM or PM.

Date and Time Defaults

If you do not include a date in a date/time literal, Visual Basic sets the date part of the value to January 1, 0001. If you do not include a time in a date/time literal, Visual Basic sets the time part of the value to the start of the day, that is, midnight (0:00:00).

Type Conversions

If you convert a `Date` value to the `String` type, Visual Basic renders the date according to the short date format specified by the run-time locale, and it renders the time according to the time format (either 12-hour or 24-hour) specified by the run-time locale.

Programming Tips

- **Interop Considerations.** If you are interfacing with components not written for the .NET Framework, for example Automation or COM objects, keep in mind that date/time types in other environments are not compatible with the Visual Basic `Date` type. If you are passing a date/time argument to such a component, declare it as `Double` instead of `Date` in your new Visual Basic code, and use the conversion methods `DateTime.FromOADate` and `DateTime.ToDateTime`.
- **Type Characters.** `Date` has no literal type character or identifier type character. However, the compiler treats literals enclosed within number signs (# #) as `Date`.
- **Framework Type.** The corresponding type in the .NET Framework is the `System.DateTime` structure.

Example

A variable or constant of the `Date` data type holds both the date and the time. The following example illustrates this.

```
Dim someDateAndTime As Date = #8/13/2002 12:14 PM#
```

See also

- [System.DateTime](#)
- [Data Types](#)
- [Standard Date and Time Format Strings](#)
- [Custom Date and Time Format Strings](#)
- [Type Conversion Functions](#)
- [Conversion Summary](#)
- [Efficient Use of Data Types](#)

Decimal Data Type (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

Holds signed 128-bit (16-byte) values representing 96-bit (12-byte) integer numbers scaled by a variable power of 10. The scaling factor specifies the number of digits to the right of the decimal point; it ranges from 0 through 28. With a scale of 0 (no decimal places), the largest possible value is +/- 79,228,162,514,264,337,593,543,950,335 (+/-7.9228162514264337593543950335E+28). With 28 decimal places, the largest value is +/- 7.9228162514264337593543950335, and the smallest nonzero value is +/- 0.0000000000000000000000000000000001 (+/-1E-28).

Remarks

The `Decimal` data type provides the greatest number of significant digits for a number. It supports up to 29 significant digits and can represent values in excess of 7.9228×10^{28} . It is particularly suitable for calculations, such as financial, that require a large number of digits but cannot tolerate rounding errors.

The default value of `Decimal` is 0.

Programming Tips

- **Precision.** `Decimal` is not a floating-point data type. The `Decimal` structure holds a binary integer value, together with a sign bit and an integer scaling factor that specifies what portion of the value is a decimal fraction. Because of this, `Decimal` numbers have a more precise representation in memory than floating-point types (`Single` and `Double`).
- **Performance.** The `Decimal` data type is the slowest of all the numeric types. You should weigh the importance of precision against performance before choosing a data type.
- **Widening.** The `Decimal` data type widens to `Single` or `Double`. This means you can convert `Decimal` to either of these types without encountering a `System.OverflowException` error.
- **Trailing Zeros.** Visual Basic does not store trailing zeros in a `Decimal` literal. However, a `Decimal` variable preserves any trailing zeros acquired computationally. The following example illustrates this.

```
Dim d1, d2, d3, d4 As Decimal
d1 = 2.375D
d2 = 1.625D
d3 = d1 + d2
d4 = 4.000D
MsgBox("d1 = " & CStr(d1) & ", d2 = " & CStr(d2) &
      ", d3 = " & CStr(d3) & ", d4 = " & CStr(d4))
```

The output of `MsgBox` in the preceding example is as follows:

```
d1 = 2.375, d2 = 1.625, d3 = 4.000, d4 = 4
```

- **Type Characters.** Appending the literal type character `D` to a literal forces it to the `Decimal` data type. Appending the identifier type character `@` to any identifier forces it to `Decimal`.
- **Framework Type.** The corresponding type in the .NET Framework is the `System.Decimal` structure.

Range

You might need to use the `D` type character to assign a large value to a `Decimal` variable or constant. This requirement is because the compiler interprets a literal as `Long` unless a literal type character follows the literal, as the following example shows.

```
Dim bigDec1 As Decimal = 9223372036854775807      ' No overflow.  
Dim bigDec2 As Decimal = 9223372036854775808      ' Overflow.  
Dim bigDec3 As Decimal = 9223372036854775808D    ' No overflow.
```

The declaration for `bigDec1` doesn't produce an overflow because the value that's assigned to it falls within the range for `Long`. The `Long` value can be assigned to the `Decimal` variable.

The declaration for `bigDec2` generates an overflow error because the value that's assigned to it is too large for `Long`. Because the numeric literal can't first be interpreted as a `Long`, it can't be assigned to the `Decimal` variable.

For `bigDec3`, the literal type character `D` solves the problem by forcing the compiler to interpret the literal as a `Decimal` instead of as a `Long`.

See also

- [System.Decimal](#)
- [Decimal.Decimal](#)
- [Math.Round](#)
- [Data Types](#)
- [Single Data Type](#)
- [Double Data Type](#)
- [Type Conversion Functions](#)
- [Conversion Summary](#)
- [Efficient Use of Data Types](#)

Double Data Type (Visual Basic)

7/30/2019 • 2 minutes to read • [Edit Online](#)

Holds signed IEEE 64-bit (8-byte) double-precision floating-point numbers that range in value from -1.79769313486231570E+308 through -4.94065645841246544E-324 for negative values and from 4.94065645841246544E-324 through 1.79769313486231570E+308 for positive values. Double-precision numbers store an approximation of a real number.

Remarks

The `Double` data type provides the largest and smallest possible magnitudes for a number.

The default value of `Double` is 0.

Programming Tips

- **Precision.** When you work with floating-point numbers, remember 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. For more information, see [Troubleshooting Data Types](#).
- **Trailing Zeros.** The floating-point data types do not have any internal representation of trailing zero characters. For example, they do not distinguish between 4.2000 and 4.2. Consequently, trailing zero characters do not appear when you display or print floating-point values.
- **Type Characters.** Appending the literal type character `R` to a literal forces it to the `Double` data type. For example, if an integer value is followed by `R`, the value is changed to a `Double`.

```
' Visual Basic expands the 4 in the statement Dim dub As Double = 4.0:  
Dim dub As Double = 4.0R
```

Appending the identifier type character `#` to any identifier forces it to `Double`. In the following example, the variable `num#` is typed as a `Double`:

```
Dim num# = 3
```

- **Framework Type.** The corresponding type in the .NET Framework is the [System.Double](#) structure.

See also

- [System.Double](#)
- [Data Types](#)
- [Decimal Data Type](#)
- [Single Data Type](#)
- [Type Conversion Functions](#)
- [Conversion Summary](#)
- [Efficient Use of Data Types](#)
- [Troubleshooting Data Types](#)
- [Type Characters](#)

Integer data type (Visual Basic)

4/28/2019 • 3 minutes to read • [Edit Online](#)

Holds signed 32-bit (4-byte) integers that range in value from -2,147,483,648 through 2,147,483,647.

Remarks

The `Integer` data type provides optimal performance on a 32-bit processor. The other integral types are slower to load and store from and to memory.

The default value of `Integer` is 0.

Literal assignments

You can declare and initialize an `Integer` variable by assigning it a decimal literal, a hexadecimal literal, an octal literal, or (starting with Visual Basic 2017) a binary literal. If the integer literal is outside the range of `Integer` (that is, if it is less than `Int32.MinValue` or greater than `Int32.MaxValue`), a compilation error occurs.

In the following example, integers equal to 90,946 that are represented as decimal, hexadecimal, and binary literals are assigned to `Integer` values.

```
Dim intValue1 As Integer = 90946
Console.WriteLine(intValue1)
Dim intValue2 As Integer = &H16342
Console.WriteLine(intValue2)

Dim intValue3 As Integer = &B0001_0110_0011_0100_0010
Console.WriteLine(intValue3)
' The example displays the following output:
'      90946
'      90946
'      90946
```

NOTE

You use the prefix `&h` or `&H` to denote a hexadecimal literal, the prefix `&b` or `&B` to denote a binary literal, and the prefix `&o` or `&O` to denote an octal literal. Decimal literals have no prefix.

Starting with Visual Basic 2017, you can also use the underscore character, `_`, as a digit separator to enhance readability, as the following example shows.

```

Dim intValue1 As Integer = 90_946
Console.WriteLine(intValue1)

Dim intValue2 As Integer = &H0001_6342
Console.WriteLine(intValue2)

Dim intValue3 As Integer = &B0001_0110_0011_0100_0010
Console.WriteLine(intValue3)
' The example displays the following output:
'      90946
'      90946
'      90946

```

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As Integer = &H_C305_F860
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```

<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>

```

For more information see [setting the Visual Basic language version](#).

Numeric literals can also include the `I` type character to denote the `Integer` data type, as the following example shows.

```
Dim number = &H_035826I
```

Programming tips

- **Interop Considerations.** If you are interfacing with components not written for the .NET Framework, such as Automation or COM objects, remember that `Integer` has a different data width (16 bits) in other environments. If you are passing a 16-bit argument to such a component, declare it as `Short` instead of `Integer` in your new Visual Basic code.
- **Widening.** The `Integer` data type widens to `Long`, `Decimal`, `Single`, or `Double`. This means you can convert `Integer` to any one of these types without encountering a [System.OverflowException](#) error.
- **Type Characters.** Appending the literal type character `I` to a literal forces it to the `Integer` data type. Appending the identifier type character `%` to any identifier forces it to `Integer`.
- **Framework Type.** The corresponding type in the .NET Framework is the [System.Int32](#) structure.

Range

If you try to set a variable of an integral type to a number outside the range for that type, an error occurs. If you try to set it to a fraction, the number is rounded up or down to the nearest integer value. If the number is equally close to two integer values, the value is rounded to the nearest even integer. This behavior minimizes rounding errors that result from consistently rounding a midpoint value in a single direction. The following code shows examples of rounding.

```
' The valid range of an Integer variable is -2147483648 through +2147483647.  
Dim k As Integer  
' The following statement causes an error because the value is too large.  
k = 2147483648  
' The following statement sets k to 6.  
k = 5.9  
' The following statement sets k to 4  
k = 4.5  
' The following statement sets k to 6  
' Note, Visual Basic uses banker's rounding (toward nearest even number)  
k = 5.5
```

See also

- [System.Int32](#)
- [Data Types](#)
- [Long Data Type](#)
- [Short Data Type](#)
- [Type Conversion Functions](#)
- [Conversion Summary](#)
- [Efficient Use of Data Types](#)

Long data type (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Holds signed 64-bit (8-byte) integers ranging in value from -9,223,372,036,854,775,808 through 9,223,372,036,854,775,807 (9.2...E+18).

Remarks

Use the `Long` data type to contain integer numbers that are too large to fit in the `Integer` data type.

The default value of `Long` is 0.

Literal assignments

You can declare and initialize a `Long` variable by assigning it a decimal literal, a hexadecimal literal, an octal literal, or (starting with Visual Basic 2017) a binary literal. If the integer literal is outside the range of `Long` (that is, if it is less than `Int64.MinValue` or greater than `Int64.MaxValue`, a compilation error occurs.

In the following example, integers equal to 4,294,967,296 that are represented as decimal, hexadecimal, and binary literals are assigned to `Long` values.

```
Dim longValue1 As Long = 4294967296
Console.WriteLine(longValue1)

Dim longValue2 As Long = &H100000000
Console.WriteLine(longValue2)

Dim longValue3 As Long = &B1_0000_0000_0000_0000_0000_0000_0000
Console.WriteLine(longValue3)
' The example displays the following output:
'      4294967296
'      4294967296
'      4294967296
```

NOTE

You use the prefix `&h` or `&H` to denote a hexadecimal literal, the prefix `&b` or `&B` to denote a binary literal, and the prefix `&o` or `&O` to denote an octal literal. Decimal literals have no prefix.

Starting with Visual Basic 2017, you can also use the underscore character, `_`, as a digit separator to enhance readability, as the following example shows.

```

Dim longValue1 As Long = 4_294_967_296
Console.WriteLine(longValue1)

Dim longValue2 As Long = &H1_0000_0000
Console.WriteLine(longValue2)

Dim longValue3 As Long = &B1_0000_0000_0000_0000_0000_0000_0000
Console.WriteLine(longValue3)
' The example displays the following output:
'      4294967296
'      4294967296
'      4294967296

```

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As Long = &H_0FAC_0326_1489_D68C
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see [setting the Visual Basic language version](#).

Numeric literals can also include the `L` type character to denote the `Long` data type, as the following example shows.

```
Dim number = &H_0FAC_0326_1489_D68CL
```

Programming tips

- **Interop Considerations.** If you are interfacing with components not written for the .NET Framework, for example Automation or COM objects, remember that `Long` has a different data width (32 bits) in other environments. If you are passing a 32-bit argument to such a component, declare it as `Integer` instead of `Long` in your new Visual Basic code.
- **Widening.** The `Long` data type widens to `Decimal`, `Single`, or `Double`. This means you can convert `Long` to any one of these types without encountering a `System.OverflowException` error.
- **Type Characters.** Appending the literal type character `L` to a literal forces it to the `Long` data type. Appending the identifier type character `&` to any identifier forces it to `Long`.
- **Framework Type.** The corresponding type in the .NET Framework is the `System.Int64` structure.

See also

- [Int64](#)
- [Data Types](#)
- [Integer Data Type](#)
- [Short Data Type](#)
- [Type Conversion Functions](#)

- Conversion Summary
- Efficient Use of Data Types

Object Data Type

7/26/2019 • 2 minutes to read • [Edit Online](#)

Holds addresses that refer to objects. You can assign any reference type (string, array, class, or interface) to an `Object` variable. An `Object` variable can also refer to data of any value type (numeric, `Boolean`, `Char`, `Date`, structure, or enumeration).

Remarks

The `Object` data type can point to data of any data type, including any object instance your application recognizes. Use `Object` when you do not know at compile time what data type the variable might point to.

The default value of `Object` is `Nothing` (a null reference).

Data Types

You can assign a variable, constant, or expression of any data type to an `Object` variable. To determine the data type an `Object` variable currently refers to, you can use the `GetTypeCode` method of the `System.Type` class. The following example illustrates this.

```
Dim myObject As Object
' Suppose myObject has now had something assigned to it.
Dim datTyp As Integer
datTyp = Type.GetTypeCode(myObject.GetType())
```

The `Object` data type is a reference type. However, Visual Basic treats an `Object` variable as a value type when it refers to data of a value type.

Storage

Whatever data type it refers to, an `Object` variable does not contain the data value itself, but rather a pointer to the value. It always uses four bytes in computer memory, but this does not include the storage for the data representing the value of the variable. Because of the code that uses the pointer to locate the data, `Object` variables holding value types are slightly slower to access than explicitly typed variables.

Programming Tips

- **Interop Considerations.** If you are interfacing with components not written for the .NET Framework, for example Automation or COM objects, keep in mind that pointer types in other environments are not compatible with the Visual Basic `Object` type.
- **Performance.** A variable you declare with the `Object` type is flexible enough to contain a reference to any object. However, when you invoke a method or property on such a variable, you always incur *late binding* (at run time). To force *early binding* (at compile time) and better performance, declare the variable with a specific class name, or cast it to the specific data type.

When you declare an object variable, try to use a specific class type, for example `OperatingSystem`, instead of the generalized `Object` type. You should also use the most specific class available, such as `TextBox` instead of `Control`, so that you can access its properties and methods. You can usually use the **Classes** list in the **Object Browser** to find available class names.

- **Widening.** All data types and all reference types widen to the `Object` data type. This means you can convert any type to `Object` without encountering a `System.OverflowException` error.

However, if you convert between value types and `Object`, Visual Basic performs operations called *boxing* and *unboxing*, which make execution slower.

- **Type Characters.** `Object` has no literal type character or identifier type character.
- **Framework Type.** The corresponding type in the .NET Framework is the `System.Object` class.

Example

The following example illustrates an `Object` variable pointing to an object instance.

```
Dim objDb As Object
Dim myCollection As New Collection()
' Suppose myCollection has now been populated.
objDb = myCollection.Item(1)
```

See also

- [Object](#)
- [Data Types](#)
- [Type Conversion Functions](#)
- [Conversion Summary](#)
- [Efficient Use of Data Types](#)
- [How to: Determine Whether Two Objects Are Related](#)
- [How to: Determine Whether Two Objects Are Identical](#)

SByte data type (Visual Basic)

10/18/2019 • 3 minutes to read • [Edit Online](#)

Holds signed 8-bit (1-byte) integers that range in value from -128 through 127.

Remarks

Use the `SByte` data type to contain integer values that do not require the full data width of `Integer` or even the half data width of `Short`. In some cases, the common language runtime might be able to pack your `SByte` variables closely together and save memory consumption.

The default value of `SByte` is 0.

Literal assignments

You can declare and initialize an `sbyte` variable by assigning it a decimal literal, a hexadecimal literal, an octal literal, or (starting with Visual Basic 2017) a binary literal.

In the following example, integers equal to -102 that are represented as decimal, hexadecimal, and binary literals are assigned to `SByte` values. This example requires that you compile with the `/removeintchecks` compiler switch.

```
Dim sbyteValue1 As SByte = -102
Console.WriteLine(sbyteValue1)

Dim sbyteValue4 As SByte = &H9A
Console.WriteLine(sbyteValue4)

Dim sbyteValue5 As SByte = &B1001_1010
Console.WriteLine(sbyteValue5)
' The example displays the following output:
'      -102
'      -102
'      -102
```

NOTE

You use the prefix `&h` or `&H` to denote a hexadecimal literal, the prefix `&b` or `&B` to denote a binary literal, and the prefix `&o` or `&O` to denote an octal literal. Decimal literals have no prefix.

Starting with Visual Basic 2017, you can also use the underscore character, `_`, as a digit separator to enhance readability, as the following example shows.

```
Dim sbyteValue3 As SByte = &B1001_1010
Console.WriteLine(sbyteValue3)
' The example displays the following output:
'      -102
```

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As SByte = &H_F9
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see [setting the Visual Basic language version](#).

If the integer literal is outside the range of `sbyte` (that is, if it is less than `SByte.MinValue` or greater than `SByte.MaxValue`), a compilation error occurs. When an integer literal has no suffix, an `Integer` is inferred. If the integer literal is outside the range of the `Integer` type, a `Long` is inferred. This means that, in the previous examples, the numeric literals `0x9A` and `0b10011010` are interpreted as 32-bit signed integers with a value of 156, which exceeds `SByte.MaxValue`. To successfully compile code like this that assigns a non-decimal integer to an `sbyte`, you can do either of the following:

- Disable integer bounds checks by compiling with the `/removeintchecks` compiler switch.
- Use a [type character](#) to explicitly define the literal value that you want to assign to the `sbyte`. The following example assigns a negative literal `Short` value to an `sbyte`. Note that, for negative numbers, the high-order bit of the high-order word of the numeric literal must be set. In the case of our example, this is bit 15 of the literal `Short` value.

```
Dim sByteValue1 As SByte = &HFF_9As
Dim sByteValue2 As SByte = &B1111_1111_1001_1010s
Console.WriteLine(sByteValue1)
Console.WriteLine(sByteValue2)
```

Programming tips

- **CLS Compliance.** The `sbyte` data type is not part of the [Common Language Specification \(CLS\)](#), so CLS-compliant code cannot consume a component that uses it.
- **Widening.** The `sbyte` data type widens to `Short`, `Integer`, `Long`, `Decimal`, `Single`, and `Double`. This means you can convert `sbyte` to any of these types without encountering a `System.OverflowException` error.
- **Type Characters.** `sbyte` has no literal type character or identifier type character.
- **Framework Type.** The corresponding type in the .NET Framework is the `System.SByte` structure.

See also

- [System.SByte](#)
- [Data Types](#)
- [Type Conversion Functions](#)
- [Conversion Summary](#)
- [Short Data Type](#)
- [Integer Data Type](#)
- [Long Data Type](#)
- [Efficient Use of Data Types](#)

Short data type (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

Holds signed 16-bit (2-byte) integers that range in value from -32,768 through 32,767.

Remarks

Use the `Short` data type to contain integer values that do not require the full data width of `Integer`. In some cases, the common language runtime can pack your `short` variables closely together and save memory consumption.

The default value of `Short` is 0.

Literal assignments

You can declare and initialize a `Short` variable by assigning it a decimal literal, a hexadecimal literal, an octal literal, or (starting with Visual Basic 2017) a binary literal. If the integer literal is outside the range of `Short` (that is, if it is less than `Int16.MinValue` or greater than `Int16.MaxValue`), a compilation error occurs.

In the following example, integers equal to 1,034 that are represented as decimal, hexadecimal, and binary literals are implicitly converted from `Integer` to `Short` values.

```
Dim shortValue1 As Short = 1034
Console.WriteLine(shortValue1)

Dim shortValue2 As Short = &H040A
Console.WriteLine(shortValue2)

Dim shortValue3 As Short = &B0100_00001010
Console.WriteLine(shortValue3)
' The example displays the following output:
'      1034
'      1034
'      1034
```

NOTE

You use the prefix `&h` or `&H` to denote a hexadecimal literal, the prefix `&b` or `&B` to denote a binary literal, and the prefix `&o` or `&O` to denote an octal literal. Decimal literals have no prefix.

Starting with Visual Basic 2017, you can also use the underscore character, `_`, as a digit separator to enhance readability, as the following example shows.

```
Dim shortValue1 As Short = 1_034
Console.WriteLine(shortValue1)

Dim shortValue3 As Short = &B00000100_00001010
Console.WriteLine(shortValue3)
' The example displays the following output:
'      1034
'      1034
```

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As Short = &H_3264
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see [setting the Visual Basic language version](#).

Numeric literals can also include the `s` type character to denote the `Short` data type, as the following example shows.

```
Dim number = &H_3264S
```

Programming tips

- **Widening.** The `Short` data type widens to `Integer`, `Long`, `Decimal`, `Single`, or `Double`. This means you can convert `Short` to any one of these types without encountering a [System.OverflowException](#) error.
- **Type Characters.** Appending the literal type character `s` to a literal forces it to the `Short` data type. `Short` has no identifier type character.
- **Framework Type.** The corresponding type in the .NET Framework is the [System.Int16](#) structure.

See also

- [System.Int16](#)
- [Data Types](#)
- [Type Conversion Functions](#)
- [Conversion Summary](#)
- [Integer Data Type](#)
- [Long Data Type](#)
- [Efficient Use of Data Types](#)

Single Data Type (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

Holds signed IEEE 32-bit (4-byte) single-precision floating-point numbers ranging in value from -3.4028235E+38 through -1.401298E-45 for negative values and from 1.401298E-45 through 3.4028235E+38 for positive values. Single-precision numbers store an approximation of a real number.

Remarks

Use the `Single` data type to contain floating-point values that do not require the full data width of `Double`. In some cases the common language runtime might be able to pack your `Single` variables closely together and save memory consumption.

The default value of `Single` is 0.

Programming Tips

- **Precision.** 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. For more information, see [Troubleshooting Data Types](#).
- **Widening.** The `Single` data type widens to `Double`. This means you can convert `Single` to `Double` without encountering a [System.OverflowException](#) error.
- **Trailing Zeros.** The floating-point data types do not have any internal representation of trailing 0 characters. For example, they do not distinguish between 4.2000 and 4.2. Consequently, trailing 0 characters do not appear when you display or print floating-point values.
- **Type Characters.** Appending the literal type character `F` to a literal forces it to the `Single` data type. Appending the identifier type character `!` to any identifier forces it to `Single`.
- **Framework Type.** The corresponding type in the .NET Framework is the [System.Single](#) structure.

See also

- [System.Single](#)
- [Data Types](#)
- [Decimal Data Type](#)
- [Double Data Type](#)
- [Type Conversion Functions](#)
- [Conversion Summary](#)
- [Efficient Use of Data Types](#)
- [Troubleshooting Data Types](#)

String Data Type (Visual Basic)

10/1/2019 • 3 minutes to read • [Edit Online](#)

Holds sequences of unsigned 16-bit (2-byte) code points that range in value from 0 through 65535. Each *code point*, or character code, represents a single Unicode character. A string can contain from 0 to approximately two billion (2^{31}) Unicode characters.

Remarks

Use the `String` data type to hold multiple characters without the array management overhead of `Char()`, an array of `Char` elements.

The default value of `String` is `Nothing` (a null reference). Note that this is not the same as the empty string (value `""`).

Unicode Characters

The first 128 code points (0–127) of Unicode correspond to the letters and symbols on a standard U.S. keyboard. These first 128 code points are the same as those the ASCII character set defines. The second 128 code points (128–255) represent special characters, such as Latin-based alphabet letters, accents, currency symbols, and fractions. Unicode uses the remaining code points (256–65535) for a wide variety of symbols. This includes worldwide textual characters, diacritics, and mathematical and technical symbols.

You can use methods such as `IsDigit` and `IsPunctuation` on an individual character in a `String` variable to determine its Unicode classification.

Format Requirements

You must enclose a `String` literal within quotation marks (`" "`). If you must include a quotation mark as one of the characters in the string, you use two contiguous quotation marks (`""`). The following example illustrates this.

```
Dim j As String = "Joe said ""Hello"" to me."
Dim h As String = "Hello"
' The following messages all display the same thing:
' "Joe said "Hello" to me."
MsgBox(j)
MsgBox("Joe said " & " " & h & " " & " to me.")
MsgBox("Joe said "" & h & "" to me.")
```

Note that the contiguous quotation marks that represent a quotation mark in the string are independent of the quotation marks that begin and end the `String` literal.

String Manipulations

Once you assign a string to a `String` variable, that string is *immutable*, which means you cannot change its length or contents. When you alter a string in any way, Visual Basic creates a new string and abandons the previous one. The `String` variable then points to the new string.

You can manipulate the contents of a `String` variable by using a variety of string functions. The following example illustrates the `Left` function

```
Dim S As String = "Database"
' The following statement sets S to a new string containing "Data".
S = Microsoft.VisualBasic.Left(S, 4)
```

A string created by another component might be padded with leading or trailing spaces. If you receive such a string, you can use the [Trim](#), [LTrim](#), and [RTrim](#) functions to remove these spaces.

For more information about string manipulations, see [Strings](#).

Programming Tips

- **Negative Numbers.** Remember that the characters held by `String` are unsigned and cannot represent negative values. In any case, you should not use `String` to hold numeric values.
- **Interop Considerations.** If you are interfacing with components not written for the .NET Framework, for example Automation or COM objects, remember that string characters have a different data width (8 bits) in other environments. If you are passing a string argument of 8-bit characters to such a component, declare it as `Byte()`, an array of `Byte` elements, instead of `string` in your new Visual Basic code.
- **Type Characters.** Appending the identifier type character `$` to any identifier forces it to the `String` data type. `String` has no literal type character. However, the compiler treats literals enclosed in quotation marks (" ") as `String`.
- **Framework Type.** The corresponding type in the .NET Framework is the [System.String](#) class.

See also

- [System.String](#)
- [Data Types](#)
- [Char Data Type](#)
- [Type Conversion Functions](#)
- [Conversion Summary](#)
- [How to: Call a Windows Function that Takes Unsigned Types](#)
- [Efficient Use of Data Types](#)

UInteger data type

10/18/2019 • 2 minutes to read • [Edit Online](#)

Holds unsigned 32-bit (4-byte) integers ranging in value from 0 through 4,294,967,295.

Remarks

The `UInteger` data type provides the largest unsigned value in the most efficient data width.

The default value of `UInteger` is 0.

Literal assignments

You can declare and initialize a `UInteger` variable by assigning it a decimal literal, a hexadecimal literal, an octal literal, or (starting with Visual Basic 2017) a binary literal. If the integer literal is outside the range of `UInteger` (that is, if it is less than `UInt32.MinValue` or greater than `UInt32.MaxValue`), a compilation error occurs.

In the following example, integers equal to 3,000,000,000 that are represented as decimal, hexadecimal, and binary literals are assigned to `UInteger` values.

```
Dim uintValue1 As UInteger = 3000000000ui
Console.WriteLine(uintValue1)

Dim uintValue2 As UInteger = &HB2D05E00ui
Console.WriteLine(uintValue2)

Dim uintValue3 As UInteger = &B1011_0010_1101_0000_0101_1110_0000_0000ui
Console.WriteLine(uintValue3)
' The example displays the following output:
'      3000000000
'      3000000000
'      3000000000
```

NOTE

You use the prefix `&h` or `&H` to denote a hexadecimal literal, the prefix `&b` or `&B` to denote a binary literal, and the prefix `&o` or `&O` to denote an octal literal. Decimal literals have no prefix.

Starting with Visual Basic 2017, you can also use the underscore character, `_`, as a digit separator to enhance readability, as the following example shows.

```
Dim uintValue1 As UInteger = 3_000_000_000ui
Console.WriteLine(uintValue1)

Dim uintValue2 As UInteger = &HB2D0_5E00ui
Console.WriteLine(uintValue2)

Dim uintValue3 As UInteger = &B1011_0010_1101_0000_0101_1110_0000_0000ui
Console.WriteLine(uintValue3)
' The example displays the following output:
'      3000000000
'      3000000000
'      3000000000
```

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As UInteger = &H_0F8C_0326
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see [setting the Visual Basic language version](#).

Numeric literals can also include the `UI` or `ui` type character to denote the `UInteger` data type, as the following example shows.

```
Dim number = &H_0FAC_14D7ui
```

Programming tips

The `UInteger` and `Integer` data types provide optimal performance on a 32-bit processor, because the smaller integer types (`UShort`, `Short`, `Byte`, and `SByte`), even though they use fewer bits, take more time to load, store, and fetch.

- **Negative Numbers.** Because `UInteger` is an unsigned type, it cannot represent a negative number. If you use the unary minus (`-`) operator on an expression that evaluates to type `UInteger`, Visual Basic converts the expression to `Long` first.
- **CLS Compliance.** The `UInteger` data type is not part of the [Common Language Specification](#) (CLS), so CLS-compliant code cannot consume a component that uses it.
- **Interop Considerations.** If you are interfacing with components not written for the .NET Framework, for example Automation or COM objects, keep in mind that types such as `uint` can have a different data width (16 bits) in other environments. If you are passing a 16-bit argument to such a component, declare it as `UShort` instead of `UInteger` in your managed Visual Basic code.
- **Widening.** The `UInteger` data type widens to `Long`, `ULong`, `Decimal`, `Single`, and `Double`. This means you can convert `UInteger` to any of these types without encountering a `System.OverflowException` error.
- **Type Characters.** Appending the literal type characters `UI` to a literal forces it to the `UInteger` data type. `UInteger` has no identifier type character.
- **Framework Type.** The corresponding type in the .NET Framework is the `System.UInt32` structure.

See also

- [UInt32](#)
- [Data Types](#)
- [Type Conversion Functions](#)
- [Conversion Summary](#)
- [How to: Call a Windows Function that Takes Unsigned Types](#)
- [Efficient Use of Data Types](#)

ULong data type (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Holds unsigned 64-bit (8-byte) integers ranging in value from 0 through 18,446,744,073,709,551,615 (more than 1.84 times 10 ¹⁹).

Remarks

Use the `ULong` data type to contain binary data too large for `UInteger`, or the largest possible unsigned integer values.

The default value of `ULong` is 0.

Literal assignments

You can declare and initialize a `ULong` variable by assigning it a decimal literal, a hexadecimal literal, an octal literal, or (starting with Visual Basic 2017) a binary literal. If the integer literal is outside the range of `ULong` (that is, if it is less than `UInt64.MinValue` or greater than `UInt64.MaxValue`), a compilation error occurs.

In the following example, integers equal to 7,934,076,125 that are represented as decimal, hexadecimal, and binary literals are assigned to `ULong` values.

```
Dim ulongValue1 As ULong = 7934076125
Console.WriteLine(ulongValue1)

Dim ulongValue2 As ULong = &H0001D8e864DD
Console.WriteLine(ulongValue2)

Dim ulongValue3 As ULong = &B0001_1101_1000_1110_1000_0110_0100_1101_1101
Console.WriteLine(ulongValue3)
' The example displays the following output:
'      7934076125
'      7934076125
'      7934076125
```

NOTE

You use the prefix `&h` or `&H` to denote a hexadecimal literal, the prefix `&b` or `&B` to denote a binary literal, and the prefix `&o` or `&O` to denote an octal literal. Decimal literals have no prefix.

Starting with Visual Basic 2017, you can also use the underscore character, `_`, as a digit separator to enhance readability, as the following example shows.

```

Dim longValue1 As Long = 4_294_967_296
Console.WriteLine(longValue1)

Dim longValue2 As Long = &H1_0000_0000
Console.WriteLine(longValue2)

Dim longValue3 As Long = &B1_0000_0000_0000_0000_0000_0000_0000
Console.WriteLine(longValue3)
' The example displays the following output:
'      4294967296
'      4294967296
'      4294967296

```

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As ULong = &H_F9AC_0326_1489_D68C
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see [setting the Visual Basic language version](#).

Numeric literals can also include the `UL` or `ul` type character to denote the `ULong` data type, as the following example shows.

```
Dim number = &H_00_00_0A_96_2F_AC_14_D7ul
```

Programming tips

- Negative Numbers.** Because `ULong` is an unsigned type, it cannot represent a negative number. If you use the unary minus (`-`) operator on an expression that evaluates to type `ULong`, Visual Basic converts the expression to `Decimal` first.
- CLS Compliance.** The `ULong` data type is not part of the [Common Language Specification \(CLS\)](#), so CLS-compliant code cannot consume a component that uses it.
- Interop Considerations.** If you are interfacing with components not written for the .NET Framework, for example Automation or COM objects, keep in mind that types such as `ulong` can have a different data width (32 bits) in other environments. If you are passing a 32-bit argument to such a component, declare it as `UInteger` instead of `ULong` in your managed Visual Basic code.

Furthermore, Automation does not support 64-bit integers on Windows 95, Windows 98, Windows ME, or Windows 2000. You cannot pass a Visual Basic `ULong` argument to an Automation component on these platforms.

- Widening.** The `ULong` data type widens to `Decimal`, `Single`, and `Double`. This means you can convert `ULong` to any of these types without encountering a [System.OverflowException](#) error.
- Type Characters.** Appending the literal type characters `UL` to a literal forces it to the `ULong` data type. `ULong` has no identifier type character.

- **Framework Type.** The corresponding type in the .NET Framework is the [System.UInt64](#) structure.

See also

- [UInt64](#)
- [Data Types](#)
- [Type Conversion Functions](#)
- [Conversion Summary](#)
- [How to: Call a Windows Function that Takes Unsigned Types](#)
- [Efficient Use of Data Types](#)

User-Defined Data Type

10/1/2019 • 2 minutes to read • [Edit Online](#)

Holds data in a format you define. The `Structure` statement defines the format.

Previous versions of Visual Basic support the user-defined type (UDT). The current version expands the UDT to a *structure*. A structure is a concatenation of one or more *members* of various data types. Visual Basic treats a structure as a single unit, although you can also access its members individually.

Remarks

Define and use a structure data type when you need to combine various data types into a single unit, or when none of the elementary data types serve your needs.

The default value of a structure data type consists of the combination of the default values of each of its members.

Declaration Format

A structure declaration starts with the [Structure Statement](#) and ends with the [End Structure](#) statement. The `Structure` statement supplies the name of the structure, which is also the identifier of the data type the structure is defining. Other parts of the code can use this identifier to declare variables, parameters, and function return values to be of this structure's data type.

The declarations between the `Structure` and `End Structure` statements define the members of the structure.

Member Access Levels

You must declare every member using a [Dim Statement](#) or a statement that specifies access level, such as [Public](#), [Friend](#), or [Private](#). If you use a `Dim` statement, the access level defaults to public.

Programming Tips

- **Memory Consumption.** As with all composite data types, you cannot safely calculate the total memory consumption of a structure by adding together the nominal storage allocations of its members. Furthermore, you cannot safely assume that the order of storage in memory is the same as your order of declaration. If you need to control the storage layout of a structure, you can apply the [StructLayoutAttribute](#) attribute to the `Structure` statement.
- **Interop Considerations.** If you are interfacing with components not written for the .NET Framework, for example Automation or COM objects, keep in mind that user-defined types in other environments are not compatible with Visual Basic structure types.
- **Widening.** There is no automatic conversion to or from any structure data type. You can define conversion operators on your structure using the [Operator Statement](#), and you can declare each conversion operator to be `Widening` or `Narrowing`.
- **Type Characters.** Structure data types have no literal type character or identifier type character.
- **Framework Type.** There is no corresponding type in the .NET Framework. All structures inherit from the .NET Framework class [System.ValueType](#), but no individual structure corresponds to [System.ValueType](#).

Example

The following paradigm shows the outline of the declaration of a structure.

```
[Public | Protected | Friend | Protected Friend | Private] Structure structname
    {Dim | Public | Friend | Private} member1 As datatype1
    ' ...
    {Dim | Public | Friend | Private} memberN As datatypeN
End Structure
```

See also

- [ValueType](#)
- [StructLayoutAttribute](#)
- [Data Types](#)
- [Type Conversion Functions](#)
- [Conversion Summary](#)
- [Structure Statement](#)
- [Widening](#)
- [Narrowing](#)
- [Structures](#)
- [Efficient Use of Data Types](#)

UShort data type (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

Holds unsigned 16-bit (2-byte) integers ranging in value from 0 through 65,535.

Remarks

Use the `ushort` data type to contain binary data too large for `Byte`.

The default value of `UShort` is 0.

Literal assignments

You can declare and initialize a `ushort` variable by assigning it a decimal literal, a hexadecimal literal, an octal literal, or (starting with Visual Basic 2017) a binary literal. If the integer literal is outside the range of `UShort` (that is, if it is less than `UInt16.MinValue` or greater than `UInt16.MaxValue`), a compilation error occurs.

In the following example, integers equal to 65,034 that are represented as decimal, hexadecimal, and binary literals are assigned to `ushort` values.

```
Dim ushortValue1 As UShort = 65034
Console.WriteLine(ushortValue1)

Dim ushortValue2 As UShort = &HFE0A
Console.WriteLine(ushortValue2)

Dim ushortValue3 As UShort = &B1111_1110_0000_1010
Console.WriteLine(ushortValue3)
' The example displays the following output:
'       65034
'       65034
'       65034
```

NOTE

You use the prefix `&h` or `&H` to denote a hexadecimal literal, the prefix `&b` or `&B` to denote a binary literal, and the prefix `&o` or `&O` to denote an octal literal. Decimal literals have no prefix.

Starting with Visual Basic 2017, you can also use the underscore character, `_`, as a digit separator to enhance readability, as the following example shows.

```
Dim ushortValue1 As UShort = 65_034
Console.WriteLine(ushortValue1)

Dim ushortValue3 As UShort = &B11111110_00001010
Console.WriteLine(ushortValue3)
' The example displays the following output:
'       65034
'       65034
```

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As UShort = &H_FF8C
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see [setting the Visual Basic language version](#).

Numeric literals can also include the `us` or `us` type character to denote the `UShort` data type, as the following example shows.

```
Dim number = &H_5826us
```

Programming tips

- **Negative Numbers.** Because `UShort` is an unsigned type, it cannot represent a negative number. If you use the unary minus (`-`) operator on an expression that evaluates to type `UShort`, Visual Basic converts the expression to `Integer` first.
- **CLS Compliance.** The `UShort` data type is not part of the [Common Language Specification](#) (CLS), so CLS-compliant code cannot consume a component that uses it.
- **Widening.** The `UShort` data type widens to `Integer`, `UInteger`, `Long`, `ULong`, `Decimal`, `Single`, and `Double`. This means you can convert `UShort` to any of these types without encountering a [System.OverflowException](#) error.
- **Type Characters.** Appending the literal type characters `us` to a literal forces it to the `UShort` data type. `UShort` has no identifier type character.
- **Framework Type.** The corresponding type in the .NET Framework is the [System.UInt16](#) structure.

See also

- [UInt16](#)
- [Data Types](#)
- [Type Conversion Functions](#)
- [Conversion Summary](#)
- [How to: Call a Windows Function that Takes Unsigned Types](#)
- [Efficient Use of Data Types](#)

Directives (Visual Basic)

8/24/2018 • 2 minutes to read • [Edit Online](#)

The topics in this section document the Visual Basic source code compiler directives.

In This Section

[#Const Directive](#) -- Define a compiler constant

[#ExternalSource Directive](#) -- Indicate a mapping between source lines and text external to the source

[#If...Then...#Else Directives](#) -- Compile selected blocks of code

[#Region Directive](#) -- Collapse and hide sections of code in the Visual Studio editor

[#Disable, #Enable](#) -- Disable and enable specific warnings for regions of code.

```
#Disable Warning BC42356 ' suppress warning about no awaits in this method
    Async Function TestAsync() As Task
        Console.WriteLine("testing")
    End Function
#Enable Warning BC42356
```

You can disable and enable a comma-separated list of warning codes too.

Related Sections

[Visual Basic Language Reference](#)

[Visual Basic](#)

#Const Directive

10/1/2019 • 2 minutes to read • [Edit Online](#)

Defines conditional compiler constants for Visual Basic.

Syntax

```
#Const constname = expression
```

Parts

constname

Required. Name of the constant being defined.

expression

Required. Literal, other conditional compiler constant, or any combination that includes any or all arithmetic or logical operators except `Is`.

Remarks

Conditional compiler constants are always private to the file in which they appear. You cannot create public compiler constants using the `#Const` directive; you can create them only in the user interface or with the `/define` compiler option.

You can use only conditional compiler constants and literals in `expression`. Using a standard constant defined with `Const` causes an error. Conversely, you can use constants defined with the `#Const` keyword only for conditional compilation. Constants can also be undefined, in which case they have a value of `Nothing`.

Example

This example uses the `#Const` directive.

```
#Const MyLocation = "USA"  
#Const Version = "8.0.0012"  
#Const CustomerNumber = 36
```

See also

- [/define \(Visual Basic\)](#)
- [#If...Then...#Else Directives](#)
- [Const Statement](#)
- [Conditional Compilation](#)
- [If...Then...Else Statement](#)

#ExternalSource Directive

10/1/2019 • 2 minutes to read • [Edit Online](#)

Indicates a mapping between specific lines of source code and text external to the source.

Syntax

```
#ExternalSource( StringLiteral , IntLiteral )
    [ LogicalLine+ ]
#End ExternalSource
```

Parts

`StringLiteral`

The path to the external source.

`IntLiteral`

The line number of the first line of the external source.

`LogicalLine`

The line where the error occurs in the external source.

`#End ExternalSource`

Terminates the `#ExternalSource` block.

Remarks

This directive is used only by the compiler and the debugger.

A source file may include external source directives, which indicate a mapping between specific lines of code in the source file and text external to the source, such as an .aspx file. If errors are encountered in the designated source code during compilation, they are identified as coming from the external source.

External source directives have no effect on compilation and cannot be nested. They are intended for internal use by the application only.

See also

- [Conditional Compilation](#)

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

10/18/2019 • 2 minutes to read • [Edit Online](#)

Conditionally compiles selected blocks of Visual Basic code.

Syntax

```
#If expression Then
    statements
[ #ElseIf expression Then
    [ statements ]
...
#ElseIf expression Then
    [ statements ] ]
[ #Else
    [ statements ] ]
#End If
```

Parts

`expression`

Required for `#If` and `#ElseIf` statements, optional elsewhere. Any expression, consisting exclusively of one or more conditional compiler constants, literals, and operators, that evaluates to `True` or `False`.

`statements`

Required for `#If` statement block, optional elsewhere. Visual Basic program lines or compiler directives that are compiled if the associated expression evaluates to `True`.

`#End If`

Terminates the `#If` statement block.

Remarks

On the surface, the behavior of the `#If...Then...#Else` directives appears the same as that of the `If...Then...Else` statements. However, the `#If...Then...#Else` directives evaluate what is compiled by the compiler, whereas the `If...Then...Else` statements evaluate conditions at run time.

Conditional compilation is typically used to compile the same program for different platforms. It is also used to prevent debugging code from appearing in an executable file. Code excluded during conditional compilation is completely omitted from the final executable file, so it has no effect on size or performance.

Regardless of the outcome of any evaluation, all expressions are evaluated using `Option Compare Binary`. The `Option Compare` statement does not affect expressions in `#If` and `#ElseIf` statements.

NOTE

No single-line form of the `#If`, `#Else`, `#ElseIf`, and `#End If` directives exists. No other code can appear on the same line as any of the directives.

The statements within a conditional compilation block must be complete logical statements. For example, you cannot conditionally compile only the attributes of a function, but you can conditionally declare the function

along with its attributes:

```
#If DEBUG Then
<WebMethod()
Public Function SomeFunction() As String
#Else
<WebMethod(CacheDuration:=86400)>
Public Function SomeFunction() As String
#End If
```

Example

This example uses the `#If...Then...#Else` construct to determine whether to compile certain statements.

```
#Const CustomerNumber = 36
#If CustomerNumber = 35 Then
    ' Insert code to be compiled for customer # 35.
#ElseIf CustomerNumber = 36 Then
    ' Insert code to be compiled for customer # 36.
#Else
    ' Insert code to be compiled for all other customers.
#End If
```

See also

- [#Const Directive](#)
- [If...Then...Else Statement](#)
- [Conditional Compilation](#)
- [System.Diagnostics.ConditionalAttribute](#)

#Region Directive

4/2/2019 • 2 minutes to read • [Edit Online](#)

Collapses and hides sections of code in Visual Basic files.

Syntax

```
#Region "identifier_string"  
#End Region
```

Parts

TERM	DEFINITION
<code>identifier_string</code>	Required. String that acts as the title of a region when it is collapsed. Regions are collapsed by default.
<code>#End Region</code>	Terminates the <code>#Region</code> block.

Remarks

Use the `#Region` directive to specify a block of code to expand or collapse when using the outlining feature of the Visual Studio Code Editor. You can place, or *nest*, regions within other regions to group similar regions together.

Example

This example uses the `#Region` directive.

```
#Region "MathFunctions"  
    ' Insert code for the Math functions here.  
#End Region
```

See also

- [#If...Then...#Else Directives](#)
- [Outlining](#)
- [How to: Collapse and Hide Sections of Code](#)

Functions (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section contain tables of the Visual Basic run-time member functions.

NOTE

You can also create functions and call them. For more information, see [Function Statement](#) and [How to: Create a Procedure that Returns a Value](#).

In This Section

[Conversion Functions](#)

[Math Functions](#)

[String Functions](#)

[Type Conversion Functions](#)

[CType Function](#)

Related Sections

[Visual Basic Language Reference](#)

[Visual Basic](#)

Conversion functions (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

- [Asc](#)
- [AscW](#)
- [CBool Function](#)
- [CByte Function](#)
- [CChar Function](#)
- [CDate Function](#)
- [CDbl Function](#)
- [CDec Function](#)
- [Chr](#)
- [ChrW](#)
- [CInt Function](#)
- [CLng Function](#)
- [CObj Function](#)
- [CSByte Function](#)
- [CShort Function](#)
- [CSng Function](#)
- [CStr Function](#)
- [CType Function](#)
- [CUInt Function](#)
- [CULng Function](#)
- [CUShort Function](#)
- [Format](#)
- [Hex](#)
- [Oct](#)
- [Str](#)
- [Val](#)

See also

- [Type Conversion Functions](#)
- [Converting Data Types](#)

Math Functions (Visual Basic)

4/2/2019 • 3 minutes to read • [Edit Online](#)

The methods of the [System.Math](#) class provide trigonometric, logarithmic, and other common mathematical functions.

Remarks

The following table lists methods of the [System.Math](#) class. You can use these in a Visual Basic program.

.NET METHOD	DESCRIPTION
Abs	Returns the absolute value of a number.
Acos	Returns the angle whose cosine is the specified number.
Asin	Returns the angle whose sine is the specified number.
Atan	Returns the angle whose tangent is the specified number.
Atan2	Returns the angle whose tangent is the quotient of two specified numbers.
BigMul	Returns the full product of two 32-bit numbers.
Ceiling	Returns the smallest integral value that's greater than or equal to the specified <code>Decimal</code> or <code>Double</code> .
Cos	Returns the cosine of the specified angle.
Cosh	Returns the hyperbolic cosine of the specified angle.
DivRem	Returns the quotient of two 32-bit or 64-bit signed integers, and also returns the remainder in an output parameter.
Exp	Returns e (the base of natural logarithms) raised to the specified power.
Floor	Returns the largest integer that's less than or equal to the specified <code>Decimal</code> or <code>Double</code> number.
IEEERemainder	Returns the remainder that results from the division of a specified number by another specified number.
Log	Returns the natural (base e) logarithm of a specified number or the logarithm of a specified number in a specified base.
Log10	Returns the base 10 logarithm of a specified number.
Max	Returns the larger of two numbers.

.NET METHOD	DESCRIPTION
Min	Returns the smaller of two numbers.
Pow	Returns a specified number raised to the specified power.
Round	Returns a <code>Decimal</code> or <code>Double</code> value rounded to the nearest integral value or to a specified number of fractional digits.
Sign	Returns an <code>Integer</code> value indicating the sign of a number.
Sin	Returns the sine of the specified angle.
Sinh	Returns the hyperbolic sine of the specified angle.
Sqrt	Returns the square root of a specified number.
Tan	Returns the tangent of the specified angle.
Tanh	Returns the hyperbolic tangent of the specified angle.
Truncate	Calculates the integral part of a specified <code>Decimal</code> or <code>Double</code> number.

To use these functions without qualification, import the `System.Math` namespace into your project by adding the following code to the top of your source file:

```
Imports System.Math
```

Example

This example uses the `Abs` method of the `Math` class to compute the absolute value of a number.

```
' Returns 50.3.
Dim MyNumber1 As Double = Math.Abs(50.3)
' Returns 50.3.
Dim MyNumber2 As Double = Math.Abs(-50.3)
```

Example

This example uses the `Atan` method of the `Math` class to calculate the value of pi.

```
Public Function GetPi() As Double
    ' Calculate the value of pi.
    Return 4.0 * Math.Atan(1.0)
End Function
```

Example

This example uses the `Cos` method of the `Math` class to return the cosine of an angle.

```
Public Function Sec(ByVal angle As Double) As Double
    ' Calculate the secant of angle, in radians.
    Return 1.0 / Math.Cos(angle)
End Function
```

Example

This example uses the [Exp](#) method of the [Math](#) class to return e raised to a power.

```
Public Function Sinh(ByVal angle As Double) As Double
    ' Calculate hyperbolic sine of an angle, in radians.
    Return (Math.Exp(angle) - Math.Exp(-angle)) / 2.0
End Function
```

Example

This example uses the [Log](#) method of the [Math](#) class to return the natural logarithm of a number.

```
Public Function Asinh(ByVal value As Double) As Double
    ' Calculate inverse hyperbolic sine, in radians.
    Return Math.Log(value + Math.Sqrt(value * value + 1.0))
End Function
```

Example

This example uses the [Round](#) method of the [Math](#) class to round a number to the nearest integer.

```
' Returns 3.
Dim MyVar2 As Double = Math.Round(2.8)
```

Example

This example uses the [Sign](#) method of the [Math](#) class to determine the sign of a number.

```
' Returns 1.
Dim MySign1 As Integer = Math.Sign(12)
' Returns -1.
Dim MySign2 As Integer = Math.Sign(-2.4)
' Returns 0.
Dim MySign3 As Integer = Math.Sign(0)
```

Example

This example uses the [Sin](#) method of the [Math](#) class to return the sine of an angle.

```
Public Function Csc(ByVal angle As Double) As Double
    ' Calculate cosecant of an angle, in radians.
    Return 1.0 / Math.Sin(angle)
End Function
```

Example

This example uses the [Sqr](#) method of the [Math](#) class to calculate the square root of a number.

```
' Returns 2.  
Dim MySqr1 As Double = Math.Sqrt(4)  
' Returns 4.79583152331272.  
Dim MySqr2 As Double = Math.Sqrt(23)  
' Returns 0.  
Dim MySqr3 As Double = Math.Sqrt(0)  
' Returns NaN (not a number).  
Dim MySqr4 As Double = Math.Sqrt(-4)
```

Example

This example uses the [Tan](#) method of the [Math](#) class to return the tangent of an angle.

```
Public Function Ctan(ByVal angle As Double) As Double  
    ' Calculate cotangent of an angle, in radians.  
    Return 1.0 / Math.Tan(angle)  
End Function
```

Requirements

Class: [Math](#)

Namespace: [System](#)

Assembly: mscorlib (in mscorlib.dll)

See also

- [Rnd](#)
- [Randomize](#)
- [NaN](#)
- [Derived Math Functions](#)
- [Arithmetic Operators](#)

String Functions (Visual Basic)

10/18/2019 • 5 minutes to read • [Edit Online](#)

The following table lists the functions that Visual Basic provides in the [Microsoft.VisualBasic.Strings](#) class to search and manipulate strings. They can be regarded as Visual Basic intrinsic functions; that is, you do not have to call them as explicit members of a class, as the examples show. Additional methods, and in some cases complementary methods, are available in the [System.String](#) class.

.NET FRAMEWORK METHOD	DESCRIPTION
Asc , AscW	Returns an <code>Integer</code> value representing the character code corresponding to a character.
Chr , ChrW	Returns the character associated with the specified character code.
Filter	Returns a zero-based array containing a subset of a <code>String</code> array based on specified filter criteria.
Format	Returns a string formatted according to instructions contained in a format <code>String</code> expression.
FormatCurrency	Returns an expression formatted as a currency value using the currency symbol defined in the system control panel.
FormatDateTime	Returns a string expression representing a date/time value.
FormatNumber	Returns an expression formatted as a number.
FormatPercent	Returns an expression formatted as a percentage (that is, multiplied by 100) with a trailing % character.
InStr	Returns an integer specifying the start position of the first occurrence of one string within another.
InStrRev	Returns the position of the first occurrence of one string within another, starting from the right side of the string.
Join	Returns a string created by joining a number of substrings contained in an array.
LCase	Returns a string or character converted to lowercase.
Left	Returns a string containing a specified number of characters from the left side of a string.
Len	Returns an integer that contains the number of characters in a string.
LSet	Returns a left-aligned string containing the specified string adjusted to the specified length.

.NET FRAMEWORK METHOD	DESCRIPTION
LTrim	Returns a string containing a copy of a specified string with no leading spaces.
Mid	Returns a string containing a specified number of characters from a string.
Replace	Returns a string in which a specified substring has been replaced with another substring a specified number of times.
Right	Returns a string containing a specified number of characters from the right side of a string.
RSet	Returns a right-aligned string containing the specified string adjusted to the specified length.
RTrim	Returns a string containing a copy of a specified string with no trailing spaces.
Space	Returns a string consisting of the specified number of spaces.
Split	Returns a zero-based, one-dimensional array containing a specified number of substrings.
StrComp	Returns -1, 0, or 1, based on the result of a string comparison.
StrConv	Returns a string converted as specified.
StrDup	Returns a string or object consisting of the specified character repeated the specified number of times.
StrReverse	Returns a string in which the character order of a specified string is reversed.
Trim	Returns a string containing a copy of a specified string with no leading or trailing spaces.
UCase	Returns a string or character containing the specified string converted to uppercase.

You can use the [Option Compare](#) statement to set whether strings are compared using a case-insensitive text sort order determined by your system's locale (`Text`) or by the internal binary representations of the characters (`Binary`). The default text comparison method is `Binary`.

Example: UCase

This example uses the `UCase` function to return an uppercase version of a string.

```
' String to convert.
Dim lowerCase As String = "Hello World 1234"
' Returns "HELLO WORLD 1234".
Dim upperCase As String = UCase(lowerCase)
```

Example: LTrim

This example uses the `LTrim` function to strip leading spaces and the `RTrim` function to strip trailing spaces from a string variable. It uses the `Trim` function to strip both types of spaces.

```
' Initializes string.  
Dim testString As String = " <-Trim-> "  
Dim trimString As String  
' Returns "<-Trim-> ".  
trimString = LTrim(testString)  
' Returns " <-Trim-> ".  
trimString = RTrim(testString)  
' Returns "<-Trim-> ".  
trimString = LTrim(RTrim(testString))  
' Using the Trim function alone achieves the same result.  
' Returns "<-Trim-> ".  
trimString = Trim(testString)
```

Example: Mid

This example uses the `Mid` function to return a specified number of characters from a string.

```
' Creates text string.  
Dim testString As String = "Mid Function Demo"  
' Returns "Mid".  
Dim firstWord As String = Mid(testString, 1, 3)  
' Returns "Demo".  
Dim lastWord As String = Mid(testString, 14, 4)  
' Returns "Function Demo".  
Dim midWords As String = Mid(testString, 5)
```

Example: Len

This example uses `Len` to return the number of characters in a string.

```
' Initializes variable.  
Dim testString As String = "Hello World"  
' Returns 11.  
Dim testLen As Integer = Len(testString)
```

Example: InStr

This example uses the `InStr` function to return the position of the first occurrence of one string within another.

```

' String to search in.
Dim searchString As String = "XXpXXpXXPXXP"
' Search for "P".
Dim searchChar As String = "P"

Dim testPos As Integer
' A textual comparison starting at position 4. Returns 6.
testPos = InStr(4, searchString, searchChar, CompareMethod.Text)

' A binary comparison starting at position 1. Returns 9.
testPos = InStr(1, searchString, searchChar, CompareMethod.Binary)

' If Option Compare is not set, or set to Binary, return 9.
' If Option Compare is set to Text, returns 3.
testPos = InStr(searchString, searchChar)

' Returns 0.
testPos = InStr(1, searchString, "W")

```

Example: Format

This example shows various uses of the `Format` function to format values using both `string` formats and user-defined formats. For the date separator (`/`), time separator (`:`), and the AM/PM indicators (`t` and `tt`), the actual formatted output displayed by your system depends on the locale settings the code is using. When times and dates are displayed in the development environment, the short time format and short date format of the code locale are used.

NOTE

For locales that use a 24-hour clock, the AM/PM indicators (`t` and `tt`) display nothing.

```

Dim testDateTime As Date = #1/27/2001 5:04:23 PM#
Dim testStr As String
' Returns current system time in the system-defined long time format.
testStr = Format(Now(), "Long Time")
' Returns current system date in the system-defined long date format.
testStr = Format(Now(), "Long Date")
' Also returns current system date in the system-defined long date
' format, using the single letter code for the format.
testStr = Format(Now(), "D")

' Returns the value of testDateTime in user-defined date/time formats.
' Returns "5:4:23".
testStr = Format(testDateTime, "h:m:s")
' Returns "05:04:23 PM".
testStr = Format(testDateTime, "hh:mm:ss tt")
' Returns "Saturday, Jan 27 2001".
testStr = Format(testDateTime, "dddd, MMM d yyyy")
' Returns "17:04:23".
testStr = Format(testDateTime, "HH:mm:ss")
' Returns "23".
testStr = Format(23)

' User-defined numeric formats.
' Returns "5,459.40".
testStr = Format(5459.4, "##,##0.00")
' Returns "334.90".
testStr = Format(334.9, "###0.00")
' Returns "500.00%".
testStr = Format(5, "0.00%")

```

See also

- [Keywords](#)
- [Visual Basic Runtime Library Members](#)
- [String Manipulation Summary](#)
- [System.String class methods](#)

Type Conversion Functions (Visual Basic)

10/18/2019 • 12 minutes to read • [Edit Online](#)

These functions are compiled inline, meaning the conversion code is part of the code that evaluates the expression. Sometimes there is no call to a procedure to accomplish the conversion, which improves performance. Each function coerces an expression to a specific data type.

Syntax

```
CBool(expression)
CByte(expression)
CChar(expression)
CDate(expression)
CDbl(expression)
CDec(expression)
CInt(expression)
CLng(expression)
CObj(expression)
CSByte(expression)
CShort(expression)
CSng(expression)
CStr(expression)
CUInt(expression)
CULng(expression)
CUShort(expression)
```

Part

`expression`

Required. Any expression of the source data type.

Return Value Data Type

The function name determines the data type of the value it returns, as shown in the following table.

FUNCTION NAME	RETURN DATA TYPE	RANGE FOR <small>EXPRESSION ARGUMENT</small>
<code>CBool</code>	Boolean Data Type	Any valid <code>Char</code> or <code>String</code> or numeric expression.

FUNCTION NAME	RETURN DATA TYPE	RANGE FOR EXPRESSION ARGUMENT
<code>CInt</code>	Integer Data Type	<p><code>Int32.MinValue</code> (-2,147,483,648) through <code>Int32.MaxValue</code> (2,147,483,647); fractional parts are rounded.¹</p> <p>Starting with Visual Basic 15.8, Visual Basic optimizes the performance of floating-point to integer conversion with the <code>CInt</code> function; see the Remarks section for more information. See the CInt Example section for an example.</p>
<code>CLng</code>	Long Data Type	<p><code>Int64.MinValue</code> (-9,223,372,036,854,775,808) through <code>Int64.MaxValue</code> (9,223,372,036,854,775,807); fractional parts are rounded.¹</p> <p>Starting with Visual Basic 15.8, Visual Basic optimizes the performance of floating-point to 64-bit integer conversion with the <code>CLng</code> function; see the Remarks section for more information. See the CInt Example section for an example.</p>
<code>COBJ</code>	Object Data Type	Any valid expression.
<code>CSByte</code>	SByte Data Type	<p><code>SByte.MinValue</code> (-128) through <code>SByte.MaxValue</code> (127); fractional parts are rounded.¹</p> <p>Starting with Visual Basic 15.8, Visual Basic optimizes the performance of floating-point to signed byte conversion with the <code>csbyte</code> function; see the Remarks section for more information. See the CInt Example section for an example.</p>

FUNCTION NAME	RETURN DATA TYPE	RANGE FOR EXPRESSION ARGUMENT
<code>CShort</code>	Short Data Type	<p><code>Int16.MinValue</code> (-32,768) through <code>Int16.MaxValue</code> (32,767); fractional parts are rounded.¹</p> <p>Starting with Visual Basic 15.8, Visual Basic optimizes the performance of floating-point to 16-bit integer conversion with the <code>CShort</code> function; see the Remarks section for more information. See the CInt Example section for an example.</p>
<code>CSng</code>	Single Data Type	<p>-3.402823E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.402823E+38 for positive values.</p>
<code>CStr</code>	String Data Type	<p>Returns for <code>cstr</code> depend on the <code>expression</code> argument. See Return Values for the CStr Function.</p>
<code>CUInt</code>	UInteger Data Type	<p><code>UInt32.MinValue</code> (0) through <code>UInt32.MaxValue</code> (4,294,967,295) (unsigned); fractional parts are rounded.¹</p> <p>Starting with Visual Basic 15.8, Visual Basic optimizes the performance of floating-point to unsigned integer conversion with the <code>CUInt</code> function; see the Remarks section for more information. See the CInt Example section for an example.</p>
<code>CULng</code>	ULong Data Type	<p><code>UInt64.MinValue</code> (0) through <code>UInt64.MaxValue</code> (18,446,744,073,709,551,615) (unsigned); fractional parts are rounded.¹</p> <p>Starting with Visual Basic 15.8, Visual Basic optimizes the performance of floating-point to unsigned long integer conversion with the <code>CULng</code> function; see the Remarks section for more information. See the CInt Example section for an example.</p>

FUNCTION NAME	RETURN DATA TYPE	RANGE FOR EXPRESSION ARGUMENT
<code>CUShort</code>	UShort Data Type	<p><code>UInt16.MinValue</code> (0) through <code>UInt16.MaxValue</code> (65,535) (unsigned); fractional parts are rounded.¹</p> <p>Starting with Visual Basic 15.8, Visual Basic optimizes the performance of floating-point to unsigned 16-bit integer conversion with the <code>CUShort</code> function; see the Remarks section for more information. See the CInt Example section for an example.</p>

¹ Fractional parts can be subject to a special type of rounding called *banker's rounding*. See "Remarks" for more information.

Remarks

As a rule, you should use the Visual Basic type conversion functions in preference to the .NET Framework methods such as `ToString()`, either on the `Convert` class or on an individual type structure or class. The Visual Basic functions are designed for optimal interaction with Visual Basic code, and they also make your source code shorter and easier to read. In addition, the .NET Framework conversion methods do not always produce the same results as the Visual Basic functions, for example when converting `Boolean` to `Integer`. For more information, see [Troubleshooting Data Types](#).

Starting with Visual Basic 15.8, the performance of floating-point-to-integer conversion is optimized when you pass the `Single` or `Double` value returned by the following methods to one of the integer conversion functions (`CByte`, `CShort`, `CInt`, `CLng`, `CSByte`, `CUShort`, `CUInt`, `CULng`):

- [Conversion.Fix\(Double\)](#)
- [Conversion.Fix\(Object\)](#)
- [Conversion.Fix\(Single\)](#)
- [Conversion.Int\(Double\)](#)
- [Conversion.Int\(Object\)](#)
- [Conversion.Int\(Single\)](#)
- [Math.Ceiling\(Double\)](#)
- [Math.Floor\(Double\)](#)
- [Math.Round\(Double\)](#)
- [Math.Truncate\(Double\)](#)

This optimization allows code that does a large number of integer conversions to run up to twice as fast. The following example illustrates these optimized floating-point-to-integer conversions:

```

Dim s As Single = 173.7619
Dim d As Double = s

Dim i1 As Integer = CInt(Fix(s))           ' Result: 173
Dim b1 As Byte = CByte(Int(d))             ' Result: 173
Dim s1 AS Short = CShort(Math.Truncate(s)) ' Result: 173
Dim i2 As Integer = CInt(Math.Ceiling(d))   ' Result: 174
Dim i3 As Integer = CInt(Math.Round(s))     ' Result: 174

```

Behavior

- **Coercion.** In general, you can use the data type conversion functions to coerce the result of an operation to a particular data type rather than the default data type. For example, use `CDec` to force decimal arithmetic in cases where single-precision, double-precision, or integer arithmetic would normally take place.
- **Failed Conversions.** If the `expression` passed to the function is outside the range of the data type to which it is to be converted, an `OverflowException` occurs.
- **Fractional Parts.** When you convert a nonintegral value to an integral type, the integer conversion functions (`CByte`, `CInt`, `CLng`, `CSByte`, `CShort`, `CUInt`, `CULng`, and `CUShort`) remove the fractional part and round the value to the closest integer.

If the fractional part is exactly 0.5, the integer conversion functions round it to the nearest even integer. For example, 0.5 rounds to 0, and 1.5 and 2.5 both round to 2. This is sometimes called *banker's rounding*, and its purpose is to compensate for a bias that could accumulate when adding many such numbers together.

`CInt` and `CLng` differ from the `Int` and `Fix` functions, which truncate, rather than round, the fractional part of a number. Also, `Fix` and `Int` always return a value of the same data type as you pass in.

- **Date/Time Conversions.** Use the `IsDate` function to determine if a value can be converted to a date and time. `CDate` recognizes date literals and time literals but not numeric values. To convert a Visual Basic 6.0 `Date` value to a `Date` value in Visual Basic 2005 or later versions, you can use the `DateTime.FromOADate` method.
- **Neutral Date/Time Values.** The `Date Data Type` always contains both date and time information. For purposes of type conversion, Visual Basic considers 1/1/0001 (January 1 of the year 1) to be a *neutral value* for the date, and 00:00:00 (midnight) to be a neutral value for the time. If you convert a `Date` value to a string, `cstr` does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original `Date` value and can be recovered with functions such as `DatePart` function.
- **Culture Sensitivity.** The type conversion functions involving strings perform conversions based on the current culture settings for the application. For example, `CDate` recognizes date formats according to the locale setting of your system. You must provide the day, month, and year in the correct order for your locale, or the date might not be interpreted correctly. A long date format is not recognized if it contains a day-of-the-week string, such as "Wednesday".

If you need to convert to or from a string representation of a value in a format other than the one specified by your locale, you cannot use the Visual Basic type

conversion functions. To do this, use the `ToString(IFormatProvider)` and `Parse(String, IFormatProvider)` methods of that value's type. For example, use `Double.Parse` when converting a string to a `Double`, and use `Double.ToString` when converting a value of type `Double` to a string.

CType Function

The [CType Function](#) takes a second argument, `typename`, and coerces `expression` to `typename`, where `typename` can be any data type, structure, class, or interface to which there exists a valid conversion.

For a comparison of `cType` with the other type conversion keywords, see [DirectCast Operator](#) and [TryCast Operator](#).

CBool Example

The following example uses the `CBool` function to convert expressions to `Boolean` values. If an expression evaluates to a nonzero value, `CBool` returns `True`; otherwise, it returns `False`.

```
Dim a, b, c As Integer
Dim check As Boolean
a = 5
b = 5
' The following line of code sets check to True.
check = CBool(a = b)
c = 0
' The following line of code sets check to False.
check = CBool(c)
```

CByte Example

The following example uses the `CByte` function to convert an expression to a `Byte`.

```
Dim aDouble As Double
Dim aByte As Byte
aDouble = 125.5678
' The following line of code sets aByte to 126.
aByte = CByte(aDouble)
```

CChar Example

The following example uses the `CChar` function to convert the first character of a `String` expression to a `Char` type.

```
Dim aString As String
Dim aChar As Char
' CChar converts only the first character of the string.
aString = "BCD"
' The following line of code sets aChar to "B".
aChar = CChar(aString)
```

The input argument to `CChar` must be of data type `Char` or `String`. You cannot use `CChar` to convert a number to a character, because `CChar` cannot accept a numeric data type. The

following example obtains a number representing a code point (character code) and converts it to the corresponding character. It uses the `InputBox` function to obtain the string of digits, `CInt` to convert the string to type `Integer`, and `ChrW` to convert the number to type `Char`.

```
Dim someDigits As String
Dim codePoint As Integer
Dim thisChar As Char
someDigits = InputBox("Enter code point of character:")
codePoint = CInt(someDigits)
' The following line of code sets thisChar to the Char value of codePoint.
thisChar = ChrW(codePoint)
```

CDate Example

The following example uses the `CDate` function to convert strings to `Date` values. In general, hard-coding dates and times as strings (as shown in this example) is not recommended. Use date literals and time literals, such as #Feb 12, 1969# and #4:45:23 PM#, instead.

```
Dim aDateString, aTimeString As String
Dim aDate, aTime As Date
aDateString = "February 12, 1969"
aTimeString = "4:35:47 PM"
' The following line of code sets aDate to a Date value.
aDate = CDate(aDateString)
' The following line of code sets aTime to Date value.
aTime = CDate(aTimeString)
```

CDbl Example

```
Dim aDec As Decimal
Dim aDbl As Double
' The following line of code uses the literal type character D to make aDec a Decimal.
aDec = 234.456784D
' The following line of code sets aDbl to 1.9225456288E+1.
aDbl = CDbl(aDec * 8.2D * 0.01D)
```

CDec Example

The following example uses the `CDec` function to convert a numeric value to `Decimal`.

```
Dim aDouble As Double
Dim aDecimal As Decimal
aDouble = 10000000.0587
' The following line of code sets aDecimal to 10000000.0587.
aDecimal = CDec(aDouble)
```

CInt Example

The following example uses the `CInt` function to convert a value to `Integer`.

```
Dim aDbl As Double
Dim anInt As Integer
aDbl = 2345.5678
' The following line of code sets anInt to 2346.
anInt = CInt(aDbl)
```

CLng Example

The following example uses the `cLng` function to convert values to `Long`.

```
Dim aDbl1, aDbl2 As Double
Dim aLng1, aLng2 As Long
aDbl1 = 25427.45
aDbl2 = 25427.55
' The following line of code sets aLng1 to 25427.
aLng1 = CLng(aDbl1)
' The following line of code sets aLng2 to 25428.
aLng2 = CLng(aDbl2)
```

CObj Example

The following example uses the `cObj` function to convert a numeric value to `Object`. The `Object` variable itself contains only a four-byte pointer, which points to the `Double` value assigned to it.

```
Dim aDouble As Double
Dim anObject As Object
aDouble = 2.7182818284
' The following line of code sets anObject to a pointer to aDouble.
anObject = CObj(aDouble)
```

CSByte Example

The following example uses the `csByte` function to convert a numeric value to `SByte`.

```
Dim aDouble As Double
Dim anSByte As SByte
aDouble = 39.501
' The following line of code sets anSByte to 40.
anSByte = CSByte(aDouble)
```

CShort Example

The following example uses the `cShort` function to convert a numeric value to `Short`.

```
Dim aByte As Byte
Dim aShort As Short
aByte = 100
' The following line of code sets aShort to 100.
aShort = CShort(aByte)
```

CSng Example

The following example uses the `CSng` function to convert values to `Single`.

```
Dim aDouble1, aDouble2 As Double
Dim aSingle1, aSingle2 As Single
aDouble1 = 75.3421105
aDouble2 = 75.3421567
' The following line of code sets aSingle1 to 75.34211.
aSingle1 = CSng(aDouble1)
' The following line of code sets aSingle2 to 75.34216.
aSingle2 = CSng(aDouble2)
```

CStr Example

The following example uses the `CStr` function to convert a numeric value to `String`.

```
Dim aDouble As Double
Dim aString As String
aDouble = 437.324
' The following line of code sets aString to "437.324".
aString = CStr(aDouble)
```

The following example uses the `CStr` function to convert `Date` values to `String` values.

```
Dim aDate As Date
Dim aString As String
' The following line of code generates a COMPILER ERROR because of invalid format.
' aDate = #February 12, 1969 00:00:00#
' Date literals must be in the format #m/d/yyyy# or they are invalid.
' The following line of code sets the time component of aDate to midnight.
aDate = #2/12/1969#
' The following conversion suppresses the neutral time value of 00:00:00.
' The following line of code sets aString to "2/12/1969".
aString = CStr(aDate)
' The following line of code sets the time component of aDate to one second past
midnight.
aDate = #2/12/1969 12:00:01 AM#
' The time component becomes part of the converted value.
' The following line of code sets aString to "2/12/1969 12:00:01 AM".
aString = CStr(aDate)
```

`CStr` always renders a `Date` value in the standard short format for the current locale, for example, "6/15/2003 4:35:47 PM". However, `CStr` suppresses the *neutral values* of 1/1/0001 for the date and 00:00:00 for the time.

For more detail on the values returned by `CStr`, see [Return Values for the CStr Function](#).

CUInt Example

The following example uses the `CUInt` function to convert a numeric value to `UInteger`.

```
Dim aDouble As Double
Dim aUInteger As UInteger
aDouble = 39.501
' The following line of code sets aUInteger to 40.
aUInteger = CUInt(aDouble)
```

CULng Example

The following example uses the `CULng` function to convert a numeric value to `ULong`.

```
Dim aDouble As Double
Dim aULong As ULong
aDouble = 39.501
' The following line of code sets aULong to 40.
aULong = CULng(aDouble)
```

CUShort Example

The following example uses the `cushort` function to convert a numeric value to `UShort`.

```
Dim aDouble As Double
Dim aUShort As UShort
aDouble = 39.501
' The following line of code sets aUShort to 40.
aUShort = CUShort(aDouble)
```

See also

- [Asc](#)
- [AscW](#)
- [Chr](#)
- [ChrW](#)
- [Int](#)
- [Fix](#)
- [Format](#)
- [Hex](#)
- [Oct](#)
- [Str](#)
- [Val](#)
- [Conversion Functions](#)
- [Type Conversions in Visual Basic](#)

Return Values for the CStr Function (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

The following table describes the return values for `cstr` for different data types of `expression`.

IF <code>expression</code> TYPE IS	<code>CSTR</code> RETURNS
Boolean Data Type	A string containing "True" or "False".
Date Data Type	A string containing a <code>Date</code> value (date and time) in the short date format of your system.
Numeric Data Types	A string representing the number.

CStr and Date

The `Date` type always contains both date and time information. For purposes of type conversion, Visual Basic considers 1/1/0001 (January 1 of the year 1) to be a *neutral value* for the date, and 00:00:00 (midnight) to be a neutral value for the time. `cstr` does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original `Date` value and can be recovered with functions such as [DatePart](#).

NOTE

The `cstr` function performs its conversion based on the current culture settings for the application. To get the string representation of a number in a particular culture, use the number's `ToString(IFormatProvider)` method. For example, use `Double.ToString` when converting a value of type `Double` to a `String`.

See also

- [DatePart](#)
- [Type Conversion Functions](#)
- [Boolean Data Type](#)
- [Date Data Type](#)

CType Function (Visual Basic)

9/28/2019 • 2 minutes to read • [Edit Online](#)

Returns the result of explicitly converting an expression to a specified data type, object, structure, class, or interface.

Syntax

```
CType(expression, typename)
```

Parts

`expression` Any valid expression. If the value of `expression` is outside the range allowed by `typename`, Visual Basic throws an exception.

`typename` Any expression that is legal within an `As` clause in a `Dim` statement, that is, the name of any data type, object, structure, class, or interface.

Remarks

TIP

You can also use the following functions to perform a type conversion:

- Type conversion functions such as `CByte`, `CDbl`, and `CInt` that perform a conversion to a specific data type. For more information, see [Type Conversion Functions](#).
- [DirectCast Operator](#) or [TryCast Operator](#). These operators require that one type inherit from or implement the other type. They can provide somewhat better performance than `CType` when converting to and from the `Object` data type.

`CType` is compiled inline, which means that the conversion code is part of the code that evaluates the expression. In some cases, the code runs faster because no procedures are called to perform the conversion.

If no conversion is defined from `expression` to `typename` (for example, from `Integer` to `Date`), Visual Basic displays a compile-time error message.

If a conversion fails at run time, the appropriate exception is thrown. If a narrowing conversion fails, an [OverflowException](#) is the most common result. If the conversion is undefined, an [InvalidOperationException](#) is thrown. For example, this can happen if `expression` is of type `Object` and its run-time type has no conversion to `typename`.

If the data type of `expression` or `typename` is a class or structure you've defined, you can define `CType` on that class or structure as a conversion operator. This makes `CType` act as an *overloaded operator*. If you do this, you can control the behavior of conversions to and from your class or structure, including the exceptions that can be thrown.

Overloading

The `CType` operator can also be overloaded on a class or structure defined outside your code. If your code

converts to or from such a class or structure, be sure you understand the behavior of its `CType` operator. For more information, see [Operator Procedures](#).

Converting Dynamic Objects

Type conversions of dynamic objects are performed by user-defined dynamic conversions that use the `TryConvert` or `BindConvert` methods. If you're working with dynamic objects, use the `CTypeDynamic` method to convert the dynamic object.

Example

The following example uses the `CType` function to convert an expression to the `Single` data type.

```
Dim testNumber As Long = 1000
' The following line of code sets testNewType to 1000.0.
Dim testNewType As Single = CType(testNumber, Single)
```

For additional examples, see [Implicit and Explicit Conversions](#).

See also

- [OverflowException](#)
- [InvalidOperationException](#)
- [Type Conversion Functions](#)
- [Conversion Functions](#)
- [Operator Statement](#)
- [How to: Define a Conversion Operator](#)
- [Type Conversion in the .NET Framework](#)

Modifiers (Visual Basic)

10/24/2018 • 2 minutes to read • [Edit Online](#)

The topics in this section document Visual Basic run-time modifiers.

In This Section

[Ansi](#)

[Assembly](#)

[Async](#)

[Auto](#)

[ByRef](#)

[ByVal](#)

[Default](#)

[Friend](#)

[In](#)

[Iterator](#)

[Key](#)

[Module <keyword>](#)

[MustInherit](#)

[MustOverride](#)

[Narrowing](#)

[NotInheritable](#)

[NotOverridable](#)

[Optional](#)

[Out](#)

[Overloads](#)

[Overridable](#)

[Overrides](#)

[ParamArray](#)

[Partial](#)

[Private](#)

[Private Protected](#)

[Protected](#)

[Protected Friend](#)

[Public](#)

[ReadOnly](#)

[Shadows](#)

[Shared](#)

[Static](#)

[Unicode](#)

[Widening](#)

[WithEvents](#)

[WriteOnly](#)

Related Sections

[Visual Basic Language Reference](#)

[Visual Basic](#)

Ansi (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Specifies that Visual Basic should marshal all strings to American National Standards Institute (ANSI) values regardless of the name of the external procedure being declared.

When you call a procedure defined outside your project, the Visual Basic compiler does not have access to the information it needs to call the procedure correctly. This information includes where the procedure is located, how it is identified, its calling sequence and return type, and the string character set it uses. The [Declare Statement](#) creates a reference to an external procedure and supplies this necessary information.

The `charsetmodifier` part in the `Declare` statement supplies the character set information for marshaling strings during a call to the external procedure. It also affects how Visual Basic searches the external file for the external procedure name. The `Ansi` modifier specifies that Visual Basic should marshal all strings to ANSI values and should look up the procedure without modifying its name during the search.

If no character set modifier is specified, `Ansi` is the default.

Remarks

The `Ansi` modifier can be used in this context:

[Declare Statement](#)

Smart Device Developer Notes

This keyword is not supported.

See also

- [Auto](#)
- [Unicode](#)
- [Keywords](#)

Assembly (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Specifies that an attribute at the beginning of a source file applies to the entire assembly.

Remarks

Many attributes pertain to an individual programming element, such as a class or property. You apply such an attribute by attaching the attribute block, within angle brackets (`<>`), directly to the declaration statement.

If an attribute pertains not only to the following element but to the entire assembly, you place the attribute block at the beginning of the source file and identify the attribute with the `Assembly` keyword. If it applies to the current assembly module, you use the `Module` keyword.

You can also apply an attribute to an assembly in the `AssemblyInfo.vb` file, in which case you do not have to use an attribute block in your main source-code file.

See also

- [Module <keyword>](#)
- [Attributes overview](#)

Async (Visual Basic)

10/17/2019 • 3 minutes to read • [Edit Online](#)

The `Async` modifier indicates that the method or [lambda expression](#) that it modifies is asynchronous. Such methods are referred to as *async methods*.

An `async` method provides a convenient way to do potentially long-running work without blocking the caller's thread. The caller of an `async` method can resume its work without waiting for the `async` method to finish.

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](#).

The following example shows the structure of an `async` method. By convention, `async` method names end in "Async."

```
Public Async Function ExampleMethodAsync() As Task(Of Integer)
    ' ...
    ' At the Await expression, execution in this method is suspended and,
    ' if AwaitedProcessAsync has not already finished, control returns
    ' to the caller of ExampleMethodAsync. When the awaited task is
    ' completed, this method resumes execution.
    Dim exampleInt As Integer = Await AwaitedProcessAsync()
    ' ...
    ' The return statement completes the task. Any method that is
    ' awaiting ExampleMethodAsync can now get the integer result.
    Return exampleInt
End Function
```

Typically, a method modified by the `Async` keyword contains at least one `Await` expression or statement. The method runs synchronously until it reaches the first `Await`, at which point it suspends until the awaited task completes. In the meantime, control is returned to the caller of the method. If the method doesn't contain an `Await` expression or statement, the method isn't suspended and executes as a synchronous method does. A compiler warning alerts you to any `async` methods that don't contain `Await` because that situation might indicate an error. For more information, see the [compiler error](#).

The `Async` keyword is an unreserved keyword. It is a keyword when it modifies a method or a lambda expression. In all other contexts, it is interpreted as an identifier.

Return Types

An `async` method is either a `Sub` procedure, or a `Function` procedure that has a return type of `Task` or `Task<TResult>`. The method cannot declare any `ByRef` parameters.

You specify `Task(Of TResult)` for the return type of an `async` method if the `Return` statement of the method has an operand of type `TResult`. You use `Task` if no meaningful value is returned when the method is completed. That is, a call to the method returns a `Task`, but when the `Task` is completed, any `Await` statement that's awaiting the `Task` doesn't produce a result value.

Async subroutines are used primarily to define event handlers where a `Sub` procedure is required. The caller of an async subroutine can't await it and can't catch exceptions that the method throws.

For more information and examples, see [Async Return Types](#).

Example

The following examples show an async event handler, an async lambda expression, and an async method. For a full example that uses these elements, see [Walkthrough: Accessing the Web by Using Async and Await](#). You can download the walkthrough code from [Developer Code Samples](#).

```
' An event handler must be a Sub procedure.  
Async Sub button1_Click(sender As Object, e As RoutedEventArgs) Handles button1.Click  
    textBox1.Clear()  
    ' SumPageSizesAsync is a method that returns a Task.  
    Await SumPageSizesAsync()  
    textBox1.Text = vbCrLf & "Control returned to button1_Click."  
End Sub  
  
' The following async lambda expression creates an equivalent anonymous  
' event handler.  
AddHandler button1.Click, Async Sub(sender, e)  
    textBox1.Clear()  
    ' SumPageSizesAsync is a method that returns a Task.  
    Await SumPageSizesAsync()  
    textBox1.Text = vbCrLf & "Control returned to button1_Click."  
End Sub  
  
' The following async method returns a Task(Of T).  
' A typical call awaits the Byte array result:  
'     Dim result As Byte() = Await GetURLContents("https://msdn.com")  
Private Async Function GetURLContentsAsync(url As String) As Task(Of Byte())  
  
    ' The downloaded resource ends up in the variable named content.  
    Dim content = New MemoryStream()  
  
    ' Initialize an HttpWebRequest for the current URL.  
    Dim webReq = CType(WebRequest.Create(url), HttpWebRequest)  
  
    ' Send the request to the Internet resource and wait for  
    ' the response.  
    Using response AsWebResponse = Await webReq.GetResponseAsync()  
        ' Get the data stream that is associated with the specified URL.  
        Using responseStream As Stream = response.GetResponseStream()  
            ' Read the bytes in responseStream and copy them to content.  
            ' CopyToAsync returns a Task, not a Task<T>.  
            Await responseStream.CopyToAsync(content)  
        End Using  
    End Using  
  
    ' Return the result as a byte array.  
    Return content.ToArray()  
End Function
```

See also

- [AsyncStateMachineAttribute](#)
- [Await Operator](#)
- [Asynchronous Programming with Async and Await](#)
- [Walkthrough: Accessing the Web by Using Async and Await](#)

Auto (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Specifies that Visual Basic should marshal strings according to .NET Framework rules based on the external name of the external procedure being declared.

When you call a procedure defined outside your project, the Visual Basic compiler does not have access to the information it must have to call the procedure correctly. This information includes where the procedure is located, how it is identified, its calling sequence and return type, and the string character set it uses. The [Declare Statement](#) creates a reference to an external procedure and supplies this necessary information.

The `charsetmodifier` part in the `Declare` statement supplies the character set information for marshaling strings during a call to the external procedure. It also affects how Visual Basic searches the external file for the external procedure name. The `Auto` modifier specifies that Visual Basic should marshal strings according to .NET Framework rules, and that it should determine the base character set of the run-time platform and possibly modify the external procedure name if the initial search fails. For more information, see "Character Sets" in [Declare Statement](#).

If no character set modifier is specified, `Ansi` is the default.

Remarks

The `Auto` modifier can be used in this context:

[Declare Statement](#)

Smart Device Developer Notes

This keyword is not supported.

See also

- [Ansi](#)
- [Unicode](#)
- [Keywords](#)

ByRef (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Specifies that an argument is passed in such a way that the called procedure can change the value of a variable underlying the argument in the calling code.

Remarks

The `ByRef` modifier can be used in these contexts:

[Declare Statement](#)

[Function Statement](#)

[Sub Statement](#)

See also

- [Keywords](#)
- [Passing Arguments by Value and by Reference](#)

ByVal (Visual Basic)

8/21/2019 • 2 minutes to read • [Edit Online](#)

Specifies that an argument is passed [by value](#), so that the called procedure or property cannot change the value of a variable underlying the argument in the calling code. If no modifier is specified, ByVal is the default.

NOTE

Because it is the default, you do not have to explicitly specify the `ByVal` keyword in method signatures. It tends to produce noisy code and often leads to the non-default `ByRef` keyword being overlooked.

Remarks

The `ByVal` modifier can be used in these contexts:

[Declare Statement](#)

[Function Statement](#)

[Operator Statement](#)

[Property Statement](#)

[Sub Statement](#)

Example

The following example demonstrates the use of the `ByVal` parameter passing mechanism with a reference type argument. In the example, the argument is `c1`, an instance of class `Class1`. `ByVal` prevents the code in the procedures from changing the underlying value of the reference argument, `c1`, but does not protect the accessible fields and properties of `c1`.

```
Module Module1

    Sub Main()

        ' Declare an instance of the class and assign a value to its field.
        Dim c1 As New Class1()
        c1.Field = 5
        Console.WriteLine(c1.Field)
        ' Output: 5

        ' ByVal does not prevent changing the value of a field or property.
        ChangeFieldValue(c1)
        Console.WriteLine(c1.Field)
        ' Output: 500

        ' ByVal does prevent changing the value of c1 itself.
        ChangeClassReference(c1)
        Console.WriteLine(c1.Field)
        ' Output: 500

        Console.ReadKey()
    End Sub

    Public Sub ChangeFieldValue(ByVal cls As Class1)
        cls.Field = 500
    End Sub

    Public Sub ChangeClassReference(ByVal cls As Class1)
        cls = New Class1()
        cls.Field = 1000
    End Sub

    Public Class Class1
        Public Field As Integer
    End Class

End Module
```

See also

- [Keywords](#)
- [Passing Arguments by Value and by Reference](#)

Default (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Identifies a property as the default property of its class, structure, or interface.

Remarks

A class, structure, or interface can designate at most one of its properties as the *default property*, provided that property takes at least one parameter. If code makes a reference to a class or structure without specifying a member, Visual Basic resolves that reference to the default property.

Default properties can result in a small reduction in source code-characters, but they can make your code more difficult to read. If the calling code is not familiar with your class or structure, when it makes a reference to the class or structure name it cannot be certain whether that reference accesses the class or structure itself, or a default property. This can lead to compiler errors or subtle run-time logic errors.

You can somewhat reduce the chance of default property errors by always using the [Option Strict Statement](#) to set compiler type checking to `On`.

If you are planning to use a predefined class or structure in your code, you must determine whether it has a default property, and if so, what its name is.

Because of these disadvantages, you should consider not defining default properties. For code readability, you should also consider always referring to all properties explicitly, even default properties.

The `Default` modifier can be used in this context:

[Property Statement](#)

See also

- [How to: Declare and Call a Default Property in Visual Basic](#)
- [Keywords](#)

Friend (Visual Basic)

9/13/2019 • 2 minutes to read • [Edit Online](#)

Specifies that one or more declared programming elements are accessible only from within the assembly that contains their declaration.

Remarks

In many cases, you want programming elements such as classes and structures to be used by the entire assembly, not only by the component that declares them. However, you might not want them to be accessible by code outside the assembly (for example, if the application is proprietary). If you want to limit access to an element in this way, you can declare it by using the `Friend` modifier.

Code in other classes, structures, and modules that are compiled to the same assembly can access all the `Friend` elements in that assembly.

`Friend` access is often the preferred level for an application's programming elements, and `Friend` is the default access level of an interface, a module, a class, or a structure.

You can use `Friend` only at the module, interface, or namespace level. Therefore, the declaration context for a `Friend` element must be a source file, a namespace, an interface, a module, a class, or a structure; it can't be a procedure.

NOTE

You can also use the `Protected Friend` access modifier, which makes a class member accessible from within that class, from derived classes, and from the same assembly in which the class is defined. To restrict access to a member from within its class and from derived classes in the same assembly, you use the `Private Protected` access modifier.

For a comparison of `Friend` and the other access modifiers, see [Access levels in Visual Basic](#).

NOTE

You can specify that another assembly is a friend assembly, which allows it to access all types and members that are marked as `Friend`. For more information, see [Friend Assemblies](#).

Example

The following class uses the `Friend` modifier to allow other programming elements within the same assembly to access certain members.

```

Class CustomerInfo

    Private p_CustomerID As Integer

    Public ReadOnly Property CustomerID() As Integer
        Get
            Return p_CustomerID
        End Get
    End Property

    ' Allow friend access to the empty constructor.
    Friend Sub New()

    End Sub

    ' Require that a customer identifier be specified for the public constructor.
    Public Sub New(ByVal customerID As Integer)
        p_CustomerID = customerID
    End Sub

    ' Allow friend programming elements to set the customer identifier.
    Friend Sub SetCustomerID(ByVal customerID As Integer)
        p_CustomerID = customerID
    End Sub
End Class

```

Usage

You can use the `Friend` modifier in these contexts:

[Class Statement](#)

[Const Statement](#)

[Declare Statement](#)

[Delegate Statement](#)

[Dim Statement](#)

[Enum Statement](#)

[Event Statement](#)

[Function Statement](#)

[Interface Statement](#)

[Module Statement](#)

[Property Statement](#)

[Structure Statement](#)

[Sub Statement](#)

See also

- [InternalsVisibleToAttribute](#)
- [Public](#)
- [Protected](#)
- [Private](#)

- Private Protected
- Protected Friend
- Access levels in Visual Basic
- Procedures
- Structures
- Objects and Classes

In (Generic Modifier) (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

For generic type parameters, the `In` keyword specifies that the type parameter is contravariant.

Remarks

Contravariance enables you to use a less derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement variant interfaces and implicit conversion of delegate types.

For more information, see [Covariance and Contravariance](#).

Rules

You can use the `In` keyword in generic interfaces and delegates.

A type parameter can be declared contravariant in a generic interface or delegate if it is used only as a type of method arguments and not used as a method return type. `ByRef` parameters cannot be covariant or contravariant.

Covariance and contravariance are supported for reference types and not supported for value types.

In Visual Basic, you cannot declare events in contravariant interfaces without specifying the delegate type. Also, contravariant interfaces cannot have nested classes, enums, or structures, but they can have nested interfaces.

Behavior

An interface that has a contravariant type parameter allows its methods to accept arguments of less derived types than those specified by the interface type parameter. For example, because in .NET Framework 4, in the `IComparer<T>` interface, type `T` is contravariant, you can assign an object of the `IComparer(Of Person)` type to an object of the `IComparer(Of Employee)` type without using any special conversion methods if `Employee` inherits from `Person`.

A contravariant delegate can be assigned another delegate of the same type, but with a less derived generic type parameter.

Example - contravariant generic interface

The following example shows how to declare, extend, and implement a contravariant generic interface. It also shows how you can use implicit conversion for classes that implement this interface.

```

' Contravariant interface.
Interface IContravariant(Of In A)
End Interface

' Extending contravariant interface.
Interface IExtContravariant(Of In A)
    Inherits IContravariant(Of A)
End Interface

' Implementing contravariant interface.
Class Sample(Of A)
    Implements IContravariant(Of A)
End Class

Sub Main()
    Dim iobj As IContravariant(Of Object) = New Sample(Of Object)()
    Dim istr As IContravariant(Of String) = New Sample(Of String)()

    ' You can assign iobj to istr, because
    ' the IContravariant interface is contravariant.
    istr = iobj
End Sub

```

Example - contravariant generic delegate

The following example shows how to declare, instantiate, and invoke a contravariant generic delegate. It also shows how you can implicitly convert a delegate type.

```

' Contravariant delegate.
Public Delegate Sub DContravariant(Of In A)(ByVal argument As A)

' Methods that match the delegate signature.
Public Shared Sub SampleControl(ByVal control As Control)
End Sub

Public Shared Sub SampleButton(ByVal control As Button)
End Sub

Private Sub Test()

    ' Instantiating the delegates with the methods.
    Dim dControl As DContravariant(Of Control) =
        AddressOf SampleControl
    Dim dButton As DContravariant(Of Button) =
        AddressOf SampleButton

    ' You can assign dControl to dButton
    ' because the DContravariant delegate is contravariant.
    dButton = dControl

    ' Invoke the delegate.
    dButton(New Button())
End Sub

```

See also

- [Variance in Generic Interfaces](#)
- [Out](#)

Iterator (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

Specifies that a function or `Get` accessor is an iterator.

Remarks

An *iterator* performs a custom iteration over a collection. An iterator uses the `Yield` statement to return each element in the collection one at a time. When a `Yield` statement is reached, the current location in code is retained. Execution is restarted from that location the next time that the iterator function is called.

An iterator can be implemented as a function or as a `Get` accessor of a property definition. The `Iterator` modifier appears in the declaration of the iterator function or `Get` accessor.

You call an iterator from client code by using a [For Each...Next Statement](#).

The return type of an iterator function or `Get` accessor can be `IEnumerable`, `IEnumerable<T>`, `IEnumerator`, or `IEnumerator<T>`.

An iterator cannot have any `ByRef` parameters.

An iterator cannot occur in an event, instance constructor, static constructor, or static destructor.

An iterator can be an anonymous function. For more information, see [Iterators](#).

Usage

The `Iterator` modifier can be used in these contexts:

- [Function Statement](#)
- [Property Statement](#)

Example

The following example demonstrates an iterator function. The iterator function has a `yield` statement that is inside a `For...Next` loop. Each iteration of the `For Each` statement body in `Main` creates a call to the `Power` iterator function. Each call to the iterator function proceeds to the next execution of the `Yield` statement, which occurs during the next iteration of the `For...Next` loop.

```

Sub Main()
    For Each number In Power(2, 8)
        Console.WriteLine(number & " ")
    Next
    ' Output: 2 4 8 16 32 64 128 256
    Console.ReadKey()
End Sub

Private Iterator Function Power(
    ByVal base As Integer, ByVal highExponent As Integer) _
    As System.Collections.Generic.IEnumerable(Of Integer)

    Dim result = 1

    For counter = 1 To highExponent
        result = result * base
        Yield result
    Next
End Function

```

Example

The following example demonstrates a `Get` accessor that is an iterator. The `Iterator` modifier is in the property declaration.

```

Sub Main()
    Dim theGalaxies As New Galaxies
    For Each theGalaxy In theGalaxies.NextGalaxy
        With theGalaxy
            Console.WriteLine(.Name & " " & .MegaLightYears)
        End With
    Next
    Console.ReadKey()
End Sub

Public Class Galaxies
    Public ReadOnly Iterator Property NextGalaxy _
        As System.Collections.Generic.IEnumerable(Of Galaxy)
        Get
            Yield New Galaxy With {.Name = "Tadpole", .MegaLightYears = 400}
            Yield New Galaxy With {.Name = "Pinwheel", .MegaLightYears = 25}
            Yield New Galaxy With {.Name = "Milky Way", .MegaLightYears = 0}
            Yield New Galaxy With {.Name = "Andromeda", .MegaLightYears = 3}
        End Get
    End Property
End Class

Public Class Galaxy
    Public Property Name As String
    Public Property MegaLightYears As Integer
End Class

```

For additional examples, see [Iterators](#).

See also

- [IteratorStateMachineAttribute](#)
- [Iterators](#)
- [Yield Statement](#)

Key (Visual Basic)

4/2/2019 • 3 minutes to read • [Edit Online](#)

The `Key` keyword enables you to specify behavior for properties of anonymous types. Only properties you designate as key properties participate in tests of equality between anonymous type instances, or calculation of hash code values. The values of key properties cannot be changed.

You designate a property of an anonymous type as a key property by placing the keyword `Key` in front of its declaration in the initialization list. In the following example, `Airline` and `FlightNo` are key properties, but `Gate` is not.

```
Dim flight1 = New With {Key .Airline = "Blue Yonder Airlines",
                      Key .FlightNo = 3554, .Gate = "C33"}
```

When a new anonymous type is created, it inherits directly from `Object`. The compiler overrides three inherited members: `Equals`, `GetHashCode`, and `ToString`. The override code that is produced for `Equals` and `GetHashCode` is based on key properties. If there are no key properties in the type, `GetHashCode` and `Equals` are not overridden.

Equality

Two anonymous type instances are equal if they are instances of the same type and if the values of their key properties are equal. In the following examples, `flight2` is equal to `flight1` from the previous example because they are instances of the same anonymous type and they have matching values for their key properties. However, `flight3` is not equal to `flight1` because it has a different value for a key property, `FlightNo`. Instance `flight4` is not the same type as `flight1` because they designate different properties as key properties.

```
Dim flight2 = New With {Key .Airline = "Blue Yonder Airlines",
                      Key .FlightNo = 3554, .Gate = "D14"}
' The following statement displays True. The values of the non-key
' property, Gate, do not have to be equal.
Console.WriteLine(flight1.Equals(flight2))

Dim flight3 = New With {Key .Airline = "Blue Yonder Airlines",
                      Key .FlightNo = 431, .Gate = "C33"}
' The following statement displays False, because flight3 has a
' different value for key property FlightNo.
Console.WriteLine(flight1.Equals(flight3))

Dim flight4 = New With {Key .Airline = "Blue Yonder Airlines",
                      .FlightNo = 3554, .Gate = "C33"}
' The following statement displays False. Instance flight4 is not the
' same type as flight1 because they have different key properties.
' FlightNo is a key property of flight1 but not of flight4.
Console.WriteLine(flight1.Equals(flight4))
```

If two instances are declared with only non-key properties, identical in name, type, order, and value, the two instances are not equal. An instance without key properties is equal only to itself.

For more information about the conditions under which two anonymous type instances are instances of the same anonymous type, see [Anonymous Types](#).

Hash Code Calculation

Like [Equals](#), the hash function that is defined in [GetHashCode](#) for an anonymous type is based on the key properties of the type. The following examples show the interaction between key properties and hash code values.

Instances of an anonymous type that have the same values for all key properties have the same hash code value, even if non-key properties do not have matching values. The following statement returns `True`.

```
Console.WriteLine(flight1.GetHashCode = flight2.GetHashCode)
```

Instances of an anonymous type that have different values for one or more key properties have different hash code values. The following statement returns `False`.

```
Console.WriteLine(flight1.GetHashCode = flight3.GetHashCode)
```

Instances of anonymous types that designate different properties as key properties are not instances of the same type. They have different hash code values even when the names and values of all properties are the same. The following statement returns `False`.

```
Console.WriteLine(flight1.GetHashCode = flight4.GetHashCode)
```

Read-Only Values

The values of key properties cannot be changed. For example, in `flight1` in the earlier examples, the `Airline` and `FlightNo` fields are read-only, but `Gate` can be changed.

```
' The following statement will not compile, because FlightNo is a key
' property and cannot be changed.
' flight1.FlightNo = 1234
'
' Gate is not a key property. Its value can be changed.
flight1.Gate = "C5"
```

See also

- [Anonymous Type Definition](#)
- [How to: Infer Property Names and Types in Anonymous Type Declarations](#)
- [Anonymous Types](#)

Module <keyword> (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Specifies that an attribute at the beginning of a source file applies to the current assembly module.

Remarks

Many attributes pertain to an individual programming element, such as a class or property. You apply such an attribute by attaching the attribute block, within angle brackets (`<>`), directly to the declaration statement.

If an attribute pertains not only to the following element but to the current assembly module, you place the attribute block at the beginning of the source file and identify the attribute with the `Module` keyword. If it applies to the entire assembly, you use the `Assembly` keyword.

The `Module` modifier is not the same as the [Module Statement](#).

See also

- [Assembly](#)
- [Module Statement](#)
- [Attributes overview](#)

MustInherit (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

Specifies that a class can be used only as a base class and that you cannot create an object directly from it.

Remarks

The purpose of a *base class* (also known as an *abstract class*) is to define functionality that is common to all the classes derived from it. This saves the derived classes from having to redefine the common elements. In some cases, this common functionality is not complete enough to make a usable object, and each derived class defines the missing functionality. In such a case, you want the consuming code to create objects only from the derived classes. You use `MustInherit` on the base class to enforce this.

Another use of a `MustInherit` class is to restrict a variable to a set of related classes. You can define a base class and derive all these related classes from it. The base class does not need to provide any functionality common to all the derived classes, but it can serve as a filter for assigning values to variables. If your consuming code declares a variable as the base class, Visual Basic allows you to assign only an object from one of the derived classes to that variable.

The .NET Framework defines several `MustInherit` classes, among them `Array`, `Enum`, and `ValueType`. `ValueType` is an example of a base class that restricts a variable. All value types derive from `ValueType`. If you declare a variable as `ValueType`, you can assign only value types to that variable.

Rules

- **Declaration Context.** You can use `MustInherit` only in a `Class` statement.
- **Combined Modifiers.** You cannot specify `MustInherit` together with `NotInheritable` in the same declaration.

Example

The following example illustrates both forced inheritance and forced overriding. The base class `shape` defines a variable `acrossLine`. The classes `circle` and `square` derive from `shape`. They inherit the definition of `acrossLine`, but they must define the function `area` because that calculation is different for each kind of shape.

```

Public MustInherit Class shape
    Public acrossLine As Double
    Public MustOverride Function area() As Double
End Class
Public Class circle : Inherits shape
    Public Overrides Function area() As Double
        Return Math.PI * acrossLine
    End Function
End Class
Public Class square : Inherits shape
    Public Overrides Function area() As Double
        Return acrossLine * acrossLine
    End Function
End Class
Public Class consumeShapes
    Public Sub makeShapes()
        Dim shape1, shape2 As shape
        shape1 = New circle
        shape2 = New square
    End Sub
End Class

```

You can declare `shape1` and `shape2` to be of type `shape`. However, you cannot create an object from `shape` because it lacks the functionality of the function `area` and is marked `MustInherit`.

Because they are declared as `shape`, the variables `shape1` and `shape2` are restricted to objects from the derived classes `circle` and `square`. Visual Basic does not allow you to assign any other object to these variables, which gives you a high level of type safety.

Usage

The `MustInherit` modifier can be used in this context:

Class Statement

See also

- [Inherits Statement](#)
- [NotInheritable](#)
- [Keywords](#)
- [Inheritance Basics](#)

MustOverride (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

Specifies that a property or procedure is not implemented in this class and must be overridden in a derived class before it can be used.

Remarks

You can use `MustOverride` only in a property or procedure declaration statement. The property or procedure that specifies `MustOverride` must be a member of a class, and the class must be marked [MustInherit](#).

Rules

- **Incomplete Declaration.** When you specify `MustOverride`, you do not supply any additional lines of code for the property or procedure, not even the `End Function`, `End Property`, or `End Sub` statement.
- **Combined Modifiers.** You cannot specify `MustOverride` together with `NotOverridable`, `Overridable`, or `Shared` in the same declaration.
- **Shadowing and Overriding.** Both shadowing and overriding redefine an inherited element, but there are significant differences between the two approaches. For more information, see [Shadowing in Visual Basic](#).
- **Alternate Terms.** An element that cannot be used except in an override is sometimes called a *pure virtual* element.

The `MustOverride` modifier can be used in these contexts:

[Function Statement](#)

[Property Statement](#)

[Sub Statement](#)

See also

- [NotOverridable](#)
- [Overridable](#)
- [Overrides](#)
- [MustInherit](#)
- [Keywords](#)
- [Shadowing in Visual Basic](#)

Narrowing (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Indicates that a conversion operator (`cType`) converts a class or structure to a type that might not be able to hold some of the possible values of the original class or structure.

Converting with the Narrowing Keyword

The conversion procedure must specify `Public Shared` in addition to `Narrowing`.

Narrowing conversions do not always succeed at run time, and can fail or incur data loss. Examples are `Long` to `Integer`, `String` to `Date`, and a base type to a derived type. This last conversion is narrowing because the base type might not contain all the members of the derived type and thus is not an instance of the derived type.

If `Option Strict` is `On`, the consuming code must use `cType` for all narrowing conversions.

The `Narrowing` keyword can be used in this context:

[Operator Statement](#)

See also

- [Operator Statement](#)
- [Widening](#)
- [Widening and Narrowing Conversions](#)
- [How to: Define an Operator](#)
- [CType Function](#)
- [Option Strict Statement](#)

NotInheritable (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Specifies that a class cannot be used as a base class.

Remarks

Alternate Terms. A class that cannot be inherited is sometimes called a *sealed* class.

The `NotInheritable` modifier can be used in this context:

[Class Statement](#)

See also

- [Inherits Statement](#)
- [MustInherit](#)
- [Keywords](#)

NotOverridable (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Specifies that a property or procedure cannot be overridden in a derived class.

Remarks

The `NotOverridable` modifier prevents a property or method from being overridden in a derived class. The `Overridable` modifier allows a property or method in a class to be overridden in a derived class. For more information, see [Inheritance Basics](#).

If the `Overridable` or `NotOverridable` modifier is not specified, the default setting depends on whether the property or method overrides a base class property or method. If the property or method overrides a base class property or method, the default setting is `Overridable`; otherwise, it is `NotOverridable`.

An element that cannot be overridden is sometimes called a *sealed* element.

You can use `NotOverridable` only in a property or procedure declaration statement. You can specify `NotOverridable` only on a property or procedure that overrides another property or procedure, that is, only in combination with `Overrides`.

Combined Modifiers

You cannot specify `Overridable` or `NotOverridable` for a `Private` method.

You cannot specify `NotOverridable` together with `MustOverride`, `Overridable`, or `Shared` in the same declaration.

Usage

The `NotOverridable` modifier can be used in these contexts:

[Function Statement](#)

[Property Statement](#)

[Sub Statement](#)

See also

- [Modifiers](#)
- [Inheritance Basics](#)
- [MustOverride](#)
- [Overridable](#)
- [Overrides](#)
- [Keywords](#)
- [Shadowing in Visual Basic](#)

Optional (Visual Basic)

7/30/2019 • 2 minutes to read • [Edit Online](#)

Specifies that a procedure argument can be omitted when the procedure is called.

Remarks

For each optional parameter, you must specify a constant expression as the default value of that parameter. If the expression evaluates to [Nothing](#), the default value of the value data type is used as the default value of the parameter.

If the parameter list contains an optional parameter, every parameter that follows it must also be optional.

The `Optional` modifier can be used in these contexts:

- [Declare Statement](#)
- [Function Statement](#)
- [Property Statement](#)
- [Sub Statement](#)

NOTE

When calling a procedure with or without optional parameters, you can pass arguments by position or by name. For more information, see [Passing Arguments by Position and by Name](#).

NOTE

You can also define a procedure with optional parameters by using overloading. If you have one optional parameter, you can define two overloaded versions of the procedure, one that accepts the parameter and one that doesn't. For more information, see [Procedure Overloading](#).

Example

The following example defines a procedure that has an optional parameter.

```

Public Function FindMatches(ByRef values As List(Of String),
                           ByVal searchString As String,
                           Optional ByVal matchCase As Boolean = False) As List(Of String)

    Dim results As IEnumerable(Of String)

    If matchCase Then
        results = From v In values
                  Where v.Contains(searchString)
    Else
        results = From v In values
                  Where UCASE(v).Contains(UCASE(searchString))
    End If

    Return results.ToList()
End Function

```

Example

The following example demonstrates how to call a procedure with arguments passed by position and with arguments passed by name. The procedure has two optional parameters.

```

Private Sub TestParameters()
    ' Call the procedure with its arguments passed by position,
    studentInfo("Mary", 19, #9/21/1981#)

    ' Omit one optional argument by holding its place with a comma.
    studentInfo("Mary", , #9/21/1981#)

    ' Call the procedure with its arguments passed by name.
    studentInfo(age:=19, birth:=#9/21/1981#, name:="Mary")

    ' Supply an argument by position and an argument by name.
    studentInfo("Mary", birth:=#9/21/1981#)
End Sub

Private Sub studentInfo(ByVal name As String,
                       Optional ByVal age As Short = 0,
                       Optional ByVal birth As Date = #1/1/2000#)

    Console.WriteLine("name: " & name)
    Console.WriteLine("age: " & age)
    Console.WriteLine("birth date: " & birth)
    Console.WriteLine()
End Sub

```

See also

- [Parameter List](#)
- [Optional Parameters](#)
- [Keywords](#)

Out (Generic Modifier) (Visual Basic)

3/5/2019 • 2 minutes to read • [Edit Online](#)

For generic type parameters, the `out` keyword specifies that the type is covariant.

Remarks

Covariance enables you to use a more derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement variant interfaces and implicit conversion of delegate types.

For more information, see [Covariance and Contravariance](#).

Rules

You can use the `out` keyword in generic interfaces and delegates.

In a generic interface, a type parameter can be declared covariant if it satisfies the following conditions:

- The type parameter is used only as a return type of interface methods and not used as a type of method arguments.

NOTE

There is one exception to this rule. If in a covariant interface you have a contravariant generic delegate as a method parameter, you can use the covariant type as a generic type parameter for this delegate. For more information about covariant and contravariant generic delegates, see [Variance in Delegates](#) and [Using Variance for Func and Action Generic Delegates](#).

- The type parameter is not used as a generic constraint for the interface methods.

In a generic delegate, a type parameter can be declared covariant if it is used only as a method return type and not used for method arguments.

Covariance and contravariance are supported for reference types, but they are not supported for value types.

In Visual Basic, you cannot declare events in covariant interfaces without specifying the delegate type. Also, covariant interfaces cannot have nested classes, enums, or structures, but they can have nested interfaces.

Behavior

An interface that has a covariant type parameter enables its methods to return more derived types than those specified by the type parameter. For example, because in .NET Framework 4, in `IEnumerable<T>`, type T is covariant, you can assign an object of the `IEnumerable(Of String)` type to an object of the `IEnumerable(Of Object)` type without using any special conversion methods.

A covariant delegate can be assigned another delegate of the same type, but with a more derived generic type parameter.

Example

The following example shows how to declare, extend, and implement a covariant generic interface. It also shows how to use implicit conversion for classes that implement a covariant interface.

```

' Covariant interface.
Interface ICovariant(Of Out R)
End Interface

' Extending covariant interface.
Interface IExtCovariant(Of Out R)
    Inherits ICovariant(Of R)
End Interface

' Implementing covariant interface.
Class Sample(Of R)
    Implements ICovariant(Of R)
End Class

Sub Main()
    Dim iobj As ICovariant(Of Object) = New Sample(Of Object)()
    Dim istr As ICovariant(Of String) = New Sample(Of String)()

    ' You can assign istr to iobj because
    ' the ICovariant interface is covariant.
    iobj = istr
End Sub

```

Example

The following example shows how to declare, instantiate, and invoke a covariant generic delegate. It also shows how you can use implicit conversion for delegate types.

```

' Covariant delegate.
Public Delegate Function DCovariant(Of Out R)() As R

' Methods that match the delegate signature.
Public Shared Function SampleControl() As Control
    Return New Control()
End Function

Public Shared Function SampleButton() As Button
    Return New Button()
End Function

Private Sub Test()

    ' Instantiating the delegates with the methods.
    Dim dControl As DCovariant(Of Control) =
        AddressOf SampleControl
    Dim dButton As DCovariant(Of Button) =
        AddressOf SampleButton

    ' You can assign dButton to dControl
    ' because the DCovariant delegate is covariant.
    dControl = dButton

    ' Invoke the delegate.
    dControl()
End Sub

```

See also

- [Variance in Generic Interfaces](#)
- [In](#)

Overloads (Visual Basic)

7/9/2019 • 2 minutes to read • [Edit Online](#)

Specifies that a property or procedure redeclares one or more existing properties or procedures with the same name.

Remarks

Overloading is the practice of supplying more than one definition for a given property or procedure name in the same scope. Redeclaring a property or procedure with a different signature is sometimes called *hiding by signature*.

Rules

- **Declaration Context.** You can use `Overloads` only in a property or procedure declaration statement.
- **Combined Modifiers.** You cannot specify `Overloads` together with `Shadows` in the same procedure declaration.
- **Required Differences.** The *signature* in this declaration must be different from the signature of every property or procedure that it overloads. The signature comprises the property or procedure name together with the following:
 - the number of parameters
 - the order of the parameters
 - the data types of the parameters
 - the number of type parameters (for a generic procedure)
 - the return type (only for a conversion operator procedure)
- All overloads must have the same name, but each must differ from all the others in one or more of the preceding respects. This allows the compiler to distinguish which version to use when code calls the property or procedure.
- **Disallowed Differences.** Changing one or more of the following is not valid for overloading a property or procedure, because they are not part of the signature:
 - whether or not it returns a value (for a procedure)
 - the data type of the return value (except for a conversion operator)
 - the names of the parameters or type parameters
 - the constraints on the type parameters (for a generic procedure)
 - parameter modifier keywords (such as `ByRef` or `Optional`)
 - property or procedure modifier keywords (such as `Public` or `Shared`)
- **Optional Modifier.** You do not have to use the `Overloads` modifier when you are defining multiple overloaded properties or procedures in the same class. However, if you use `Overloads` in one of the declarations, you must use it in all of them.

- **Shadowing and Overloading.** `Overloads` can also be used to shadow an existing member, or set of overloaded members, in a base class. When you use `Overloads` in this way, you declare the property or method with the same name and the same parameter list as the base class member, and you do not supply the `Shadows` keyword.

If you use `Overrides`, the compiler implicitly adds `overloads` so that your library APIs work with C# more easily.

The `Overloads` modifier can be used in these contexts:

- [Function Statement](#)
- [Operator Statement](#)
- [Property Statement](#)
- [Sub Statement](#)

See also

- [Shadows](#)
- [Procedure Overloading](#)
- [Generic Types in Visual Basic](#)
- [Operator Procedures](#)
- [How to: Define a Conversion Operator](#)

Overridable (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Specifies that a property or procedure can be overridden by an identically named property or procedure in a derived class.

Remarks

The `Overridable` modifier allows a property or method in a class to be overridden in a derived class. The `NotOverridable` modifier prevents a property or method from being overridden in a derived class. For more information, see [Inheritance Basics](#).

If the `Overridable` or `NotOverridable` modifier is not specified, the default setting depends on whether the property or method overrides a base class property or method. If the property or method overrides a base class property or method, the default setting is `Overridable`; otherwise, it is `NotOverridable`.

You can shadow or override to redefine an inherited element, but there are significant differences between the two approaches. For more information, see [Shadowing in Visual Basic](#).

An element that can be overridden is sometimes referred to as a *virtual* element. If it can be overridden, but does not have to be, it is sometimes also called a *concrete* element.

You can use `Overridable` only in a property or procedure declaration statement.

Combined Modifiers

You cannot specify `Overridable` or `NotOverridable` for a `Private` method.

You cannot specify `Overridable` together with `MustOverride`, `NotOverridable`, or `Shared` in the same declaration.

Because an overriding element is implicitly overridable, you cannot combine `Overridable` with `Overrides`.

Usage

The `Overridable` modifier can be used in these contexts:

[Function Statement](#)

[Property Statement](#)

[Sub Statement](#)

See also

- [Modifiers](#)
- [Inheritance Basics](#)
- [MustOverride](#)
- [NotOverridable](#)
- [Overrides](#)
- [Keywords](#)
- [Shadowing in Visual Basic](#)

Overrides (Visual Basic)

4/26/2019 • 2 minutes to read • [Edit Online](#)

Specifies that a property or procedure overrides an identically named property or procedure inherited from a base class.

Rules

- **Declaration Context.** You can use `Overrides` only in a property or procedure declaration statement.
- **Combined Modifiers.** You cannot specify `Overrides` together with `Shadows` or `Shared` in the same declaration. Because an overriding element is implicitly overridable, you cannot combine `Overridable` with `Overrides`.
- **Matching Signatures.** The signature of this declaration must exactly match the *signature* of the property or procedure that it overrides. This means the parameter lists must have the same number of parameters, in the same order, with the same data types.

In addition to the signature, the overriding declaration must also exactly match the following:

- The access level
- The return type, if any
- **Generic Signatures.** For a generic procedure, the signature includes the number of type parameters. Therefore, the overriding declaration must match the base class version in that respect as well.
- **Additional Matching.** In addition to matching the signature of the base class version, this declaration must also match it in the following respects:
 - Access-level modifier (such as `Public`)
 - Passing mechanism of each parameter (`ByVal` or `ByRef`)
 - Constraint lists on each type parameter of a generic procedure
- **Shadowing and Overriding.** Both shadowing and overriding redefine an inherited element, but there are significant differences between the two approaches. For more information, see [Shadowing in Visual Basic](#).

If you use `Overrides`, the compiler implicitly adds `Overloads` so that your library APIs work with C# more easily.

The `Overrides` modifier can be used in these contexts:

- [Function Statement](#)
- [Property Statement](#)
- [Sub Statement](#)

See also

- [MustOverride](#)
- [NotOverridable](#)

- [Overridable](#)
- [Keywords](#)
- [Shadowing in Visual Basic](#)
- [Generic Types in Visual Basic](#)
- [Type List](#)

ParamArray (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

Specifies that a procedure parameter takes an optional array of elements of the specified type. `ParamArray` can be used only on the last parameter of a parameter list.

Remarks

`ParamArray` allows you to pass an arbitrary number of arguments to the procedure. A `ParamArray` parameter is always declared using [ByVal](#).

You can supply one or more arguments to a `ParamArray` parameter by passing an array of the appropriate data type, a comma-separated list of values, or nothing at all. For details, see "Calling a ParamArray" in [Parameter Arrays](#).

IMPORTANT

Whenever you deal with an array which can be indefinitely large, there is a risk of overrunning some internal capacity of your application. If you accept a parameter array from the calling code, you should test its length and take appropriate steps if it is too large for your application.

The `ParamArray` modifier can be used in these contexts:

[Declare Statement](#)

[Function Statement](#)

[Property Statement](#)

[Sub Statement](#)

See also

- [Keywords](#)
- [Parameter Arrays](#)

Partial (Visual Basic)

9/28/2019 • 3 minutes to read • [Edit Online](#)

Indicates that a type declaration is a partial definition of the type.

You can divide the definition of a type among several declarations by using the `Partial` keyword. You can use as many partial declarations as you want, in as many different source files as you want. However, all the declarations must be in the same assembly and the same namespace.

NOTE

Visual Basic supports *partial methods*, which are typically implemented in partial classes. For more information, see [Partial Methods](#) and [Sub Statement](#).

Syntax

```
[ <attrlist> ] [ accessmodifier ] [ Shadows ] [ MustInherit | NotInheritable ] _  
Partial { Class | Structure | Interface | Module } name [ (Of typelist) ]  
    [ Inherits classname ]  
    [ Implements interfacenames ]  
    [ variabledeclarations ]  
    [ proceduredeclarations ]  
{ End Class | End Structure }
```

Parts

TERM	DEFINITION
<code>attrlist</code>	Optional. List of attributes that apply to this type. You must enclose the Attribute List in angle brackets (<code><></code>).
<code>accessmodifier</code>	Optional. Specifies what code can access this type. See Access levels in Visual Basic .
<code>Shadows</code>	Optional. See Shadows .
<code>MustInherit</code>	Optional. See MustInherit .
<code>NotInheritable</code>	Optional. See NotInheritable .
<code>name</code>	Required. Name of this type. Must match the name defined in all other partial declarations of the same type.
<code>of</code>	Optional. Specifies that this is a generic type. See Generic Types in Visual Basic .
<code>typelist</code>	Required if you use <code>Of</code> . See Type List .
<code>Inherits</code>	Optional. See Inherits Statement .

TERM	DEFINITION
<code>classname</code>	Required if you use <code>Inherits</code> . The name of the class or interface from which this class derives.
<code>Implements</code>	Optional. See Implements Statement .
<code>interfacenames</code>	Required if you use <code>Implements</code> . The names of the interfaces this type implements.
<code>variabledeclarations</code>	Optional. Statements which declare additional variables and events for the type.
<code>proceduredeclarations</code>	Optional. Statements which declare and define additional procedures for the type.
<code>End Class</code> or <code>End Structure</code>	Ends this partial <code>Class</code> or <code>Structure</code> definition.

Remarks

Visual Basic uses partial-class definitions to separate generated code from user-authored code in separate source files. For example, the **Windows Form Designer** defines partial classes for controls such as `Form`. You should not modify the generated code in these controls.

All the rules for class, structure, interface, and module creation, such as those for modifier usage and inheritance, apply when creating a partial type.

Best Practices

- Under normal circumstances, you should not split the development of a single type across two or more declarations. Therefore, in most cases you do not need the `Partial` keyword.
- For readability, every partial declaration of a type should include the `Partial` keyword. The compiler allows at most one partial declaration to omit the keyword; if two or more omit it the compiler signals an error.

Behavior

- Union of Declarations.** The compiler treats the type as the union of all its partial declarations. Every modifier from every partial definition applies to the entire type, and every member from every partial definition is available to the entire type.
- Type Promotion Not Allowed For Partial Types in Modules.** If a partial definition is inside a module, type promotion of that type is automatically defeated. In such a case, a set of partial definitions can cause unexpected results and even compiler errors. For more information, see [Type Promotion](#).

The compiler merges partial definitions only when their fully qualified paths are identical.

The `Partial` keyword can be used in these contexts:

[Class Statement](#)

[Structure Statement](#)

Example

The following example splits the definition of class `sampleClass` into two declarations, each of which defines a different `Sub` procedure.

```
Partial Public Class sampleClass
    Public Sub sub1()
    End Sub
End Class
Partial Public Class sampleClass
    Public Sub sub2()
    End Sub
End Class
```

The two partial definitions in the preceding example could be in the same source file or in two different source files.

See also

- [Class Statement](#)
- [Structure Statement](#)
- [Type Promotion](#)
- [Shadows](#)
- [Generic Types in Visual Basic](#)
- [Partial Methods](#)

Private (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

Specifies that one or more declared programming elements are accessible only from within their declaration context, including from within any contained types.

Remarks

If a programming element represents proprietary functionality, or contains confidential data, you usually want to limit access to it as strictly as possible. You achieve the maximum limitation by allowing only the module, class, or structure that defines it to access it. To limit access to an element in this way, you can declare it with `Private`.

NOTE

You can also use the [Private Protected](#) access modifier, which makes a member accessible from within that class and from derived classes located in its containing assembly.

Rules

- **Declaration Context.** You can use `Private` only at module level. This means the declaration context for a `Private` element must be a module, class, or structure, and cannot be a source file, namespace, interface, or procedure.

Behavior

- **Access Level.** All code within a declaration context can access its `Private` elements. This includes code within a contained type, such as a nested class or an assignment expression in an enumeration. No code outside of the declaration context can access its `Private` elements.
- **Access Modifiers.** The keywords that specify access level are called *access modifiers*. For a comparison of the access modifiers, see [Access levels in Visual Basic](#).

The `Private` modifier can be used in these contexts:

[Class Statement](#)

[Const Statement](#)

[Declare Statement](#)

[Delegate Statement](#)

[Dim Statement](#)

[Enum Statement](#)

[Event Statement](#)

[Function Statement](#)

[Interface Statement](#)

[Property Statement](#)

[Structure Statement](#)

[Sub Statement](#)

See also

- [Public](#)
- [Protected](#)
- [Friend](#)
- [Private Protected](#)
- [Protected Friend Access levels in Visual Basic](#)
- [Procedures](#)
- [Structures](#)
- [Objects and Classes](#)

Private Protected (Visual Basic)

3/5/2019 • 2 minutes to read • [Edit Online](#)

The `Private Protected` keyword combination is a member access modifier. A `Private Protected` member is accessible by all members in its containing class, as well as by types derived from the containing class, but only if they are found in its containing assembly.

You can specify `Private Protected` only on members of classes; you cannot apply `Private Protected` to members of a structure because structures cannot be inherited.

The `Private Protected` access modifier is supported by Visual Basic 15.5 and later. To use it, you can add the following element to your Visual Basic project (*.vbproj) file. As long as Visual Basic 15.5 or later is installed on your system, it lets you take advantage of all the language features supported by the latest version of the Visual Basic compiler:

```
<PropertyGroup>
    <LangVersion>latest</LangVersion>
</PropertyGroup>
```

For more information see [setting the Visual Basic language version](#).

NOTE

In Visual Studio, selecting F1 help on `private protected` provides help for either `private` or `protected`. The IDE picks the single token under the cursor rather than the compound word.

Rules

- **Declaration Context.** You can use `Private Protected` only at the class level. This means the declaration context for a `Protected` element must be a class, and cannot be a source file, namespace, interface, module, structure, or procedure.

Behavior

- **Access Level.** All code in a class can access its elements. Code in any class that derives from a base class and is contained in the same assembly can access all the `Private Protected` elements of the base class. However, code in any class that derives from a base class and is contained in a different assembly can't access the base class `Private Protected` elements.
- **Access Modifiers.** The keywords that specify access level are called *access modifiers*. For a comparison of the access modifiers, see [Access levels in Visual Basic](#).

The `Private Protected` modifier can be used in these contexts:

- [Class Statement](#) of a nested class
- [Const Statement](#)
- [Declare Statement](#)
- [Delegate Statement](#) of a delegate nested in a class

- [Dim Statement](#)
- [Enum Statement](#) of an enumeration nested in a class
- [Event Statement](#)
- [Function Statement](#)
- [Interface Statement](#) of an interface nested in a class
- [Property Statement](#)
- [Structure Statement](#) of a structure nested in a class
- [Sub Statement](#)

See also

- [Public](#)
- [Protected](#)
- [Friend](#)
- [Private](#)
- [Protected Friend](#)
- [Access levels in Visual Basic](#)
- [Procedures](#)
- [Structures](#)
- [Objects and Classes](#)

Protected (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

A member access modifier that specifies that one or more declared programming elements are accessible only from within their own class or from a derived class.

Remarks

Sometimes a programming element declared in a class contains sensitive data or restricted code, and you want to limit access to the element. However, if the class is inheritable and you expect a hierarchy of derived classes, it might be necessary for these derived classes to access the data or code. In such a case, you want the element to be accessible both from the base class and from all derived classes. To limit access to an element in this manner, you can declare it with `Protected`.

NOTE

The `Protected` access modifier can be combined with two other modifiers:

- The `Protected Friend` modifier makes a class member accessible from within that class, from derived classes, and from the same assembly in which the class is defined.
- The `Private Protected` modifier makes a class member accessible by derived types, but only within its containing assembly.

Rules

Declaration Context. You can use `Protected` only at the class level. This means the declaration context for a `Protected` element must be a class, and cannot be a source file, namespace, interface, module, structure, or procedure.

Behavior

- **Access Level.** All code in a class can access its elements. Code in any class that derives from a base class can access all the `Protected` elements of the base class. This is true for all generations of derivation. This means that a class can access `Protected` elements of the base class of the base class, and so on.

Protected access is not a superset or subset of friend access.

- **Access Modifiers.** The keywords that specify access level are called *access modifiers*. For a comparison of the access modifiers, see [Access levels in Visual Basic](#).

The `Protected` modifier can be used in these contexts:

- [Class Statement](#)
- [Const Statement](#)
- [Declare Statement](#)
- [Delegate Statement](#)
- [Dim Statement](#)

- [Enum Statement](#)
- [Event Statement](#)
- [Function Statement](#)
- [Interface Statement](#)
- [Property Statement](#)
- [Structure Statement](#)
- [Sub Statement](#)

See also

- [Public](#)
- [Friend](#)
- [Private](#)
- [Private Protected](#)
- [Protected Friend](#)
- [Access levels in Visual Basic](#)
- [Procedures](#)
- [Structures](#)
- [Objects and Classes](#)

Protected Friend (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

The `Protected Friend` keyword combination is a member access modifier. It confers both [Friend](#) access and [Protected](#) access on the declared elements, so they are accessible from anywhere in the same assembly, from their own class, and from derived classes. You can specify `Protected Friend` only on members of classes; you cannot apply `Protected Friend` to members of a structure because structures cannot be inherited.

NOTE

In Visual Studio, selecting F1 help on `protected friend` provides help for either [protected](#) or [friend](#). The IDE picks the single token under the cursor rather than the compound word.

Rules

Declaration Context. You can use `Protected Friend` only at the class level. This means the declaration context for a `Protected` element must be a class, and cannot be a source file, namespace, interface, module, structure, or procedure.

See also

- [Public](#)
- [Protected](#)
- [Friend](#)
- [Private](#)
- [Private Protected](#)
- [Access levels in Visual Basic](#)
- [Procedures](#)
- [Structures](#)
- [Objects and Classes](#)

Public (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

Specifies that one or more declared programming elements have no access restrictions.

Remarks

If you are publishing a component or set of components, such as a class library, you usually want the programming elements to be accessible by any code that interoperates with your assembly. To confer such unlimited access on an element, you can declare it with `Public`.

Public access is the normal level for a programming element when you do not need to limit access to it. Note that the access level of an element declared within an interface, module, class, or structure defaults to `Public` if you do not declare it otherwise.

Rules

- **Declaration Context.** You can use `Public` only at module, interface, or namespace level. This means the declaration context for a `Public` element must be a source file, namespace, interface, module, class, or structure, and cannot be a procedure.

Behavior

- **Access Level.** All code that can access a module, class, or structure can access its `Public` elements.
- **Default Access.** Local variables inside a procedure default to public access, and you cannot use any access modifiers on them.
- **Access Modifiers.** The keywords that specify access level are called *access modifiers*. For a comparison of the access modifiers, see [Access levels in Visual Basic](#).

The `Public` modifier can be used in these contexts:

[Class Statement](#)

[Const Statement](#)

[Declare Statement](#)

[Delegate Statement](#)

[Dim Statement](#)

[Enum Statement](#)

[Event Statement](#)

[Function Statement](#)

[Interface Statement](#)

[Module Statement](#)

[Operator Statement](#)

[Property Statement](#)

[Structure Statement](#)

[Sub Statement](#)

See also

- [Protected](#)
- [Friend](#)
- [Private](#)
- [Private Protected](#)
- [Protected Friend](#)
- [Access levels in Visual Basic](#)
- [Procedures](#)
- [Structures](#)
- [Objects and Classes](#)

ReadOnly (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Specifies that a variable or property can be read but not written.

Remarks

Rules

- **Declaration Context.** You can use `ReadOnly` only at module level. This means the declaration context for a `ReadOnly` element must be a class, structure, or module, and cannot be a source file, namespace, or procedure.
- **Combined Modifiers.** You cannot specify `ReadOnly` together with `static` in the same declaration.
- **Assigning a Value.** Code consuming a `ReadOnly` property cannot set its value. But code that has access to the underlying storage can assign or change the value at any time.

You can assign a value to a `ReadOnly` variable only in its declaration or in the constructor of a class or structure in which it is defined.

When to Use a ReadOnly Variable

There are situations in which you cannot use a [Const Statement](#) to declare and assign a constant value. For example, the `Const` statement might not accept the data type you want to assign, or you might not be able to compute the value at compile time with a constant expression. You might not even know the value at compile time. In these cases, you can use a `ReadOnly` variable to hold a constant value.

IMPORTANT

If the data type of the variable is a reference type, such as an array or a class instance, its members can be changed even if the variable itself is `ReadOnly`. The following example illustrates this.

```
ReadOnly characterArray() As Char = {"x"c, "y"c, "z"c}
Sub ChangeArrayElement()
    characterArray(1) = "M"c
End Sub
```

When initialized, the array pointed to by `characterArray()` holds "x", "y", and "z". Because the variable `characterArray` is `ReadOnly`, you cannot change its value once it is initialized; that is, you cannot assign a new array to it. However, you can change the values of one or more of the array members. Following a call to the procedure `ChangeArrayElement`, the array pointed to by `characterArray()` holds "x", "M", and "z".

Note that this is similar to declaring a procedure parameter to be [ByVal](#), which prevents the procedure from changing the calling argument itself but allows it to change its members.

Example

The following example defines a `ReadOnly` property for the date on which an employee was hired. The class stores the property value internally as a `Private` variable, and only code inside the class can change that value.

However, the property is `Public`, and any code that can access the class can read the property.

```
Class employee
    ' Only code inside class employee can change the value of hireDateValue.
    Private hireDateValue As Date
    ' Any code that can access class employee can read property dateHired.
    Public ReadOnly Property dateHired() As Date
        Get
            Return hireDateValue
        End Get
    End Property
End Class
```

The `ReadOnly` modifier can be used in these contexts:

- [Dim Statement](#)
- [Property Statement](#)

See also

- [WriteOnly](#)
- [Keywords](#)

Shadows (Visual Basic)

7/9/2019 • 2 minutes to read • [Edit Online](#)

Specifies that a declared programming element redeclares and hides an identically named element, or set of overloaded elements, in a base class.

Remarks

The main purpose of shadowing (which is also known as *hiding by name*) is to preserve the definition of your class members. The base class might undergo a change that creates an element with the same name as one you have already defined. If this happens, the `Shadows` modifier forces references through your class to be resolved to the member you defined, instead of to the new base class element.

Both shadowing and overriding redefine an inherited element, but there are significant differences between the two approaches. For more information, see [Shadowing in Visual Basic](#).

Rules

- **Declaration Context.** You can use `Shadows` only at class level. This means the declaration context for a `Shadows` element must be a class, and cannot be a source file, namespace, interface, module, structure, or procedure.

You can declare only one shadowing element in a single declaration statement.

- **Combined Modifiers.** You cannot specify `Shadows` together with `Overloads`, `Overrides`, or `Static` in the same declaration.
- **Element Types.** You can shadow any kind of declared element with any other kind. If you shadow a property or procedure with another property or procedure, the parameters and the return type do not have to match those in the base class property or procedure.
- **Accessing.** The shadowed element in the base class is normally unavailable from within the derived class that shadows it. However, the following considerations apply.
 - If the shadowing element is not accessible from the code referring to it, the reference is resolved to the shadowed element. For example, if a `Private` element shadows a base class element, code that does not have permission to access the `Private` element accesses the base class element instead.
 - If you shadow an element, you can still access the shadowed element through an object declared with the type of the base class. You can also access it through `MyBase`.

The `Shadows` modifier can be used in these contexts:

- [Class Statement](#)
- [Const Statement](#)
- [Declare Statement](#)
- [Delegate Statement](#)
- [Dim Statement](#)
- [Enum Statement](#)

- [Event Statement](#)
- [Function Statement](#)
- [Interface Statement](#)
- [Property Statement](#)
- [Structure Statement](#)
- [Sub Statement](#)

See also

- [Shared](#)
- [Static](#)
- [Private](#)
- [Me, My, MyBase, and MyClass](#)
- [Inheritance Basics](#)
- [MustOverride](#)
- [NotOverridable](#)
- [Overloads](#)
- [Overridable](#)
- [Overrides](#)
- [Shadowing in Visual Basic](#)

Shared (Visual Basic)

4/28/2019 • 3 minutes to read • [Edit Online](#)

Specifies that one or more declared programming elements are associated with a class or structure at large, and not with a specific instance of the class or structure.

Remarks

When to Use Shared

Sharing a member of a class or structure makes it available to every instance, rather than *nonshared*, where each instance keeps its own copy. This is useful, for example, if the value of a variable applies to the entire application. If you declare that variable to be `Shared`, then all instances access the same storage location, and if one instance changes the variable's value, all instances access the updated value.

Sharing does not alter the access level of a member. For example, a class member can be shared and private (accessible only from within the class), or nonshared and public. For more information, see [Access levels in Visual Basic](#).

Rules

- **Declaration Context.** You can use `Shared` only at module level. This means the declaration context for a `Shared` element must be a class or structure, and cannot be a source file, namespace, or procedure.
- **Combined Modifiers.** You cannot specify `Shared` together with [Overrides](#), [Overridable](#), [NotOverridable](#), [MustOverride](#), or [Static](#) in the same declaration.
- **Accessing.** You access a shared element by qualifying it with its class or structure name, not with the variable name of a specific instance of its class or structure. You do not even have to create an instance of a class or structure to access its shared members.

The following example calls the shared procedure `IsNaN` exposed by the `Double` structure.

```
If Double.NaN(result) Then MsgBox("Result is mathematically undefined.")
```

- **Implicit Sharing.** You cannot use the `Shared` modifier in a [Const Statement](#), but constants are implicitly shared. Similarly, you cannot declare a member of a module or an interface to be `Shared`, but they are implicitly shared.

Behavior

- **Storage.** A shared variable or event is stored in memory only once, no matter how many or few instances you create of its class or structure. Similarly, a shared procedure or property holds only one set of local variables.
- **Accessing through an Instance Variable.** It is possible to access a shared element by qualifying it with the name of a variable that contains a specific instance of its class or structure. Although this usually works as expected, the compiler generates a warning message and makes the access through the class or structure name instead of the variable.
- **Accessing through an Instance Expression.** If you access a shared element through an expression that returns an instance of its class or structure, the compiler makes the access through the class or

structure name instead of evaluating the expression. This produces unexpected results if you intended the expression to perform other actions as well as returning the instance. The following example illustrates this.

```
Sub main()
    shareTotal.total = 10
    ' The preceding line is the preferred way to access total.
    Dim instanceVar As New shareTotal
    instanceVar.total += 100
    ' The preceding line generates a compiler warning message and
    ' accesses total through class shareTotal instead of through
    ' the variable instanceVar. This works as expected and adds
    ' 100 to total.
    returnClass().total += 1000
    ' The preceding line generates a compiler warning message and
    ' accesses total through class shareTotal instead of calling
    ' returnClass(). This adds 1000 to total but does not work as
    ' expected, because the MsgBox in returnClass() does not run.
    MsgBox("Value of total is " & CStr(shareTotal.total))
End Sub
Public Function returnClass() As shareTotal
    MsgBox("Function returnClass() called")
    Return New shareTotal
End Function
Public Class shareTotal
    Public Shared total As Integer
End Class
```

In the preceding example, the compiler generates a warning message both times the code accesses the shared variable `total` through an instance. In each case it makes the access directly through the class `shareTotal` and does not make use of any instance. In the case of the intended call to the procedure `returnClass`, this means it does not even generate a call to `returnClass`, so the additional action of displaying "Function returnClass() called" is not performed.

The `Shared` modifier can be used in these contexts:

[Dim Statement](#)

[Event Statement](#)

[Function Statement](#)

[Operator Statement](#)

[Property Statement](#)

[Sub Statement](#)

See also

- [Shadows](#)
- [Static](#)
- [Lifetime in Visual Basic](#)
- [Procedures](#)
- [Structures](#)
- [Objects and Classes](#)

Static (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

Specifies that one or more declared local variables are to continue to exist and retain their latest values after termination of the procedure in which they are declared.

Remarks

Normally, a local variable in a procedure ceases to exist as soon as the procedure stops. A static variable continues to exist and retains its most recent value. The next time your code calls the procedure, the variable is not reinitialized, and it still holds the latest value that you assigned to it. A static variable continues to exist for the lifetime of the class or module that it is defined in.

Rules

- **Declaration Context.** You can use `Static` only on local variables. This means the declaration context for a `Static` variable must be a procedure or a block in a procedure, and it cannot be a source file, namespace, class, structure, or module.

You cannot use `Static` inside a structure procedure.
- The data types of `Static` local variables cannot be inferred. For more information, see [Local Type Inference](#).
- **Combined Modifiers.** You cannot specify `Static` together with `ReadOnly`, `Shadows`, or `Shared` in the same declaration.

Behavior

When you declare a static variable in a `Shared` procedure, only one copy of the static variable is available for the whole application. You call a `Shared` procedure by using the class name, not a variable that points to an instance of the class.

When you declare a static variable in a procedure that isn't `Shared`, only one copy of the variable is available for each instance of the class. You call a non-shared procedure by using a variable that points to a specific instance of the class.

Example

The following example demonstrates the use of `Static`.

```
Function updateSales(ByVal thisSale As Decimal) As Decimal
    Static totalSales As Decimal = 0
    totalSales += thisSale
    Return totalSales
End Function
```

The `Static` variable `totalSales` is initialized to 0 only one time. Each time that you enter `updateSales`, `totalSales` still has the most recent value that you calculated for it.

The `Static` modifier can be used in this context:

See also

- [Shadows](#)
- [Shared](#)
- [Lifetime in Visual Basic](#)
- [Variable Declaration](#)
- [Structures](#)
- [Local Type Inference](#)
- [Objects and Classes](#)

Unicode (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Specifies that Visual Basic should marshal all strings to Unicode values regardless of the name of the external procedure being declared.

When you call a procedure defined outside your project, the Visual Basic compiler does not have access to the information it must have in order to call the procedure correctly. This information includes where the procedure is located, how it is identified, its calling sequence and return type, and the string character set it uses. The [Declare Statement](#) creates a reference to an external procedure and supplies this necessary information.

The `charsetmodifier` part in the `Declare` statement supplies the character set information to marshal strings during a call to the external procedure. It also affects how Visual Basic searches the external file for the external procedure name. The `Unicode` modifier specifies that Visual Basic should marshal all strings to Unicode values and should look up the procedure without modifying its name during the search.

If no character set modifier is specified, `Ansi` is the default.

Remarks

The `Unicode` modifier can be used in this context:

[Declare Statement](#)

Smart Device Developer Notes

This keyword is not supported.

See also

- [Ansi](#)
- [Auto](#)
- [Keywords](#)

Widening (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Indicates that a conversion operator (`CType`) converts a class or structure to a type that can hold all possible values of the original class or structure.

Converting with the Widening Keyword

The conversion procedure must specify `Public Shared` in addition to `Widening`.

Widening conversions always succeed at run time and never incur data loss. Examples are `Single` to `Double`, `Char` to `String`, and a derived type to its base type. This last conversion is widening because the derived type contains all the members of the base type and thus is an instance of the base type.

The consuming code does not have to use `CType` for widening conversions, even if `Option Strict` is `On`.

The `Widening` keyword can be used in this context:

Operator Statement

For example definitions of widening and narrowing conversion operators, see [How to: Define a Conversion Operator](#).

See also

- [Operator Statement](#)
- [Narrowing](#)
- [Widening and Narrowing Conversions](#)
- [How to: Define an Operator](#)
- [CType Function](#)
- [Option Strict Statement](#)
- [How to: Define a Conversion Operator](#)

WithEvents (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Specifies that one or more declared member variables refer to an instance of a class that can raise events.

Remarks

When a variable is defined using `WithEvents`, you can declaratively specify that a method handles the variable's events using the `Handles` keyword.

You can use `WithEvents` only at class or module level. This means the declaration context for a `WithEvents` variable must be a class or module and cannot be a source file, namespace, structure, or procedure.

You cannot use `WithEvents` on a structure member.

You can declare only individual variables—not arrays—with `WithEvents`.

Rules

Element Types. You must declare `WithEvents` variables to be object variables so that they can accept class instances. However, you cannot declare them as `Object`. You must declare them as the specific class that can raise the events.

The `WithEvents` modifier can be used in this context: [Dim Statement](#)

Example

```
Dim WithEvents app As Application
```

See also

- [Handles](#)
- [Keywords](#)
- [Events](#)

WriteOnly (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

Specifies that a property can be written but not read.

Remarks

Rules

Declaration Context. You can use `WriteOnly` only at module level. This means the declaration context for a `WriteOnly` property must be a class, structure, or module, and cannot be a source file, namespace, or procedure.

You can declare a property as `WriteOnly`, but not a variable.

When to Use WriteOnly

Sometimes you want the consuming code to be able to set a value but not discover what it is. For example, sensitive data, such as a social registration number or a password, needs to be protected from access by any component that did not set it. In these cases, you can use a `WriteOnly` property to set the value.

IMPORTANT

When you define and use a `WriteOnly` property, consider the following additional protective measures:

- **Overriding.** If the property is a member of a class, allow it to default to `NotOverridable`, and do not declare it `Overridable` or `MustOverride`. This prevents a derived class from making undesired access through an override.
- **Access Level.** If you hold the property's sensitive data in one or more variables, declare them `Private` so that no other code can access them.
- **Encryption.** Store all sensitive data in encrypted form rather than in plain text. If malicious code somehow gains access to that area of memory, it is more difficult to make use of the data. Encryption is also useful if it is necessary to serialize the sensitive data.
- **Resetting.** When the class, structure, or module defining the property is being terminated, reset the sensitive data to default values or to other meaningless values. This gives extra protection when that area of memory is freed for general access.
- **Persistence.** Do not persist any sensitive data, for example on disk, if you can avoid it. Also, do not write any sensitive data to the Clipboard.

The `WriteOnly` modifier can be used in this context:

Property Statement

See also

- [ReadOnly](#)
- [Private](#)
- [Keywords](#)

Modules (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic provides several modules that enable you to simplify common tasks in your code, including manipulating strings, performing mathematical calculations, getting system information, performing file and directory operations, and so on. The following table lists the modules provided by Visual Basic.

Constants	Contains miscellaneous constants. These constants can be used anywhere in your code.
ControlChars	Contains constant control characters for printing and displaying text.
Conversion	Contains members that convert decimal numbers to other bases, numbers to strings, strings to numbers, and one data type to another.
DateAndTime	Contains members that get the current date or time, perform date calculations, return a date or time, set the date or time, or time the duration of a process.
ErrObject	Contains information about run-time errors and methods to raise or clear an error.
FileSystem	Contains members that perform file, directory or folder, and system operations.
Financial	Contains procedures that are used to perform financial calculations.
Globals	Contains information about the current scripting engine version.
Information	Contains the members that return, test for, or verify information such as array size, type names, and so on.
Interaction	Contains members interact with objects, applications, and systems.
Strings	Contains members that perform string operations such as reformatting strings, searching a string, getting the length of a string, and so on.
VBMATH	Contains members perform mathematical operations.

See also

- [Visual Basic Language Reference](#)
- [Visual Basic](#)

Nothing (Visual Basic)

8/22/2019 • 3 minutes to read • [Edit Online](#)

Represents the default value of any data type. For reference types, the default value is the `null` reference. For value types, the default value depends on whether the value type is nullable.

NOTE

For non-nullable value types, `Nothing` in Visual Basic differs from `null` in C#. In Visual Basic, if you set a variable of a non-nullable value type to `Nothing`, the variable is set to the default value for its declared type. In C#, if you assign a variable of a non-nullable value type to `null`, a compile-time error occurs.

Remarks

`Nothing` represents the default value of a data type. The default value depends on whether the variable is of a value type or of a reference type.

A variable of a *value type* directly contains its value. Value types include all numeric data types, `Boolean`, `Char`, `Date`, all structures, and all enumerations. A variable of a *reference type* stores a reference to an instance of the object in memory. Reference types include classes, arrays, delegates, and strings. For more information, see [Value Types and Reference Types](#).

If a variable is of a value type, the behavior of `Nothing` depends on whether the variable is of a nullable data type. To represent a nullable value type, add a `?` modifier to the type name. Assigning `Nothing` to a nullable variable sets the value to `null`. For more information and examples, see [Nullable Value Types](#).

If a variable is of a value type that is not nullable, assigning `Nothing` to it sets it to the default value for its declared type. If that type contains variable members, they are all set to their default values. The following example illustrates this for scalar types.

```

Module Module1

Sub Main()
    Dim ts As TestStruct
    Dim i As Integer
    Dim b As Boolean

    ' The following statement sets ts.Name to null and ts.Number to 0.
    ts = Nothing

    ' The following statements set i to 0 and b to False.
    i = Nothing
    b = Nothing

    Console.WriteLine($"ts.Name: {ts.Name}")
    Console.WriteLine($"ts.Number: {ts.Number}")
    Console.WriteLine($"i: {i}")
    Console.WriteLine($"b: {b}")

    Console.ReadKey()
End Sub

Public Structure TestStruct
    Public Name As String
    Public Number As Integer
End Structure
End Module

```

If a variable is of a reference type, assigning `Nothing` to the variable sets it to a `null` reference of the variable's type. A variable that is set to a `null` reference is not associated with any object. The following example demonstrates this.

```

Module Module1

Sub Main()

    Dim testObject As Object
    ' The following statement sets testObject so that it does not refer to
    ' any instance.
    testObject = Nothing

    Dim tc As New TestClass
    tc = Nothing
    ' The fields of tc cannot be accessed. The following statement causes
    ' a NullReferenceException at run time. (Compare to the assignment of
    ' Nothing to structure ts in the previous example.)
    'Console.WriteLine(tc.Field1)

    End Sub

    Class TestClass
        Public Field1 As Integer
        ' ...
    End Class
End Module

```

When checking whether a reference (or nullable value type) variable is `null`, do not use `= Nothing` or `<> Nothing`. Always use `Is Nothing` or `IsNot Nothing`.

For strings in Visual Basic, the empty string equals `Nothing`. Therefore, `"" = Nothing` is true.

The following example shows comparisons that use the `Is` and `IsNot` operators.

```

Module Module1
    Sub Main()

        Dim testObject As Object
        testObject = Nothing
        Console.WriteLine(testObject Is Nothing)
        ' Output: True

        Dim tc As New TestClass
        tc = Nothing
        Console.WriteLine(tc IsNot Nothing)
        ' Output: False

        ' Declare a nullable value type.
        Dim n? As Integer
        Console.WriteLine(n Is Nothing)
        ' Output: True

        n = 4
        Console.WriteLine(n Is Nothing)
        ' Output: False

        n = Nothing
        Console.WriteLine(n IsNot Nothing)
        ' Output: False

        Console.ReadKey()
    End Sub

    Class TestClass
        Public Field1 As Integer
        Private field2 As Boolean
    End Class
End Module

```

If you declare a variable without using an `As` clause and set it to `Nothing`, the variable has a type of `Object`. An example of this is `Dim something = Nothing`. A compile-time error occurs in this case when `Option Strict` is on and `Option Infer` is off.

When you assign `Nothing` to an object variable, it no longer refers to any object instance. If the variable had previously referred to an instance, setting it to `Nothing` does not terminate the instance itself. The instance is terminated, and the memory and system resources associated with it are released, only after the garbage collector (GC) detects that there are no active references remaining.

`Nothing` differs from the `DBNull` object, which represents an uninitialized variant or a nonexistent database column.

See also

- [Dim Statement](#)
- [Object Lifetime: How Objects Are Created and Destroyed](#)
- [Lifetime in Visual Basic](#)
- [Is Operator](#)
- [IsNot Operator](#)
- [Nullable Value Types](#)

Objects (Visual Basic)

5/18/2019 • 2 minutes to read • [Edit Online](#)

This topic provides links to other topics that document the Visual Basic run-time objects and contain tables of their member procedures, properties, and events.

Visual Basic Run-time Objects

Collection	Provides a convenient way to see a related group of items as a single object.
Err	Contains information about run-time errors.
<p>The <code>My.Application</code> object consists of the following classes:</p> <p>ApplicationBase provides members that are available in all projects.</p> <p>WindowsFormsApplicationBase provides members available in Windows Forms applications.</p> <p>ConsoleApplicationBase provides members available in console applications.</p>	<p>Provides data that is associated only with the current application or DLL. No system-level information can be altered with <code>My.Application</code>.</p> <p>Some members are available only for Windows Forms or console applications.</p>
My.Application.Info (Info)	Provides properties for getting the information about an application, such as the version number, description, loaded assemblies, and so on.
My.Application.Log (Log)	Provides a property and methods to write event and exception information to the application's log listeners.
My.Computer (Computer)	Provides properties for manipulating computer components such as audio, the clock, the keyboard, the file system, and so on.
My.Computer.Audio (Audio)	Provides methods for playing sounds.
My.Computer.Clipboard (Clipboard)	Provides methods for manipulating the Clipboard.
My.Computer.Clock (Clock)	Provides properties for accessing the current local time and Universal Coordinated Time (equivalent to Greenwich Mean Time) from the system clock.
My.Computer.FileSystem (FileSystem)	Provides properties and methods for working with drives, files, and directories.
My.Computer.FileSystem.SpecialDirectories (SpecialDirectories)	Provides properties for accessing commonly referenced directories.

<code>My.Computer.Info</code> (ComputerInfo)	Provides properties for getting information about the computer's memory, loaded assemblies, name, and operating system.
<code>My.Computer.Keyboard</code> (Keyboard)	Provides properties for accessing the current state of the keyboard, such as what keys are currently pressed, and provides a method to send keystrokes to the active window.
<code>My.Computer.Mouse</code> (Mouse)	Provides properties for getting information about the format and configuration of the mouse that is installed on the local computer.
<code>My.Computer.Network</code> (Network)	Provides a property, an event, and methods for interacting with the network to which the computer is connected.
<code>My.Computer.Ports</code> (Ports)	Provides a property and a method for accessing the computer's serial ports.
<code>My.Computer.Registry</code> (RegistryProxy)	Provides properties and methods for manipulating the registry.
My.Forms Object	Provides properties for accessing an instance of each Windows Form declared in the current project.
<code>My.Log</code> (AspLog)	Provides a property and methods for writing event and exception information to the application's log listeners for Web applications.
My.Request Object	<p>Gets the HttpRequest object for the requested page. The <code>My.Request</code> object contains information about the current HTTP request.</p> <p>The <code>My.Request</code> object is available only for ASP.NET applications.</p>
My.Resources Object	Provides properties and classes for accessing an application's resources.
My.Response Object	<p>Gets the HttpResponse object that is associated with the Page. This object allows you to send HTTP response data to a client and contains information about that response.</p> <p>The <code>My.Response</code> object is available only for ASP.NET applications.</p>
My.Settings Object	Provides properties and methods for accessing an application's settings.
<code>My.User</code> (User)	Provides access to information about the current user.
My.WebServices Object	Provides properties for creating and accessing a single instance of each Web service that is referenced by the current project.

[TextFieldParser](#)

Provides methods and properties for parsing structured text files.

See also

- [Visual Basic Language Reference](#)
- [Visual Basic](#)

My.Application Object

4/28/2019 • 2 minutes to read • [Edit Online](#)

Provides properties, methods, and events related to the current application.

Remarks

For information about the methods and properties of the `My.Application` object, see the following resources:

- [ApplicationBase](#) for members that are available in all projects.
- [WindowsFormsApplicationBase](#) for members that are available in Windows Forms applications.
- [ConsoleApplicationBase](#) for members that are available in console applications.

Requirements

Namespace: Microsoft.VisualBasic.ApplicationServices

Class: [WindowsFormsApplicationBase](#) (the base class [ConsoleApplicationBase](#) provides members available in console applications, and its base class [ApplicationBase](#) provides the members that are available in all projects)

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- [My.Application.Info Object](#)
- [My.Application.Log Object](#)

My.Application.Info Object

8/22/2019 • 2 minutes to read • [Edit Online](#)

Provides properties for getting the information about the application, such as the version number, description, loaded assemblies, and so on.

Remarks

For information about the methods and properties of the `My.Application.Info` object, see [AssemblyInfo](#).

NOTE

You can use properties of the `System.Diagnostics.FileVersionInfo` class to obtain information about a file on disk.

Requirements

Namespace: Microsoft.VisualBasic.ApplicationServices

Class: [AssemblyInfo](#)

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- [My.Application Object](#)

My.Application.Log Object

8/22/2019 • 2 minutes to read • [Edit Online](#)

Provides a property and methods to write event and exception information to the application's log listeners.

Remarks

For information about the methods and properties of the `My.Application.Log` object, see [Log](#).

For more information, see [Logging Information from the Application](#).

NOTE

You can also use classes in the .NET Framework to log information from your application. For more information, see [Tracing and Instrumenting Applications](#).

Requirements

Namespace: [Microsoft.VisualBasic.Logging](#)

Class: [Log](#)

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- [My.Application Object](#)

My.Computer Object

4/2/2019 • 2 minutes to read • [Edit Online](#)

Provides properties for manipulating computer components such as audio, the clock, the keyboard, the file system, and so on.

Remarks

For information about the methods and properties of the `My.Computer` object, see [Computer](#). The base class [ServerComputer](#) provides the members that are available in all projects.

Requirements

Namespace: [Microsoft.VisualBasic.Devices](#)

Class: [Computer](#) (the base class [ServerComputer](#) provides the members that are available in all projects).

Assembly: Visual Basic Runtime Library (in [Microsoft.VisualBasic.dll](#))

See also

- [My.Computer.Audio Object](#)
- [My.Computer.Clipboard Object](#)
- [My.Computer.Clock Object](#)
- [My.Computer.FileSystem Object](#)
- [My.Computer.FileSystem.SpecialDirectories Object](#)
- [My.Computer.Info Object](#)
- [My.Computer.Keyboard Object](#)
- [My.Computer.Mouse Object](#)
- [My.Computer.Network Object](#)
- [My.Computer.Ports Object](#)
- [My.Computer.Registry Object](#)

My.Computer.Audio Object

4/2/2019 • 2 minutes to read • [Edit Online](#)

Provides methods for playing sounds.

Remarks

For information about the methods and properties of the `My.Computer.Audio` object, see [Audio](#).

For more information, see [Playing Sounds](#).

Requirements

Namespace: [Microsoft.VisualBasic.Devices](#)

Class: [Audio](#)

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- [My.Computer Object](#)

My.Computer.Clipboard Object

8/22/2019 • 2 minutes to read • [Edit Online](#)

Provides methods for manipulating the Clipboard.

Remarks

For information about the methods and properties of the `My.Computer.Clipboard` object, see [ClipboardProxy](#).

For more information, see [Storing Data to and Reading from the Clipboard](#).

NOTE

You can also use methods of the `System.Windows.Forms.Clipboard` class to manipulate the Clipboard.

Requirements

Namespace: Microsoft.VisualBasic.MyServices

Class: ClipboardProxy (provides access to `Clipboard`)

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- [Clipboard](#)
- [My.Computer Object](#)

My.Computer.Clock Object

4/2/2019 • 2 minutes to read • [Edit Online](#)

Provides properties for accessing the current local time and Universal Coordinated Time (equivalent to Greenwich Mean Time) from the system clock.

Remarks

For information about the methods and properties of the `My.Computer.Clock` object, see [Clock](#).

Requirements

Namespace: [Microsoft.VisualBasic.Devices](#)

Class: [Clock](#)

Assembly: Visual Basic Runtime Library (in `Microsoft.VisualBasic.dll`)

See also

- [My.Computer Object](#)

My.Computer.FileSystem Object

8/22/2019 • 2 minutes to read • [Edit Online](#)

Provides properties and methods for working with drives, files, and directories.

Remarks

For information about the methods and properties of the `My.Computer.FileSystem` object, see [FileSystem](#).

For more information, see [File Access with Visual Basic](#).

NOTE

You can also use classes in the `System.IO` namespace to work with drives, files, and directories.

Requirements

Namespace: `Microsoft.VisualBasic.MyServices`

Class: `FileSystemProxy` (provides access to `FileSystem`)

Assembly: Visual Basic Runtime Library (in `Microsoft.VisualBasic.dll`)

See also

- [My.Computer.FileSystem.SpecialDirectories Object](#)
- [My.Computer Object](#)

My.Computer.FileSystem.SpecialDirectories Object

6/13/2019 • 2 minutes to read • [Edit Online](#)

Provides properties for accessing commonly referenced directories.

Remarks

For information about the methods and properties of the `My.Computer.FileSystem.SpecialDirectories` object, see [SpecialDirectories](#).

For more information, see [How to: Retrieve the Contents of the My Documents Directory](#).

Requirements

Namespace: [Microsoft.VisualBasic.MyServices](#)

Class: [SpecialDirectoriesProxy](#) (provides access to [SpecialDirectories](#))

Assembly: Visual Basic Runtime Library (in `Microsoft.VisualBasic.dll`)

See also

- [My.Computer.FileSystem Object](#)
- [My.Computer Object](#)

My.Computer.Info Object

4/2/2019 • 2 minutes to read • [Edit Online](#)

Provides properties for getting information about the computer's memory, loaded assemblies, name, and operating system.

Remarks

For information about the properties of the `My.Computer.Info` object, see [ComputerInfo](#).

Requirements

Namespace: [Microsoft.VisualBasic.Devices](#)

Class: [ComputerInfo](#)

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- [My.Computer Object](#)

My.Computer.Keyboard Object

4/2/2019 • 2 minutes to read • [Edit Online](#)

Provides properties for accessing the current state of the keyboard, such as what keys are currently pressed, and provides a method to send keystrokes to the active window.

Remarks

For information about the methods and properties of the `My.Computer.Keyboard` object, see [Keyboard](#).

For more information, see [Accessing the Keyboard](#).

Requirements

Namespace: [Microsoft.VisualBasic.Devices](#)

Class: [Keyboard](#)

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- [My.Computer Object](#)

My.Computer.Mouse Object

4/2/2019 • 2 minutes to read • [Edit Online](#)

Provides properties for getting information about the format and configuration of the mouse installed on the local computer.

Remarks

For information about the methods and properties of the `My.Computer.Mouse` object, see [Mouse](#).

For more information, see [Accessing the Mouse](#).

Requirements

Namespace: [Microsoft.VisualBasic.Devices](#)

Class: [Mouse](#)

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- [My.Computer Object](#)

My.Computer.Network Object

4/2/2019 • 2 minutes to read • [Edit Online](#)

Provides a property, event, and methods for interacting with the network to which the computer is connected.

Remarks

For information about the methods and properties of the `My.Computer.Network` object, see [Network](#).

For more information, see [Performing Network Operations](#).

Requirements

Namespace: [Microsoft.VisualBasic.Devices](#)

Class: [Network](#)

Assembly: Visual Basic Runtime Library (in `Microsoft.VisualBasic.dll`)

See also

- [My.Computer Object](#)

My.Computer.Ports Object

8/22/2019 • 2 minutes to read • [Edit Online](#)

Provides a property and a method for accessing the computer's serial ports.

Remarks

For information about the methods and properties of the `My.Computer.Ports` object, see [Ports](#).

For more information, see [Accessing the Computer's Ports](#).

NOTE

You can also use properties and methods of the `System.IO.Ports.SerialPort` class to access the computer's serial ports.

Requirements

Namespace: Microsoft.VisualBasic.Devices

Class: Ports

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- [My.Computer Object](#)

My.Computer.Registry Object

8/22/2019 • 2 minutes to read • [Edit Online](#)

Provides properties and methods for manipulating the registry.

Remarks

For information about the methods and properties of the `My.Computer.Registry` object, see [RegistryProxy](#).

For more information, see [Reading from and Writing to the Registry](#).

NOTE

You can also manipulate the registry by using methods of the [Microsoft.Win32.Registry](#) class.

Requirements

Namespace: [Microsoft.VisualBasic.MyServices](#)

Class: [RegistryProxy](#) (provides access to [Registry](#))

Assembly: Visual Basic Runtime Library (in [Microsoft.VisualBasic.dll](#))

See also

- [My.Computer Object](#)

My.Forms Object

10/18/2019 • 2 minutes to read • [Edit Online](#)

Provides properties for accessing an instance of each Windows form declared in the current project.

Remarks

The `My.Forms` object provides an instance of each form in the current project. The name of the property is the same as the name of the form that the property accesses.

You can access the forms provided by the `My.Forms` object by using the name of the form, without qualification. Because the property name is the same as the form's type name, this allows you to access a form as if it had a default instance. For example, `My.Forms.Form1.Show` is equivalent to `Form1.Show`.

The `My.Forms` object exposes only the forms associated with the current project. It does not provide access to forms declared in referenced DLLs. To access a form that a DLL provides, you must use the qualified name of the form, written as `DllName.FormName`.

You can use the [OpenForms](#) property to get a collection of all the application's open forms.

The object and its properties are available only for Windows applications.

Properties

Each property of the `My.Forms` object provides access to an instance of a form in the current project. The name of the property is the same as the name of the form that the property accesses, and the property type is the same as the form's type.

NOTE

If there is a name collision, the property name to access a form is `RootNamespace_Namespace_FormName`. For example, consider two forms named `Form1`. If one of these forms is in the root namespace `WindowsApplication1` and in the namespace `Namespace1`, you would access that form through `My.Forms.WindowsApplication1_Namespace1_Form1`.

The `My.Forms` object provides access to the instance of the application's main form that was created on startup. For all other forms, the `My.Forms` object creates a new instance of the form when it is accessed and stores it. Subsequent attempts to access that property return that instance of the form.

You can dispose of a form by assigning `Nothing` to the property for that form. The property setter calls the `Close` method of the form, and then assigns `Nothing` to the stored value. If you assign any value other than `Nothing` to the property, the setter throws an [ArgumentException](#) exception.

You can test whether a property of the `My.Forms` object stores an instance of the form by using the `Is` or `IsNot` operator. You can use those operators to check if the value of the property is `Nothing`.

NOTE

Typically, the `Is` or `IsNot` operator has to read the value of the property to perform the comparison. However, if the property currently stores `Nothing`, the property creates a new instance of the form and then returns that instance. However, the Visual Basic compiler treats the properties of the `My.Forms` object differently and allows the `Is` or `IsNot` operator to check the status of the property without altering its value.

Example

This example changes the title of the default `SidebarMenu` form.

```
Sub ShowSidebarMenu(ByVal newTitle As String)
    If My.Forms.SidebarMenu IsNot Nothing Then
        My.Forms.SidebarMenu.Text = newTitle
    End If
End Sub
```

For this example to work, your project must have a form named `SidebarMenu`.

This code will work only in a Windows Application project.

Requirements

Availability by Project Type

PROJECT TYPE	AVAILABLE
Windows Application	Yes
Class Library	No
Console Application	No
Windows Control Library	No
Web Control Library	No
Windows Service	No
Web Site	No

See also

- [OpenForms](#)
- [Form](#)
- [Close](#)
- [Objects](#)
- [Is Operator](#)
- [IsNot Operator](#)
- [Accessing Application Forms](#)

My.Log Object

6/13/2019 • 2 minutes to read • [Edit Online](#)

Provides a property and methods for writing event and exception information to the application's log listeners.

Remarks

For information about the methods and properties of the `My.Log` object, see [AspLog](#).

The `My.Log` object is available for ASP.NET applications only. For client applications, use [My.Application.Log Object](#).

Requirements

Namespace: [Microsoft.VisualBasic.Logging](#)

Class: [AspLog](#)

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

My.Request Object

8/17/2019 • 2 minutes to read • [Edit Online](#)

Gets the [HttpRequest](#) object for the requested page.

Remarks

The `My.Request` object contains information about the current HTTP request.

The `My.Request` object is available only for ASP.NET applications.

Example

The following example gets the header collection from the `My.Request` object and uses the `My.Response` object to write it to the ASP.NET page.

```
<script runat="server">
    Public Sub ShowHeaders()
        ' Load the header collection from the Request object.
        Dim coll As System.Collections.Specialized.NameValueCollection
        coll = My.Request.Headers

        ' Put the names of all keys into a string array.
        For Each key As String In coll.AllKeys
            My.Response.Write("Key: " & key & "<br>")

            ' Get all values under this key.
            For Each value As String In coll.GetValues(key)
                My.Response.Write("Value: " & _
                    Server.HtmlEncode(value) & "<br>")
            Next
        Next
    End Sub
</script>
```

See also

- [HttpRequest](#)
- [My.Response Object](#)

My.Response Object

8/17/2019 • 2 minutes to read • [Edit Online](#)

Gets the [HttpResponse](#) object associated with the [Page](#). This object allows you to send HTTP response data to a client and contains information about that response.

Remarks

The `My.Response` object contains the current [HttpResponse](#) object associated with the page.

The `My.Response` object is only available for ASP.NET applications.

Example

The following example gets the header collection from the `My.Request` object and uses the `My.Response` object to write it to the ASP.NET page.

```
<script runat="server">
    Public Sub ShowHeaders()
        ' Load the header collection from the Request object.
        Dim coll As System.Collections.Specialized.NameValueCollection
        coll = My.Request.Headers

        ' Put the names of all keys into a string array.
        For Each key As String In coll.AllKeys
            My.Response.Write("Key: " & key & "<br>")

            ' Get all values under this key.
            For Each value As String In coll.GetValues(key)
                My.Response.Write("Value: " & _
                    Server.HtmlEncode(value) & "<br>")
            Next
        Next
    End Sub
</script>
```

See also

- [HttpResponse](#)
- [My.Request Object](#)

My.Resources Object

4/28/2019 • 3 minutes to read • [Edit Online](#)

Provides properties and classes for accessing the application's resources.

Remarks

The `My.Resources` object provides access to the application's resources and lets you dynamically retrieve resources for your application. For more information, see [Managing Application Resources \(.NET\)](#).

The `My.Resources` object exposes only global resources. It does not provide access to resource files associated with forms. You must access the form resources from the form.

You can access the application's culture-specific resource files from the `My.Resources` object. By default, the `My.Resources` object looks up resources from the resource file that matches the culture in the `UICulture` property. However, you can override this behavior and specify a particular culture to use for the resources. For more information, see [Resources in Desktop Apps](#).

Properties

The properties of the `My.Resources` object provide read-only access to your application's resources. To add or remove resources, use the **Project Designer**. You can access resources added through the **Project Designer** by using `My.Resources.resourceName`.

You can also add or remove resource files by selecting your project in **Solution Explorer** and clicking **Add New Item** or **Add Existing Item** from the **Project** menu. You can access resources added in this manner by using `My.Resources.resourceFileName.resourceName`.

Each resource has a name, category, and value, and these resource settings determine how the property to access the resource appears in the `My.Resources` object. For resources added in the **Project Designer**:

- The name determines the name of the property,
- The resource data is the value of the property,
- The category determines the type of the property:

CATEGORY	PROPERTY DATA TYPE
Strings	String
Images	Bitmap
Icons	Icon
Audio	UnmanagedMemoryStream The <code>UnmanagedMemoryStream</code> class derives from the <code>Stream</code> class, so it can be used with methods that take streams, such as the <code>Play</code> method.

CATEGORY	PROPERTY DATA TYPE
Files	- String for text files. - Bitmap for image files. - Icon for icon files. - UnmanagedMemoryStream for sound files.
Other	Determined by the information in the designer's Type column.

Classes

The `My.Resources` object exposes each resource file as a class with shared properties. The class name is the same as the name of the resource file. As described in the previous section, the resources in a resource file are exposed as properties in the class.

Example

This example sets the title of a form to the string resource named `Form1Title` in the application resource file. For the example to work, the application must have a string named `Form1Title` in its resource file.

```
Sub SetFormTitle()
    Me.Text = My.Resources.Form1Title
End Sub
```

Example

This example sets the icon of the form to the icon named `Form1Icon` that is stored in the application's resource file. For the example to work, the application must have an icon named `Form1Icon` in its resource file.

```
Sub SetFormIcon()
    Me.Icon = My.Resources.Form1Icon
End Sub
```

Example

This example sets the background image of a form to the image resource named `Form1Background`, which is in the application resource file. For this example to work, the application must have an image resource named `Form1Background` in its resource file.

```
Sub SetFormBackgroundImage()
    Me.BackgroundImage = My.Resources.Form1Background
End Sub
```

Example

This example plays the sound that is stored as an audio resource named `Form1Greeting` in the application's resource file. For the example to work, the application must have an audio resource named `Form1Greeting` in its resource file. The `My.Computer.Audio.Play` method is available only for Windows Forms applications.

```
Sub PlayFormGreeting()
    My.Computer.Audio.Play(My.Resources.Form1Greeting,
        AudioPlayMode.Background)
End Sub
```

Example

This example retrieves the French-culture version of a string resource of the application. The resource is named `Message`. To change the culture that the `My.Resources` object uses, the example uses `ChangeUICulture`.

For this example to work, the application must have a string named `Message` in its resource file, and the application should have the French-culture version of that resource file, Resources.fr-FR.resx. If the application does not have the French-culture version of the resource file, the `My.Resource` object retrieves the resource from the default-culture resource file.

```
Sub ShowLocalizedMessage()
    Dim culture As String = My.Application.UICulture.Name
    My.Application.ChangeUICulture("fr-FR")
    MsgBox(My.Resources.Message)
    My.Application.ChangeUICulture(culture)
End Sub
```

See also

- [Managing Application Resources \(.NET\)](#)
- [Resources in Desktop Apps](#)

My.Settings Object

4/28/2019 • 2 minutes to read • [Edit Online](#)

Provides properties and methods for accessing the application's settings.

Remarks

The `My.Settings` object provides access to the application's settings and allows you to dynamically store and retrieve property settings and other information for your application. For more information, see [Managing Application Settings \(.NET\)](#).

Properties

The properties of the `My.Settings` object provide access to your application's settings. To add or remove settings, use the **Settings Designer**.

Each setting has a **Name**, **Type**, **Scope**, and **Value**, and these settings determine how the property to access each setting appears in the `My.Settings` object:

- **Name** determines the name of the property.
- **Type** determines the type of the property.
- **Scope** indicates if the property is read-only. If the value is **Application**, the property is read-only; if the value is **User**, the property is read-write.
- **Value** is the default value of the property.

Methods

METHOD	DESCRIPTION
<code>Reload</code>	Reloads the user settings from the last saved values.
<code>Save</code>	Saves the current user settings.

The `My.Settings` object also provides advanced properties and methods, inherited from the [ApplicationSettingsBase](#) class.

Tasks

The following table lists examples of tasks involving the `My.Settings` object.

TO	SEE
Read an application setting	How to: Read Application Settings in Visual Basic
Change a user setting	How to: Change User Settings in Visual Basic
Persist user settings	How to: Persist User Settings in Visual Basic

TO	SEE
Create a property grid for user settings	How to: Create Property Grids for User Settings in Visual Basic

Example

This example displays the value of the `Nickname` setting.

```
Sub ShowNickname()
    MsgBox("Nickname is " & My.Settings.Nickname)
End Sub
```

For this example to work, your application must have a `Nickname` setting, of type `String`.

See also

- [ApplicationSettingsBase](#)
- [How to: Read Application Settings in Visual Basic](#)
- [How to: Change User Settings in Visual Basic](#)
- [How to: Persist User Settings in Visual Basic](#)
- [How to: Create Property Grids for User Settings in Visual Basic](#)
- [Managing Application Settings \(.NET\)](#)

My.User Object

4/2/2019 • 2 minutes to read • [Edit Online](#)

Provides access to information about the current user.

Remarks

For information about the methods and properties of the `My.User` object, see [Microsoft.VisualBasic.ApplicationServices.User](#).

For more information, see [Accessing User Data](#).

Requirements

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- [IPrincipal](#)
- [CurrentPrincipal](#)
- [User](#)
- [Current](#)

My.WebServices Object

8/22/2019 • 2 minutes to read • [Edit Online](#)

Provides properties for creating and accessing a single instance of each XML Web service referenced by the current project.

Remarks

The `My.WebServices` object provides an instance of each Web service referenced by the current project. Each instance is instantiated on demand. You can access these Web services through the properties of the `My.WebServices` object. The name of the property is the same as the name of the Web service that the property accesses. Any class that inherits from `SoapHttpClientProtocol` is a Web service. For information about adding Web services to a project, see [Accessing Application Web Services](#).

The `My.WebServices` object exposes only the Web services associated with the current project. It does not provide access to Web services declared in referenced DLLs. To access a Web service that a DLL provides, you must use the qualified name of the Web service, in the form `DllName.WebServiceName`. For more information, see [Accessing Application Web Services](#).

The object and its properties are not available for Web applications.

Properties

Each property of the `My.WebServices` object provides access to an instance of a Web service referenced by the current project. The name of the property is the same as the name of the Web service that the property accesses, and the property type is the same as the Web service's type.

NOTE

If there is a name collision, the property name for accessing a Web service is `RootNamespace_Namespace_ServiceName`. For example, consider two Web services named `Service1`. If one of these services is in the root namespace `WindowsApplication1` and in the namespace `Namespace1`, you would access that service by using `My.WebServices.WindowsApplication1_Namespace1_Service1`.

When you first access one of the `My.WebServices` object's properties, it creates a new instance of the Web service and stores it. Subsequent accesses of that property return that instance of the Web service.

You can dispose of a Web service by assigning `Nothing` to the property for that Web service. The property setter assigns `Nothing` to the stored value. If you assign any value other than `Nothing` to the property, the setter throws an [ArgumentException](#) exception.

You can test whether a property of the `My.WebServices` object stores an instance of the Web service by using the `Is` or `IsNot` operator. You can use those operators to check if the value of the property is `Nothing`.

NOTE

Typically, the `Is` or `IsNot` operator has to read the value of the property to perform the comparison. However, if the property currently stores `Nothing`, the property creates a new instance of the Web service and then returns that instance. However, the Visual Basic compiler treats the properties of the `My.WebServices` object specially, and allows the `Is` or `IsNot` operator to check the status of the property without altering its value.

Example

This example calls the `FahrenheitToCelsius` method of the `TemperatureConverter` XML Web service, and returns the result.

```
Function ConvertFromFahrenheitToCelsius(  
    ByVal dFahrenheit As Double) As Double  
  
    Return My.WebServices.TemperatureConverter.FahrenheitToCelsius(dFahrenheit)  
End Function
```

For this example to work, your project must reference a Web service named `Converter`, and that Web service must expose the `ConvertTemperature` method. For more information, see [Accessing Application Web Services](#).

This code does not work in a Web application project.

Requirements

Availability by Project Type

PROJECT TYPE	AVAILABLE
Windows Application	Yes
Class Library	Yes
Console Application	Yes
Windows Control Library	Yes
Web Control Library	Yes
Windows Service	Yes
Web Site	No

See also

- [SoapHttpClientProtocol](#)
- [ArgumentException](#)
- [Accessing Application Web Services](#)

TextFieldParser Object

9/28/2019 • 2 minutes to read • [Edit Online](#)

Provides methods and properties for parsing structured text files.

Syntax

```
Public Class TextFieldParser
```

Remarks

For information about the methods and properties of the `TextFieldParser` object, see [TextFieldParser](#).

For more information, see [Reading from Files](#).

Requirements

Namespace: [Microsoft.VisualBasic.FileIO](#)

Class: `TextFieldParser`

Assembly: Visual Basic Runtime Library (in `Microsoft.VisualBasic.dll`)

Operators (Visual Basic)

10/19/2018 • 2 minutes to read • [Edit Online](#)

In this section

[Operator precedence in Visual Basic](#)

[Operators listed by functionality](#)

[Data types of operator results](#)

[DirectCast operator](#)

[TryCast operator](#)

[New operator](#)

[Null-conditional operators](#)

[Arithmetic operators](#)

[Assignment operators](#)

[Bit Shift operators](#)

[Comparison operators](#)

[Concatenation operators](#)

[Logical/Bitwise operators](#)

[Miscellaneous operators](#)

Related sections

[Visual Basic language reference](#)

[Visual Basic](#)

Operator Precedence in Visual Basic

8/21/2019 • 3 minutes to read • [Edit Online](#)

When several operations occur in an expression, each part is evaluated and resolved in a predetermined order called *operator precedence*.

Precedence Rules

When expressions contain operators from more than one category, they are evaluated according to the following rules:

- The arithmetic and concatenation operators have the order of precedence described in the following section, and all have greater precedence than the comparison, logical, and bitwise operators.
- All comparison operators have equal precedence, and all have greater precedence than the logical and bitwise operators, but lower precedence than the arithmetic and concatenation operators.
- The logical and bitwise operators have the order of precedence described in the following section, and all have lower precedence than the arithmetic, concatenation, and comparison operators.
- Operators with equal precedence are evaluated left to right in the order in which they appear in the expression.

Precedence Order

Operators are evaluated in the following order of precedence:

Await Operator

Await

Arithmetic and Concatenation Operators

Exponentiation (`^`)

Unary identity and negation (`+`, `-`)

Multiplication and floating-point division (`*`, `/`)

Integer division (`\`)

Modular arithmetic (`Mod`)

Addition and subtraction (`+`, `-`)

String concatenation (`&`)

Arithmetic bit shift (`<<`, `>>`)

Comparison Operators

All comparison operators (`=`, `<>`, `<`, `<=`, `>`, `>=`, `Is`, `IsNot`, `Like`, `TypeOf` ... `Is`)

Logical and Bitwise Operators

Negation (`Not`)

Conjunction (`And`, `AndAlso`)

Inclusive disjunction (`Or`, `OrElse`)

Exclusive disjunction (`Xor`)

Comments

The `=` operator is only the equality comparison operator, not the assignment operator.

The string concatenation operator (`&`) is not an arithmetic operator, but in precedence it is grouped with the arithmetic operators.

The `Is` and `IsNot` operators are object reference comparison operators. They do not compare the values of two objects; they check only to determine whether two object variables refer to the same object instance.

Associativity

When operators of equal precedence appear together in an expression, for example multiplication and division, the compiler evaluates each operation as it encounters it from left to right. The following example illustrates this.

```
Dim n1 As Integer = 96 / 8 / 4
Dim n2 As Integer = (96 / 8) / 4
Dim n3 As Integer = 96 / (8 / 4)
```

The first expression evaluates the division $96 / 8$ (which results in 12) and then the division $12 / 4$, which results in three. Because the compiler evaluates the operations for `n1` from left to right, the evaluation is the same when that order is explicitly indicated for `n2`. Both `n1` and `n2` have a result of three. By contrast, `n3` has a result of 48, because the parentheses force the compiler to evaluate $8 / 4$ first.

Because of this behavior, operators are said to be *left associative* in Visual Basic.

Overriding Precedence and Associativity

You can use parentheses to force some parts of an expression to be evaluated before others. This can override both the order of precedence and the left associativity. Visual Basic always performs operations that are enclosed in parentheses before those outside. However, within parentheses, it maintains ordinary precedence and associativity, unless you use parentheses within the parentheses. The following example illustrates this.

```
Dim a, b, c, d, e, f, g As Double
a = 8.0
b = 3.0
c = 4.0
d = 2.0
e = 1.0
f = a - b + c / d * e
' The preceding line sets f to 7.0. Because of natural operator
' precedence and associativity, it is exactly equivalent to the
' following line.
f = (a - b) + ((c / d) * e)
' The following line overrides the natural operator precedence
' and left associativity.
g = (a - (b + c)) / (d * e)
' The preceding line sets g to 0.5.
```

See also

- [= Operator](#)

- [Is Operator](#)
- [IsNot Operator](#)
- [Like Operator](#)
- [TypeOf Operator](#)
- [Await Operator](#)
- [Operators Listed by Functionality](#)
- [Operators and Expressions](#)

Data Types of Operator Results (Visual Basic)

9/10/2019 • 7 minutes to read • [Edit Online](#)

Visual Basic determines the result data type of an operation based on the data types of the operands. In some cases this might be a data type with a greater range than that of either operand.

Data Type Ranges

The ranges of the relevant data types, in order from smallest to largest, are as follows:

- [Boolean](#) — two possible values
- [SByte](#), [Byte](#) — 256 possible integral values
- [Short](#), [UShort](#) — 65,536 (6.5...E+4) possible integral values
- [Integer](#), [UInteger](#) — 4,294,967,296 (4.2...E+9) possible integral values
- [Long](#), [ULong](#) — 18,446,744,073,709,551,615 (1.8...E+19) possible integral values
- [Decimal](#) — 1.5...E+29 possible integral values, maximum range 7.9...E+28 (absolute value)
- [Single](#) — maximum range 3.4...E+38 (absolute value)
- [Double](#) — maximum range 1.7...E+308 (absolute value)

For more information on Visual Basic data types, see [Data Types](#).

If an operand evaluates to [Nothing](#), the Visual Basic arithmetic operators treat it as zero.

Decimal Arithmetic

Note that the [Decimal](#) data type is neither floating-point nor integer.

If either operand of a [+](#), [-](#), [*](#), [/](#), or [Mod](#) operation is [Decimal](#) and the other is not [Single](#) or [Double](#), Visual Basic widens the other operand to [Decimal](#). It performs the operation in [Decimal](#), and the result data type is [Decimal](#).

Floating-Point Arithmetic

Visual Basic performs most floating-point arithmetic in [Double](#), which is the most efficient data type for such operations. However, if one operand is [Single](#) and the other is not [Double](#), Visual Basic performs the operation in [Single](#). It widens each operand as necessary to the appropriate data type before the operation, and the result has that data type.

/ and ^ Operators

The [/](#) operator is defined only for the [Decimal](#), [Single](#), and [Double](#) data types. Visual Basic widens each operand as necessary to the appropriate data type before the operation, and the result has that data type.

The following table shows the result data types for the [/](#) operator. Note that this table is symmetric; for a given combination of operand data types, the result data type is the same regardless of the order of the operands.

	Decimal	Single	Double	Any integer type
Decimal	Decimal	Single	Double	Decimal
Single	Single	Single	Double	Single
Double	Double	Double	Double	Double
Any integer type	Decimal	Single	Double	Double

The `^` operator is defined only for the `Double` data type. Visual Basic widens each operand as necessary to `Double` before the operation, and the result data type is always `Double`.

Integer Arithmetic

The result data type of an integer operation depends on the data types of the operands. In general, Visual Basic uses the following policies for determining the result data type:

- If both operands of a binary operator have the same data type, the result has that data type. An exception is `Boolean`, which is forced to `Short`.
 - If an unsigned operand participates with a signed operand, the result has a signed type with at least as large a range as either operand.
 - Otherwise, the result usually has the larger of the two operand data types.

Note that the result data type might not be the same as either operand data type.

NOTE

The result data type is not always large enough to hold all possible values resulting from the operation. An [OverflowException](#) exception can occur if the value is too large for the result data type.

Unary + and - Operators

The following table shows the result data types for the two unary operators, `+` and `-`.

	Boolean	SByte	Byte	Short	UShort	Integer	UInteger	Long	ULong
Unary +	Short	SByte	Byte	Short	UShort	Integer	UInteger	Long	ULong
Unary -	Short	SByte	Short	Short	Integer	Integer	Long	Long	Decimal

<< and >> Operators

The following table shows the result data types for the two bit-shift operators, `<<` and `>>`. Visual Basic treats each bit-shift operator as a unary operator on its left operand (the bit pattern to be shifted).

Boolean	SByte	Byte	Short	UShort	Integer	UInteger	Long	ULong	

<code><<</code> , <code>>></code>	Short	SByte	Byte	Short	UShort	Integer	UInteger	Long	ULong
--	-------	-------	------	-------	--------	---------	----------	------	-------

If the left operand is `Decimal`, `Single`, `Double`, or `String`, Visual Basic attempts to convert it to `Long` before the operation, and the result data type is `Long`. The right operand (the number of bit positions to shift) must be `Integer` or a type that widens to `Integer`.

Binary `+, -, *, and Mod Operators`

The following table shows the result data types for the binary `+` and `-` operators and the `*` and `Mod` operators. Note that this table is symmetric; for a given combination of operand data types, the result data type is the same regardless of the order of the operands.

	Boolean	SByte	Byte	Short	UShort	Integer	UInteger	Long	ULong
Boolean	Short	SByte	Short	Short	Integer	Integer	Long	Long	Decimal
SByte	SByte	SByte	Short	Short	Integer	Integer	Long	Long	Decimal
Byte	Short	Short	Byte	Short	UShort	Integer	UInteger	Long	ULong
Short	Short	Short	Short	Short	Integer	Integer	Long	Long	Decimal
UShort	Integer	Integer	UShort	Integer	UShort	Integer	UInteger	Long	ULong
Integer	Integer	Integer	Integer	Integer	Integer	Integer	Long	Long	Decimal
UInteger	Long	Long	UInteger	Long	UInteger	Long	UInteger	Long	ULong
Long	Long	Long	Long	Long	Long	Long	Long	Long	Decimal
ULong	Decimal	Decimal	ULong	Decimal	ULong	Decimal	ULong	Decimal	ULong

\ Operator

The following table shows the result data types for the `\` operator. Note that this table is symmetric; for a given combination of operand data types, the result data type is the same regardless of the order of the operands.

	Boolean	SByte	Byte	Short	UShort	Integer	UInteger	Long	ULong
Boolean	Short	SByte	Short	Short	Integer	Integer	Long	Long	Long
SByte	SByte	SByte	Short	Short	Integer	Integer	Long	Long	Long
Byte	Short	Short	Byte	Short	UShort	Integer	UInteger	Long	ULong
Short	Short	Short	Short	Short	Integer	Integer	Long	Long	Long

UShort	Integer	Integer	UShort	Integer	UShort	Integer	UInteger	Long	ULong
Integer	Integer	Integer	Integer	Integer	Integer	Integer	Long	Long	Long
UInteger	Long	Long	UInteger	Long	UInteger	Long	UInteger	Long	ULong
Long	Long	Long	Long	Long	Long	Long	Long	Long	Long
ULong	Long	Long	ULong	Long	ULong	Long	ULong	Long	ULong

If either operand of the `\` operator is [Decimal](#), [Single](#), or [Double](#), Visual Basic attempts to convert it to [Long](#) before the operation, and the result data type is `Long`.

Relational and Bitwise Comparisons

The result data type of a relational operation (`=`, `<>`, `<`, `>`, `<=`, `>=`) is always `Boolean` [Boolean Data Type](#). The same is true for logical operations (`And`, `AndAlso`, `Not`, `Or`, `OrElse`, `Xor`) on `Boolean` operands.

The result data type of a bitwise logical operation depends on the data types of the operands. Note that `AndAlso` and `OrElse` are defined only for `Boolean`, and Visual Basic converts each operand as necessary to `Boolean` before performing the operation.

=, <>, <, >, <=, and >= Operators

If both operands are `Boolean`, Visual Basic considers `True` to be less than `False`. If a numeric type is compared with a `String`, Visual Basic attempts to convert the `String` to `Double` before the operation. A `Char` or `Date` operand can be compared only with another operand of the same data type. The result data type is always `Boolean`.

Bitwise Not Operator

The following table shows the result data types for the bitwise `Not` operator.

	Boolean	SByte	Byte	Short	UShort	Integer	UInteger	Long	ULong
Not	Boolean	SByte	Byte	Short	UShort	Integer	UInteger	Long	ULong

If the operand is `Decimal`, `Single`, `Double`, or `String`, Visual Basic attempts to convert it to `Long` before the operation, and the result data type is `Long`.

Bitwise And, Or, and Xor Operators

The following table shows the result data types for the bitwise `And`, `or`, and `xor` operators. Note that this table is symmetric; for a given combination of operand data types, the result data type is the same regardless of the order of the operands.

	Boolean	SByte	Byte	Short	UShort	Integer	UInteger	Long	ULong
Boolean	Boolean	SByte	Short	Short	Integer	Integer	Long	Long	Long

SByte	SByte	SByte	Short	Short	Integer	Integer	Long	Long	Long
Byte	Short	Short	Byte	Short	UShort	Integer	UInteger	Long	ULong
Short	Short	Short	Short	Short	Integer	Integer	Long	Long	Long
UShort	Integer	Integer	UShort	Integer	UShort	Integer	UInteger	Long	ULong
Integer	Integer	Integer	Integer	Integer	Integer	Integer	Long	Long	Long
UInteger	Long	Long	UInteger	Long	UInteger	Long	UInteger	Long	ULong
Long	Long	Long	Long	Long	Long	Long	Long	Long	Long
ULong	Long	Long	ULong	Long	ULong	Long	ULong	Long	ULong

If an operand is `Decimal`, `Single`, `Double`, or `String`, Visual Basic attempts to convert it to `Long` before the operation, and the result data type is the same as if that operand had already been `Long`.

Miscellaneous Operators

The `&` operator is defined only for concatenation of `String` operands. Visual Basic converts each operand as necessary to `String` before the operation, and the result data type is always `String`. For the purposes of the `&` operator, all conversions to `String` are considered to be widening, even if `Option Strict` is `On`.

The `Is` and `IsNot` operators require both operands to be of a reference type. The `TypeOf ... Is` expression requires the first operand to be of a reference type and the second operand to be the name of a data type. In all these cases the result data type is `Boolean`.

The `Like` operator is defined only for pattern matching of `String` operands. Visual Basic attempts to convert each operand as necessary to `String` before the operation. The result data type is always `Boolean`.

See also

- [Data Types](#)
- [Operators and Expressions](#)
- [Arithmetic Operators in Visual Basic](#)
- [Comparison Operators in Visual Basic](#)
- [Operators](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Arithmetic Operators](#)
- [Comparison Operators](#)
- [Option Strict Statement](#)

Operators Listed by Functionality (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

See one of the categories listed below, or open this portion of the Help table of contents to see an alphabetical list of Visual Basic operators.

Categories of Operators

OPERATORS	DESCRIPTION
Arithmetic Operators	These operators perform mathematical calculations.
Assignment Operators	These operators perform assignment operations.
Comparison Operators	These operators perform comparisons.
Concatenation Operators	These operators combine strings.
Logical/Bitwise Operators	These operators perform logical operations.
Bit Shift Operators	These operators perform arithmetic shifts on bit patterns.
Miscellaneous Operators	These operators perform miscellaneous operations.

See also

- [Operators and Expressions](#)
- [Operator Precedence in Visual Basic](#)

Arithmetic Operators (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

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)

4/2/2019 • 2 minutes to read • [Edit Online](#)

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](#)
- [Statements](#)

Bit Shift Operators (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

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

[`<< Operator`](#)

[`>> Operator`](#)

See also

- [Operators Listed by Functionality](#)

Comparison Operators (Visual Basic)

8/22/2019 • 5 minutes to read • [Edit Online](#)

The following are the comparison operators defined in Visual Basic.

`<` operator

`<=` operator

`>` operator

`>=` operator

`=` operator

`<>` operator

[Is Operator](#)

[IsNot Operator](#)

[Like Operator](#)

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.

`expression1`, `expression2`

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
<code><</code> (Less than)	<code>expression1 < expression2</code>	<code>expression1 >= expression2</code>
<code><=</code> (Less than or equal to)	<code>expression1 <= expression2</code>	<code>expression1 > expression2</code>
<code>></code> (Greater than)	<code>expression1 > expression2</code>	<code>expression1 <= expression2</code>
<code>>=</code> (Greater than or equal to)	<code>expression1 >= expression2</code>	<code>expression1 < expression2</code>
<code>=</code> (Equal to)	<code>expression1 = expression2</code>	<code>expression1 <> expression2</code>
<code><></code> (Not equal to)	<code>expression1 <> expression2</code>	<code>expression1 = expression2</code>

NOTE

The [= Operator](#) 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](#). For more information, see [Troubleshooting Data Types](#).

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 < È < 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) < (È= 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 <code>String</code>	Sort comparison based on string sorting characteristics.
Both numeric	Objects converted to <code>Double</code> , numeric comparison.
One numeric and one <code>String</code>	The <code>String</code> is converted to a <code>Double</code> and numeric comparison is performed. If the <code>String</code> cannot be converted to <code>Double</code> , an <code>InvalidCastException</code> is thrown.
Either or both are reference types other than <code>String</code>	An <code>InvalidCastException</code> 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](#).

Notice that the `= Operator` 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.

```
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, the first comparison returns `False` and the remaining comparisons return `True`.

See also

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

Concatenation Operators (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

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)

4/2/2019 • 2 minutes to read • [Edit Online](#)

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

[And Operator](#)

[Not Operator](#)

[Or Operator](#)

[Xor Operator](#)

[AndAlso Operator](#)

[OrElse Operator](#)

[IsFalse Operator](#)

[IsTrue Operator](#)

See also

- [Operator Precedence in Visual Basic](#)
- [Logical and Bitwise Operators in Visual Basic](#)

Miscellaneous operators (Visual Basic)

1/23/2019 • 2 minutes to read • [Edit Online](#)

The following are miscellaneous operators defined in Visual Basic.

[?. null-conditional operator](#)

[?\(\) null-conditional operator](#)

[AddressOf operator](#)

[Await operator](#)

[GetType operator](#)

[Function expression](#)

[If operator](#)

[TypeOf operator](#)

See also

- [Operators listed by functionality](#)

& Operator (Visual Basic)

9/28/2019 • 2 minutes to read • [Edit Online](#)

Generates a string concatenation of two expressions.

Syntax

```
result = expression1 & expression2
```

Parts

`result`

Required. Any `string` or `Object` variable.

`expression1`

Required. Any expression with a data type that widens to `String`.

`expression2`

Required. Any expression with a data type that widens to `String`.

Remarks

If the data type of `expression1` or `expression2` is not `String` but widens to `String`, it is converted to `String`. If either of the data types does not widen to `String`, the compiler generates an error.

The data type of `result` is `String`. If one or both expressions evaluate to `Nothing` or have a value of `DBNull.Value`, they are treated as a string with a value of "".

NOTE

The `&` 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](#).

NOTE

The ampersand (&) character can also be used to identify variables as type `Long`. For more information, see [Type Characters](#).

Example

This example uses the `&` operator to force string concatenation. The result is a string value representing the concatenation of the two string operands.

```
Dim sampleStr As String  
sampleStr = "Hello" & " World"  
' The preceding statement sets sampleStr to "Hello World".
```

See also

- [&= Operator](#)
- [Concatenation Operators](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Concatenation Operators in Visual Basic](#)

&= Operator (Visual Basic)

9/28/2019 • 2 minutes to read • [Edit Online](#)

Concatenates a `String` expression to a `String` variable or property and assigns the result to the variable or property.

Syntax

```
variableorproperty &= expression
```

Parts

`variableorproperty`

Required. Any `string` variable or property.

`expression`

Required. Any `string` expression.

Remarks

The element on the left side of the `&=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be [ReadOnly](#). The `&=` operator concatenates the `String` expression on its right to the `String` variable or property on its left, and assigns the result to the variable or property on its left.

Overloading

The [& 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. Overloading the `&` operator affects the behavior of the `&=` operator. If your code uses `&=` on a class or structure that overloads `&`, be sure you understand its redefined behavior. For more information, see [Operator Procedures](#).

Example

The following example uses the `&=` operator to concatenate two `String` variables and assign the result to the first variable.

```
Dim var1 As String = "Hello "
Dim var2 As String = "World!"
var1 &= var2
' The value of var1 is now "Hello World!".
```

See also

- [& Operator](#)
- [+= Operator](#)
- [Assignment Operators](#)
- [Concatenation Operators](#)
- [Operator Precedence in Visual Basic](#)

- Operators Listed by Functionality
- Statements

* Operator (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

Multiplies two numbers.

Syntax

```
number1 * number2
```

Parts

TERM	DEFINITION
number1	Required. Any numeric expression.
number2	Required. Any numeric expression.

Result

The result is the product of `number1` and `number2`.

Supported Types

All numeric types, including the unsigned and floating-point types and `Decimal`.

Remarks

The data type of the result depends on the types of the operands. The following table shows how the data type of the result is determined.

OPERAND DATA TYPES	RESULT DATA TYPE
Both expressions are integral data types (SByte , Byte , Short , UShort , Integer , UInteger , Long , ULong)	A numeric data type appropriate for the data types of <code>number1</code> and <code>number2</code> . See the "Integer Arithmetic" tables in Data Types of Operator Results .
Both expressions are <code>Decimal</code>	<code>Decimal</code>
Both expressions are <code>Single</code>	<code>Single</code>
Either expression is a floating-point data type (<code>Single</code> or <code>Double</code>) but not both <code>Single</code> (note <code>Decimal</code> is not a floating-point data type)	<code>Double</code>

If an expression evaluates to [Nothing](#), it is treated as zero.

Overloading

The `*` 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](#).

Example

This example uses the `*` operator to multiply two numbers. The result is the product of the two operands.

```
Dim testValue As Double
testValue = 2 * 2
' The preceding statement sets testValue to 4.
testValue = 459.35 * 334.9
' The preceding statement sets testValue to 153836.315.
```

See also

- [*= Operator](#)
- [Arithmetic Operators](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Arithmetic Operators in Visual Basic](#)

*= Operator (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

Multiples the value of a variable or property by the value of an expression and assigns the result to the variable or property.

Syntax

```
variableorproperty *= expression
```

Parts

`variableorproperty`

Required. Any numeric variable or property.

`expression`

Required. Any numeric expression.

Remarks

The element on the left side of the `=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be [ReadOnly](#).

The `=` operator first multiplies the value of the expression (on the right-hand side of the operator) by the value of the variable or property (on the left-hand side of the operator). The operator then assigns the result of that operation to the variable or property.

Overloading

The [* 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. Overloading the `*` operator affects the behavior of the `=` operator. If your code uses `=` on a class or structure that overloads `*`, be sure you understand its redefined behavior. For more information, see [Operator Procedures](#).

Example

The following example uses the `=` operator to multiply one `Integer` variable by a second and assign the result to the first variable.

```
Dim var1 As Integer = 10
Dim var2 As Integer = 3
var1 *= var2
' The value of var1 is now 30.
```

See also

- [* Operator](#)
- [Assignment Operators](#)

- Arithmetic Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Statements

+ Operator (Visual Basic)

10/1/2019 • 4 minutes to read • [Edit Online](#)

Adds two numbers or returns the positive value of a numeric expression. Can also be used to concatenate two string expressions.

Syntax

```
expression1 + expression2
```

or

```
+expression1
```

Parts

TERM	DEFINITION
expression1	Required. Any numeric or string expression.
expression2	Required unless the <code>+</code> operator is calculating a negative value. Any numeric or string expression.

Result

If `expression1` and `expression2` are both numeric, the result is their arithmetic sum.

If `expression2` is absent, the `+` operator is the *unary* identity operator for the unchanged value of an expression.

In this sense, the operation consists of retaining the sign of `expression1`, so the result is negative if `expression1` is negative.

If `expression1` and `expression2` are both strings, the result is the concatenation of their values.

If `expression1` and `expression2` are of mixed types, the action taken depends on their types, their contents, and the setting of the [Option Strict Statement](#). For more information, see the tables in "Remarks."

Supported Types

All numeric types, including the unsigned and floating-point types and `Decimal`, and `String`.

Remarks

In general, `+` performs arithmetic addition when possible, and concatenates only when both expressions are strings.

If neither expression is an `Object`, Visual Basic takes the following actions.

DATA TYPES OF EXPRESSIONS	ACTION BY COMPILER
Both expressions are numeric data types (<code>SByte</code> , <code>Byte</code> , <code>Short</code> , <code>UShort</code> , <code>Integer</code> , <code>UInteger</code> , <code>Long</code> , <code>ULong</code> , <code>Decimal</code> , <code>Single</code> , or <code>Double</code>)	Add. The result data type is a numeric type appropriate for the data types of <code>expression1</code> and <code>expression2</code> . See the "Integer Arithmetic" tables in Data Types of Operator Results .
Both expressions are of type <code>String</code>	Concatenate.
One expression is a numeric data type and the other is a string	<p>If <code>Option Strict</code> is <code>on</code>, then generate a compiler error.</p> <p>If <code>Option Strict</code> is <code>off</code>, then implicitly convert the <code>String</code> to <code>Double</code> and add.</p> <p>If the <code>String</code> cannot be converted to <code>Double</code>, then throw an InvalidCastException exception.</p>
One expression is a numeric data type, and the other is <code>Nothing</code>	Add, with <code>Nothing</code> valued as zero.
One expression is a string, and the other is <code>Nothing</code>	Concatenate, with <code>Nothing</code> valued as "".

If one expression is an `Object` expression, Visual Basic takes the following actions.

DATA TYPES OF EXPRESSIONS	ACTION BY COMPILER
<code>Object</code> expression holds a numeric value and the other is a numeric data type	<p>If <code>Option Strict</code> is <code>on</code>, then generate a compiler error.</p> <p>If <code>Option Strict</code> is <code>off</code>, then add.</p>
<code>Object</code> expression holds a numeric value and the other is of type <code>String</code>	<p>If <code>Option Strict</code> is <code>on</code>, then generate a compiler error.</p> <p>If <code>Option Strict</code> is <code>off</code>, then implicitly convert the <code>String</code> to <code>Double</code> and add.</p> <p>If the <code>String</code> cannot be converted to <code>Double</code>, then throw an InvalidCastException exception.</p>
<code>Object</code> expression holds a string and the other is a numeric data type	<p>If <code>Option Strict</code> is <code>on</code>, then generate a compiler error.</p> <p>If <code>Option Strict</code> is <code>off</code>, then implicitly convert the string <code>Object</code> to <code>Double</code> and add.</p> <p>If the string <code>Object</code> cannot be converted to <code>Double</code>, then throw an InvalidCastException exception.</p>
<code>Object</code> expression holds a string and the other is of type <code>String</code>	<p>If <code>Option Strict</code> is <code>on</code>, then generate a compiler error.</p> <p>If <code>Option Strict</code> is <code>off</code>, then implicitly convert <code>Object</code> to <code>String</code> and concatenate.</p>

If both expressions are `Object` expressions, Visual Basic takes the following actions (`Option Strict Off` only).

DATA TYPES OF EXPRESSIONS	ACTION BY COMPILER
Both <code>Object</code> expressions hold numeric values	Add.

DATA TYPES OF EXPRESSIONS	ACTION BY COMPILER
Both <code>Object</code> expressions are of type <code>String</code>	Concatenate.
One <code>Object</code> expression holds a numeric value and the other holds a string	<p>Implicitly convert the string <code>Object</code> to <code>Double</code> and add.</p> <p>If the string <code>Object</code> cannot be converted to a numeric value, then throw an <code>InvalidCastException</code> exception.</p>

If either `Object` expression evaluates to `Nothing` or `DBNull`, the `+` operator treats it as a `String` with a value of `""`.

NOTE

When you use the `+` operator, you might not be able to determine whether addition or string concatenation will occur. Use the `&` operator for concatenation to eliminate ambiguity and to provide self-documenting code.

Overloading

The `+` 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](#).

Example

The following example uses the `+` operator to add numbers. If the operands are both numeric, Visual Basic computes the arithmetic result. The arithmetic result represents the sum of the two operands.

```
Dim sumNumber As Integer
sumNumber = 2 + 2
sumNumber = 4257.04 + 98112
' The preceding statements set sumNumber to 4 and 102369.
```

You can also use the `+` operator to concatenate strings. If the operands are both strings, Visual Basic concatenates them. The concatenation result represents a single string consisting of the contents of the two operands one after the other.

If the operands are of mixed types, the result depends on the setting of the [Option Strict Statement](#). The following example illustrates the result when `Option Strict` is `On`.

```
Option Strict On
```

```
Dim var1 As String = "34"
Dim var2 As Integer = 6
Dim concatenatedNumber As Integer = var1 + var2
```

' The preceding statement generates a COMPILER ERROR.

The following example illustrates the result when `Option Strict` is `Off`.

```
Option Strict Off
```

```
Dim var1 As String = "34"  
Dim var2 As Integer = 6  
Dim concatenatedNumber As Integer = var1 + var2
```

' The preceding statement returns 40 after the string in var1 is
' converted to a numeric value. This might be an unexpected result.
' We do not recommend use of Option Strict Off for these operations.

To eliminate ambiguity, you should use the `&` operator instead of `+` for concatenation.

See also

- [& Operator](#)
- [Concatenation Operators](#)
- [Arithmetic Operators](#)
- [Operators Listed by Functionality](#)
- [Operator Precedence in Visual Basic](#)
- [Arithmetic Operators in Visual Basic](#)
- [Option Strict Statement](#)

+= Operator (Visual Basic)

9/28/2019 • 2 minutes to read • [Edit Online](#)

Adds the value of a numeric expression to the value of a numeric variable or property and assigns the result to the variable or property. Can also be used to concatenate a `String` expression to a `String` variable or property and assign the result to the variable or property.

Syntax

```
variableorproperty += expression
```

Parts

`variableorproperty`

Required. Any numeric or `String` variable or property.

`expression`

Required. Any numeric or `String` expression.

Remarks

The element on the left side of the `+=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be [ReadOnly](#).

The `+=` operator adds the value on its right to the variable or property on its left, and assigns the result to the variable or property on its left. The `+=` operator can also be used to concatenate the `String` expression on its right to the `String` variable or property on its left, and assign the result to the variable or property on its left.

NOTE

When you use the `+=` operator, you might not be able to determine whether addition or string concatenation will occur. Use the `&=` operator for concatenation to eliminate ambiguity and to provide self-documenting code.

This assignment operator implicitly performs widening but not narrowing conversions if the compilation environment enforces strict semantics. For more information on these conversions, see [Widening and Narrowing Conversions](#). For more information on strict and permissive semantics, see [Option Strict Statement](#).

If permissive semantics are allowed, the `+=` operator implicitly performs a variety of string and numeric conversions identical to those performed by the `+` operator. For details on these conversions, see [+ Operator](#).

Overloading

The `+` 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. Overloading the `+` operator affects the behavior of the `+=` operator. If your code uses `+=` on a class or structure that overloads `+`, be sure you understand its redefined behavior. For more information, see [Operator Procedures](#).

Example

The following example uses the `+=` operator to combine the value of one variable with another. The first part uses `+=` with numeric variables to add one value to another. The second part uses `+=` with `String` variables to concatenate one value with another. In both cases, the result is assigned to the first variable.

```
' This part uses numeric variables.  
Dim num1 As Integer = 10  
Dim num2 As Integer = 3  
num1 += num2
```

```
' This part uses string variables.  
Dim str1 As String = "10"  
Dim str2 As String = "3"  
str1 += str2
```

The value of `num1` is now 13, and the value of `str1` is now "103".

See also

- [+ Operator](#)
- [Assignment Operators](#)
- [Arithmetic Operators](#)
- [Concatenation Operators](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Statements](#)

= Operator (Visual Basic)

9/28/2019 • 2 minutes to read • [Edit Online](#)

Assigns a value to a variable or property.

Syntax

```
variableorproperty = value
```

Parts

`variableorproperty`

Any writable variable or any property.

`value`

Any literal, constant, or expression.

Remarks

The element on the left side of the equal sign (`=`) can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be [ReadOnly](#). The `=` operator assigns the value on its right to the variable or property on its left.

NOTE

The `=` operator is also used as a comparison operator. For details, see [Comparison Operators](#).

Overloading

The `=` operator can be overloaded only as a relational comparison operator, not as an assignment operator. For more information, see [Operator Procedures](#).

Example

The following example demonstrates the assignment operator. The value on the right is assigned to the variable on the left.

```
Dim testInt As Integer
Dim testString As String
Dim testButton As System.Windows.Forms.Button
Dim testObject As Object
testInt = 42
testString = "This is an example of a string literal."
testButton = New System.Windows.Forms.Button()
testObject = testInt
testObject = testString
testObject = testButton
```

See also

- [&= Operator](#)
- [*= Operator](#)
- [+= Operator](#)
- [-= Operator \(Visual Basic\)](#)
- [/= Operator \(Visual Basic\)](#)
- [\= Operator](#)
- [^= Operator](#)
- [Statements](#)
- [Comparison Operators](#)
- [ReadOnly](#)
- [Local Type Inference](#)

- Operator (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

Returns the difference between two numeric expressions or the negative value of a numeric expression.

Syntax

```
expression1 - expression2
```

or

```
-expression1
```

Parts

`expression1`

Required. Any numeric expression.

`expression2`

Required unless the `-` operator is calculating a negative value. Any numeric expression.

Result

The result is the difference between `expression1` and `expression2`, or the negated value of `expression1`.

The result data type is a numeric type appropriate for the data types of `expression1` and `expression2`. See the "Integer Arithmetic" tables in [Data Types of Operator Results](#).

Supported Types

All numeric types. This includes the unsigned and floating-point types and `Decimal`.

Remarks

In the first usage shown in the syntax shown previously, the `-` operator is the *binary* arithmetic subtraction operator for the difference between two numeric expressions.

In the second usage shown in the syntax shown previously, the `-` operator is the *unary* negation operator for the negative value of an expression. In this sense, the negation consists of reversing the sign of `expression1` so that the result is positive if `expression1` is negative.

If either expression evaluates to [Nothing](#), the `-` operator treats it as zero.

NOTE

The `-` 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, make sure that you understand its redefined behavior. For more information, see [Operator Procedures](#).

Example

The following example uses the `-` operator to calculate and return the difference between two numbers, and then to negate a number.

```
Dim binaryResult As Double = 459.35 - 334.9
Dim unaryResult As Double = -334.9
```

Following the execution of these statements, `binaryResult` contains 124.45 and `unaryResult` contains -334.90.

See also

- [-= Operator \(Visual Basic\)](#)
- [Arithmetic Operators](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Arithmetic Operators in Visual Basic](#)

-= Operator (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

Subtracts the value of an expression from the value of a variable or property and assigns the result to the variable or property.

Syntax

```
variableorproperty -= expression
```

Parts

`variableorproperty`

Required. Any numeric variable or property.

`expression`

Required. Any numeric expression.

Remarks

The element on the left side of the `-=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be [ReadOnly](#).

The `-=` operator first subtracts the value of the expression (on the right-hand side of the operator) from the value of the variable or property (on the left-hand side of the operator). The operator then assigns the result of that operation to the variable or property.

Overloading

The [- Operator \(Visual Basic\)](#) 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. Overloading the `-` operator affects the behavior of the `-=` operator. If your code uses `-=` on a class or structure that overloads `-`, be sure you understand its redefined behavior. For more information, see [Operator Procedures](#).

Example

The following example uses the `-=` operator to subtract one `Integer` variable from another and assign the result to the latter variable.

```
Dim var1 As Integer = 10
Dim var2 As Integer = 3
var1 -= var2
' The value of var1 is now 7.
```

See also

- [- Operator \(Visual Basic\)](#)
- [Assignment Operators](#)

- [Arithmetic Operators](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Statements](#)

Comparison Operators (Visual Basic)

8/22/2019 • 5 minutes to read • [Edit Online](#)

The following are the comparison operators defined in Visual Basic.

`<` operator

`<=` operator

`>` operator

`>=` operator

`=` operator

`<>` operator

[Is Operator](#)

[IsNot Operator](#)

[Like Operator](#)

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.

`expression1`, `expression2`

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
<code><</code> (Less than)	<code>expression1 < expression2</code>	<code>expression1 >= expression2</code>
<code><=</code> (Less than or equal to)	<code>expression1 <= expression2</code>	<code>expression1 > expression2</code>
<code>></code> (Greater than)	<code>expression1 > expression2</code>	<code>expression1 <= expression2</code>
<code>>=</code> (Greater than or equal to)	<code>expression1 >= expression2</code>	<code>expression1 < expression2</code>
<code>=</code> (Equal to)	<code>expression1 = expression2</code>	<code>expression1 <> expression2</code>
<code><></code> (Not equal to)	<code>expression1 <> expression2</code>	<code>expression1 = expression2</code>

NOTE

The [= Operator](#) 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](#). For more information, see [Troubleshooting Data Types](#).

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 < È < 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) < (È= 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 <code>String</code>	Sort comparison based on string sorting characteristics.
Both numeric	Objects converted to <code>Double</code> , numeric comparison.
One numeric and one <code>String</code>	The <code>String</code> is converted to a <code>Double</code> and numeric comparison is performed. If the <code>String</code> cannot be converted to <code>Double</code> , an <code>InvalidCastException</code> is thrown.
Either or both are reference types other than <code>String</code>	An <code>InvalidCastException</code> 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](#).

Notice that the `= Operator` 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.

```
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, the first comparison returns `False` and the remaining comparisons return `True`.

See also

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

Comparison Operators (Visual Basic)

8/22/2019 • 5 minutes to read • [Edit Online](#)

The following are the comparison operators defined in Visual Basic.

`<` operator

`<=` operator

`>` operator

`>=` operator

`=` operator

`<>` operator

[Is Operator](#)

[IsNot Operator](#)

[Like Operator](#)

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.

`expression1`, `expression2`

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
<code><</code> (Less than)	<code>expression1 < expression2</code>	<code>expression1 >= expression2</code>
<code><=</code> (Less than or equal to)	<code>expression1 <= expression2</code>	<code>expression1 > expression2</code>
<code>></code> (Greater than)	<code>expression1 > expression2</code>	<code>expression1 <= expression2</code>
<code>>=</code> (Greater than or equal to)	<code>expression1 >= expression2</code>	<code>expression1 < expression2</code>
<code>=</code> (Equal to)	<code>expression1 = expression2</code>	<code>expression1 <> expression2</code>
<code><></code> (Not equal to)	<code>expression1 <> expression2</code>	<code>expression1 = expression2</code>

NOTE

The [= Operator](#) 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](#). For more information, see [Troubleshooting Data Types](#).

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 < È < 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) < (È= 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 <code>String</code>	Sort comparison based on string sorting characteristics.
Both numeric	Objects converted to <code>Double</code> , numeric comparison.
One numeric and one <code>String</code>	The <code>String</code> is converted to a <code>Double</code> and numeric comparison is performed. If the <code>String</code> cannot be converted to <code>Double</code> , an <code>InvalidCastException</code> is thrown.
Either or both are reference types other than <code>String</code>	An <code>InvalidCastException</code> 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](#).

Notice that the `= Operator` 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.

```
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, the first comparison returns `False` and the remaining comparisons return `True`.

See also

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

Comparison Operators (Visual Basic)

8/22/2019 • 5 minutes to read • [Edit Online](#)

The following are the comparison operators defined in Visual Basic.

`<` operator

`<=` operator

`>` operator

`>=` operator

`=` operator

`<>` operator

[Is Operator](#)

[IsNot Operator](#)

[Like Operator](#)

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.

`expression1`, `expression2`

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
<code><</code> (Less than)	<code>expression1 < expression2</code>	<code>expression1 >= expression2</code>
<code><=</code> (Less than or equal to)	<code>expression1 <= expression2</code>	<code>expression1 > expression2</code>
<code>></code> (Greater than)	<code>expression1 > expression2</code>	<code>expression1 <= expression2</code>
<code>>=</code> (Greater than or equal to)	<code>expression1 >= expression2</code>	<code>expression1 < expression2</code>
<code>=</code> (Equal to)	<code>expression1 = expression2</code>	<code>expression1 <> expression2</code>
<code><></code> (Not equal to)	<code>expression1 <> expression2</code>	<code>expression1 = expression2</code>

NOTE

The [= Operator](#) 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](#). For more information, see [Troubleshooting Data Types](#).

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 < È < 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) < (È= 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 <code>String</code>	Sort comparison based on string sorting characteristics.
Both numeric	Objects converted to <code>Double</code> , numeric comparison.
One numeric and one <code>String</code>	The <code>String</code> is converted to a <code>Double</code> and numeric comparison is performed. If the <code>String</code> cannot be converted to <code>Double</code> , an <code>InvalidCastException</code> is thrown.
Either or both are reference types other than <code>String</code>	An <code>InvalidCastException</code> 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](#).

Notice that the `= Operator` 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.

```
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, the first comparison returns `False` and the remaining comparisons return `True`.

See also

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

Comparison Operators (Visual Basic)

8/22/2019 • 5 minutes to read • [Edit Online](#)

The following are the comparison operators defined in Visual Basic.

`<` operator

`<=` operator

`>` operator

`>=` operator

`=` operator

`<>` operator

[Is Operator](#)

[IsNot Operator](#)

[Like Operator](#)

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.

`expression1`, `expression2`

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
<code><</code> (Less than)	<code>expression1 < expression2</code>	<code>expression1 >= expression2</code>
<code><=</code> (Less than or equal to)	<code>expression1 <= expression2</code>	<code>expression1 > expression2</code>
<code>></code> (Greater than)	<code>expression1 > expression2</code>	<code>expression1 <= expression2</code>
<code>>=</code> (Greater than or equal to)	<code>expression1 >= expression2</code>	<code>expression1 < expression2</code>
<code>=</code> (Equal to)	<code>expression1 = expression2</code>	<code>expression1 <> expression2</code>
<code><></code> (Not equal to)	<code>expression1 <> expression2</code>	<code>expression1 = expression2</code>

NOTE

The `= Operator` 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`. For more information, see [Troubleshooting Data Types](#).

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 < È < 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) < (È= 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 <code>String</code>	Sort comparison based on string sorting characteristics.
Both numeric	Objects converted to <code>Double</code> , numeric comparison.
One numeric and one <code>String</code>	The <code>String</code> is converted to a <code>Double</code> and numeric comparison is performed. If the <code>String</code> cannot be converted to <code>Double</code> , an <code>InvalidCastException</code> is thrown.
Either or both are reference types other than <code>String</code>	An <code>InvalidCastException</code> 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](#).

Notice that the `= Operator` 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.

```
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, the first comparison returns `False` and the remaining comparisons return `True`.

See also

- [InvalidOperationException](#)
- [= Operator](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Troubleshooting Data Types](#)
- [Comparison Operators in Visual Basic](#)

<< Operator (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

Performs an arithmetic left shift on a bit pattern.

Syntax

```
result = pattern << amount
```

Parts

`result`

Required. Integral numeric value. The result of shifting the bit pattern. The data type is the same as that of `pattern`.

`pattern`

Required. Integral numeric expression. The bit pattern to be shifted. The data type must be an integral type (`SByte`, `Byte`, `Short`, `UShort`, `Integer`, `UInteger`, `Long`, or `ULong`).

`amount`

Required. Numeric expression. The number of bits to shift the bit pattern. The data type must be `Integer` or widen to `Integer`.

Remarks

Arithmetic shifts are not circular, which means the bits shifted off one end of the result are not reintroduced at the other end. In an arithmetic left shift, the bits shifted beyond the range of the result data type are discarded, and the bit positions vacated on the right are set to zero.

To prevent a shift by more bits than the result can hold, Visual Basic masks the value of `amount` with a size mask that corresponds to the data type of `pattern`. The binary AND of these values is used for the shift amount. The size masks are as follows:

DATA TYPE OF <code>PATTERN</code>	SIZE MASK (DECIMAL)	SIZE MASK (HEXADECIMAL)
<code>SByte</code> , <code>Byte</code>	7	&H00000007
<code>Short</code> , <code>UShort</code>	15	&H0000000F
<code>Integer</code> , <code>UInteger</code>	31	&H0000001F
<code>Long</code> , <code>ULong</code>	63	&H0000003F

If `amount` is zero, the value of `result` is identical to the value of `pattern`. If `amount` is negative, it is taken as an unsigned value and masked with the appropriate size mask.

Arithmetic shifts never generate overflow exceptions.

NOTE

The `<<` 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 that you understand its redefined behavior. For more information, see [Operator Procedures](#).

Example

The following example uses the `<<` operator to perform arithmetic left shifts on integral values. The result always has the same data type as that of the expression being shifted.

```
Dim pattern As Short = 192
' The bit pattern is 0000 0000 1100 0000.
Dim result1, result2, result3, result4, result5 As Short
result1 = pattern << 0
result2 = pattern << 4
result3 = pattern << 9
result4 = pattern << 17
result5 = pattern << -1
```

The results of the previous example are as follows:

- `result1` is 192 (0000 0000 1100 0000).
- `result2` is 3072 (0000 1100 0000 0000).
- `result3` is -32768 (1000 0000 0000 0000).
- `result4` is 384 (0000 0001 1000 0000).
- `result5` is 0 (shifted 15 places to the left).

The shift amount for `result4` is calculated as 17 AND 15, which equals 1.

See also

- [Bit Shift Operators](#)
- [Assignment Operators](#)
- [<<= Operator](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Arithmetic Operators in Visual Basic](#)

<<= Operator (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

Performs an arithmetic left shift on the value of a variable or property and assigns the result back to the variable or property.

Syntax

```
variableorproperty <<= amount
```

Parts

`variableorproperty`

Required. Variable or property of an integral type (`SByte`, `Byte`, `Short`, `UShort`, `Integer`, `UInteger`, `Long`, or `ULong`).

`amount`

Required. Numeric expression of a data type that widens to `Integer`.

Remarks

The element on the left side of the `<<=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be [ReadOnly](#).

The `<<=` operator first performs an arithmetic left shift on the value of the variable or property. The operator then assigns the result of that operation back to that variable or property.

Arithmetic shifts are not circular, which means the bits shifted off one end of the result are not reintroduced at the other end. In an arithmetic left shift, the bits shifted beyond the range of the result data type are discarded, and the bit positions vacated on the right are set to zero.

Overloading

The [<< 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. Overloading the `<<` operator affects the behavior of the `<<=` operator. If your code uses `<<=` on a class or structure that overloads `<<`, be sure you understand its redefined behavior. For more information, see [Operator Procedures](#).

Example

The following example uses the `<<=` operator to shift the bit pattern of an `Integer` variable left by the specified amount and assign the result to the variable.

```
Dim var As Integer = 10
Dim shift As Integer = 3
var <<= shift
' The value of var is now 80.
```

See also

- [<< Operator](#)
- [Assignment Operators](#)
- [Bit Shift Operators](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Statements](#)

>> Operator (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

Performs an arithmetic right shift on a bit pattern.

Syntax

```
result = pattern >> amount
```

Parts

`result`

Required. Integral numeric value. The result of shifting the bit pattern. The data type is the same as that of `pattern`.

`pattern`

Required. Integral numeric expression. The bit pattern to be shifted. The data type must be an integral type (`SByte`, `Byte`, `Short`, `UShort`, `Integer`, `UInteger`, `Long`, or `ULong`).

`amount`

Required. Numeric expression. The number of bits to shift the bit pattern. The data type must be `Integer` or widen to `Integer`.

Remarks

Arithmetic shifts are not circular, which means the bits shifted off one end of the result are not reintroduced at the other end. In an arithmetic right shift, the bits shifted beyond the rightmost bit position are discarded, and the leftmost (sign) bit is propagated into the bit positions vacated at the left. This means that if `pattern` has a negative value, the vacated positions are set to one; otherwise they are set to zero.

Note that the data types `Byte`, `UShort`, `UInteger`, and `ULong` are unsigned, so there is no sign bit to propagate. If `pattern` is of any unsigned type, the vacated positions are always set to zero.

To prevent shifting by more bits than the result can hold, Visual Basic masks the value of `amount` with a size mask corresponding to the data type of `pattern`. The binary AND of these values is used for the shift amount. The size masks are as follows:

DATA TYPE OF <code>PATTERN</code>	SIZE MASK (DECIMAL)	SIZE MASK (HEXADECIMAL)
<code>SByte</code> , <code>Byte</code>	7	&H00000007
<code>Short</code> , <code>UShort</code>	15	&H0000000F
<code>Integer</code> , <code>UInteger</code>	31	&H0000001F
<code>Long</code> , <code>ULong</code>	63	&H0000003F

If `amount` is zero, the value of `result` is identical to the value of `pattern`. If `amount` is negative, it is taken as an unsigned value and masked with the appropriate size mask.

Arithmetic shifts never generate overflow exceptions.

Overloading

The `>>` 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](#).

Example

The following example uses the `>>` operator to perform arithmetic right shifts on integral values. The result always has the same data type as that of the expression being shifted.

```
Dim pattern As Short = 2560
' The bit pattern is 0000 1010 0000 0000.
Dim result1, result2, result3, result4, result5 As Short
result1 = pattern >> 0
result2 = pattern >> 4
result3 = pattern >> 10
result4 = pattern >> 18
result5 = pattern >> -1
```

The results of the preceding example are as follows:

- `result1` is 2560 (0000 1010 0000 0000).
- `result2` is 160 (0000 0000 1010 0000).
- `result3` is 2 (0000 0000 0000 0010).
- `result4` is 640 (0000 0010 1000 0000).
- `result5` is 0 (shifted 15 places to the right).

The shift amount for `result4` is calculated as 18 AND 15, which equals 2.

The following example shows arithmetic shifts on a negative value.

```
Dim negPattern As Short = -8192
' The bit pattern is 1110 0000 0000 0000.
Dim negResult1, negResult2 As Short
negResult1 = negPattern >> 4
negResult2 = negPattern >> 13
```

The results of the preceding example are as follows:

- `negResult1` is -512 (1111 1110 0000 0000).
- `negResult2` is -1 (the sign bit is propagated).

See also

- [Bit Shift Operators](#)
- [Assignment Operators](#)
- [>>= Operator](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)

- Arithmetic Operators in Visual Basic

>>= Operator (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

Performs an arithmetic right shift on the value of a variable or property and assigns the result back to the variable or property.

Syntax

```
variableorproperty >>= amount
```

Parts

`variableorproperty`

Required. Variable or property of an integral type (`sByte`, `Byte`, `Short`, `UShort`, `Integer`, `UInteger`, `Long`, or `ULong`).

`amount`

Required. Numeric expression of a data type that widens to `Integer`.

Remarks

The element on the left side of the `>>=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be [ReadOnly](#).

The `>>=` operator first performs an arithmetic right shift on the value of the variable or property. The operator then assigns the result of that operation back to the variable or property.

Arithmetic shifts are not circular, which means the bits shifted off one end of the result are not reintroduced at the other end. In an arithmetic right shift, the bits shifted beyond the rightmost bit position are discarded, and the leftmost bit is propagated into the bit positions vacated at the left. This means that if `variableorproperty` has a negative value, the vacated positions are set to one. If `variableorproperty` is positive, or if its data type is an unsigned type, the vacated positions are set to zero.

Overloading

The [>> 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. Overloading the `>>` operator affects the behavior of the `>>=` operator. If your code uses `>>=` on a class or structure that overloads `>>`, be sure you understand its redefined behavior. For more information, see [Operator Procedures](#).

Example

The following example uses the `>>=` operator to shift the bit pattern of an `Integer` variable right by the specified amount and assign the result to the variable.

```
Dim var As Integer = 10
Dim shift As Integer = 2
var >>= shift
' The value of var is now 2 (two bits were lost off the right end).
```

See also

- [>> Operator](#)
- [Assignment Operators](#)
- [Bit Shift Operators](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Statements](#)

/ Operator (Visual Basic)

9/28/2019 • 2 minutes to read • [Edit Online](#)

Divides two numbers and returns a floating-point result.

Syntax

```
expression1 / expression2
```

Parts

`expression1`

Required. Any numeric expression.

`expression2`

Required. Any numeric expression.

Supported Types

All numeric types, including the unsigned and floating-point types and `Decimal`.

Result

The result is the full quotient of `expression1` divided by `expression2`, including any remainder.

The [\ Operator \(Visual Basic\)](#) returns the integer quotient, which drops the remainder.

Remarks

The data type of the result depends on the types of the operands. The following table shows how the data type of the result is determined.

OPERAND DATA TYPES	RESULT DATA TYPE
Both expressions are integral data types (<code>SByte</code> , <code>Byte</code> , <code>Short</code> , <code>UShort</code> , <code>Integer</code> , <code>UInteger</code> , <code>Long</code> , <code>ULong</code>)	<code>Double</code>
One expression is a <code>Single</code> data type and the other is not a <code>Double</code>	<code>Single</code>
One expression is a <code>Decimal</code> data type and the other is not a <code>Single</code> or a <code>Double</code>	<code>Decimal</code>
Either expression is a <code>Double</code> data type	<code>Double</code>

Before division is performed, any integral numeric expressions are widened to `Double`. If you assign the result to an integral data type, Visual Basic attempts to convert the result from `Double` to that type. This can throw an exception if the result does not fit in that type. In particular, see "Attempted Division by Zero" on this Help page.

If `expression1` or `expression2` evaluates to `Nothing`, it is treated as zero.

Attempted Division by Zero

If `expression2` evaluates to zero, the `/` operator behaves differently for different operand data types. The following table shows the possible behaviors.

OPERAND DATA TYPES	BEHAVIOR IF <code>EXPRESSION2</code> IS ZERO
Floating-point (<code>Single</code> or <code>Double</code>)	Returns infinity (<code>PositiveInfinity</code> or <code>NegativeInfinity</code>), or <code>NaN</code> (not a number) if <code>expression1</code> is also zero
<code>Decimal</code>	Throws <code>DivideByZeroException</code>
Integral (signed or unsigned)	Attempted conversion back to integral type throws <code>OverflowException</code> because integral types cannot accept <code>PositiveInfinity</code> , <code>NegativeInfinity</code> , or <code>NaN</code>

NOTE

The `/` 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](#).

Example

This example uses the `/` operator to perform floating-point division. The result is the quotient of the two operands.

```
Dim resultValue As Double  
resultValue = 10 / 4  
resultValue = 10 / 3
```

The expressions in the preceding example return values of 2.5 and 3.333333. Note that the result is always floating-point (`Double`), even though both operands are integer constants.

See also

- [/= Operator \(Visual Basic\)](#)
- [\ Operator \(Visual Basic\)](#)
- [Data Types of Operator Results](#)
- [Arithmetic Operators](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Arithmetic Operators in Visual Basic](#)

/= Operator (Visual Basic)

9/28/2019 • 2 minutes to read • [Edit Online](#)

Divides the value of a variable or property by the value of an expression and assigns the floating-point result to the variable or property.

Syntax

```
variableorproperty /= expression
```

Parts

`variableorproperty`

Required. Any numeric variable or property.

`expression`

Required. Any numeric expression.

Remarks

The element on the left side of the `/=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be [ReadOnly](#).

The `/=` operator first divides the value of the variable or property (on the left-hand side of the operator) by the value of the expression (on the right-hand side of the operator). The operator then assigns the floating-point result of that operation to the variable or property.

This statement assigns a `Double` value to the variable or property on the left. If `Option Strict` is `On`, `variableorproperty` must be a `Double`. If `Option Strict` is `Off`, Visual Basic performs an implicit conversion and assigns the resulting value to `variableorproperty`, with a possible error at run time. For more information, see [Widening and Narrowing Conversions](#) and [Option Strict Statement](#).

Overloading

The [/ Operator \(Visual Basic\)](#) 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. Overloading the `/` operator affects the behavior of the `/=` operator. If your code uses `/=` on a class or structure that overloads `/`, be sure you understand its redefined behavior. For more information, see [Operator Procedures](#).

Example

The following example uses the `/=` operator to divide one `Integer` variable by a second and assign the quotient to the first variable.

```
Dim var1 As Integer = 12
Dim var2 As Integer = 3
var1 /= var2
' The value of var1 is now 4.
```

See also

- [/ Operator \(Visual Basic\)](#)
- [\= Operator](#)
- [Assignment Operators](#)
- [Arithmetic Operators](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Statements](#)

\ Operator (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

Divides two numbers and returns an integer result.

Syntax

```
expression1 \ expression2
```

Parts

`expression1`

Required. Any numeric expression.

`expression2`

Required. Any numeric expression.

Supported Types

All numeric types, including the unsigned and floating-point types and `Decimal`.

Result

The result is the integer quotient of `expression1` divided by `expression2`, which discards any remainder and retains only the integer portion. This is known as *truncation*.

The result data type is a numeric type appropriate for the data types of `expression1` and `expression2`. See the "Integer Arithmetic" tables in [Data Types of Operator Results](#).

The [/ Operator \(Visual Basic\)](#) returns the full quotient, which retains the remainder in the fractional portion.

Remarks

Before performing the division, Visual Basic attempts to convert any floating-point numeric expression to `Long`. If `Option Strict` is `On`, a compiler error occurs. If `Option Strict` is `Off`, an [OverflowException](#) is possible if the value is outside the range of the [Long Data Type](#). The conversion to `Long` is also subject to *banker's rounding*. For more information, see "Fractional Parts" in [Type Conversion Functions](#).

If `expression1` or `expression2` evaluates to [Nothing](#), it is treated as zero.

Attempted Division by Zero

If `expression2` evaluates to zero, the `\` operator throws a [DivideByZeroException](#) exception. This is true for all numeric data types of the operands.

NOTE

The `\` 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](#).

Example

The following example uses the `\` operator to perform integer division. The result is an integer that represents the integer quotient of the two operands, with the remainder discarded.

```
Dim resultValue As Integer  
resultValue = 11 \ 4  
resultValue = 9 \ 3  
resultValue = 100 \ 3  
resultValue = 67 \ -3
```

The expressions in the preceding example return values of 2, 3, 33, and -22, respectively.

See also

- [\= Operator](#)
- [/ Operator \(Visual Basic\)](#)
- [Option Strict Statement](#)
- [Arithmetic Operators](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Arithmetic Operators in Visual Basic](#)

\= Operator

10/1/2019 • 2 minutes to read • [Edit Online](#)

Divides the value of a variable or property by the value of an expression and assigns the integer result to the variable or property.

Syntax

```
variableorproperty \= expression
```

Parts

`variableorproperty`

Required. Any numeric variable or property.

`expression`

Required. Any numeric expression.

Remarks

The element on the left side of the `\=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be [ReadOnly](#).

The `\=` operator divides the value of a variable or property on its left by the value on its right, and assigns the integer result to the variable or property on its left.

For further information on integer division, see [\ Operator \(Visual Basic\)](#).

Overloading

The `\` 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. Overloading the `\` operator affects the behavior of the `\=` operator. If your code uses `\=` on a class or structure that overloads `\`, be sure you understand its redefined behavior. For more information, see [Operator Procedures](#).

Example

The following example uses the `\=` operator to divide one `Integer` variable by a second and assign the integer result to the first variable.

```
Dim var1 As Integer = 10
Dim var2 As Integer = 3
var1 \= var2
' The value of var1 is now 3.
```

See also

- [\ Operator \(Visual Basic\)](#)
- [/= Operator \(Visual Basic\)](#)

- Assignment Operators
- Arithmetic Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Statements

`^` Operator (Visual Basic)

9/28/2019 • 2 minutes to read • [Edit Online](#)

Raises a number to the power of another number.

Syntax

```
number ^ exponent
```

Parts

`number`

Required. Any numeric expression.

`exponent`

Required. Any numeric expression.

Result

The result is `number` raised to the power of `exponent`, always as a `Double` value.

Supported Types

`Double`. Operands of any different type are converted to `Double`.

Remarks

Visual Basic always performs exponentiation in the [Double Data Type](#).

The value of `exponent` can be fractional, negative, or both.

When more than one exponentiation is performed in a single expression, the `^` operator is evaluated as it is encountered from left to right.

NOTE

The `^` 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](#).

Example

The following example uses the `^` operator to raise a number to the power of an exponent. The result is the first operand raised to the power of the second.

```
Dim exp1, exp2, exp3, exp4, exp5, exp6 As Double  
exp1 = 2 ^ 2  
exp2 = 3 ^ 3 ^ 3  
exp3 = (-5) ^ 3  
exp4 = (-5) ^ 4  
exp5 = 8 ^ (1.0 / 3.0)  
exp6 = 8 ^ (-1.0 / 3.0)
```

The preceding example produces the following results:

`exp1` is set to 4 (2 squared).

`exp2` is set to 19683 (3 cubed, then that value cubed).

`exp3` is set to -125 (-5 cubed).

`exp4` is set to 625 (-5 to the fourth power).

`exp5` is set to 2 (cube root of 8).

`exp6` is set to 0.5 (1.0 divided by the cube root of 8).

Note the importance of the parentheses in the expressions in the preceding example. Because of *operator precedence*, Visual Basic normally performs the `^` operator before any others, even the unary `-` operator. If `exp4` and `exp6` had been calculated without parentheses, they would have produced the following results:

`exp4 = -5 ^ 4` would be calculated as -(5 to the fourth power), which would result in -625.

`exp6 = 8 ^ -1.0 / 3.0` would be calculated as (8 to the -1 power, or 0.125) divided by 3.0, which would result in 0.04166666666666666666666666666667.

See also

- [^= Operator](#)
- [Arithmetic Operators](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Arithmetic Operators in Visual Basic](#)

`^=` Operator (Visual Basic)

9/28/2019 • 2 minutes to read • [Edit Online](#)

Raises the value of a variable or property to the power of an expression and assigns the result back to the variable or property.

Syntax

```
variableorproperty ^= expression
```

Parts

`variableorproperty`

Required. Any numeric variable or property.

`expression`

Required. Any numeric expression.

Remarks

The element on the left side of the `^=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be [ReadOnly](#).

The `^=` operator first raises the value of the variable or property (on the left-hand side of the operator) to the power of the value of the expression (on the right-hand side of the operator). The operator then assigns the result of that operation back to the variable or property.

Visual Basic always performs exponentiation in the [Double Data Type](#). Operands of any different type are converted to `Double`, and the result is always `Double`.

The value of `expression` can be fractional, negative, or both.

Overloading

The [^ 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. Overloading the `^` operator affects the behavior of the `^=` operator. If your code uses `^=` on a class or structure that overloads `^`, be sure you understand its redefined behavior. For more information, see [Operator Procedures](#).

Example

The following example uses the `^=` operator to raise the value of one `Integer` variable to the power of a second variable and assign the result to the first variable.

```
Dim var1 As Integer = 10
Dim var2 As Integer = 3
var1 ^= var2
' The value of var1 is now 1000.
```

See also

- [^ Operator](#)
- [Assignment Operators](#)
- [Arithmetic Operators](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Statements](#)

? . and ?() null-conditional operators (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Tests the value of the left-hand operand for null (`Nothing`) before performing a member access (`?.`) or index (`?()`) operation; returns `Nothing` if the left-hand operand evaluates to `Nothing`. Note that in expressions that ordinarily return value types, the null-conditional operator returns a `Nullable<T>`.

These operators help you write less code to handle null checks, especially when descending into data structures. For example:

```
' Nothing if customers is Nothing
Dim length As Integer? = customers?.Length

' Nothing if customers is Nothing
Dim first As Customer = customers?(0)

' Nothing if customers, the first customer, or Orders is Nothing
Dim count As Integer? = customers?(0)?.Orders?.Count()
```

For comparison, the alternative code for the first of these expressions without a null-conditional operator is:

```
Dim length As Integer
If customers IsNot Nothing Then
    length = customers.Length
End If
```

Sometimes you need to take an action on an object that may be null, based on the value of a Boolean member on that object (like the Boolean property `IsAllowedFreeShipping` in the following example):

```
Dim customer = FindCustomerByID(123) 'customer will be Nothing if not found.

If customer IsNot Nothing AndAlso customer.IsAllowedFreeShipping Then
    ApplyFreeShippingToOrders(customer)
End If
```

You can shorten your code and avoid manually checking for null by using the null-conditional operator as follows:

```
Dim customer = FindCustomerByID(123) 'customer will be Nothing if not found.

If customer?.IsAllowedFreeShipping Then ApplyFreeShippingToOrders(customer)
```

The null-conditional operators are short-circuiting. If one operation in a chain of conditional member access and index operations returns `Nothing`, the rest of the chain's execution stops. In the following example, `C(E)` isn't evaluated if `A`, `B`, or `C` evaluates to `Nothing`.

```
A?.B?.C?(E);
```

Another use for null-conditional member access is to invoke delegates in a thread-safe way with much less code. The following example defines two types, a `NewsBroadcaster` and a `NewsReceiver`. News items are sent to the receiver by the `NewsBroadcaster.SendNews` delegate.

```

Public Module NewsBroadcaster
    Dim SendNews As Action(Of String)

    Public Sub Main()
        Dim rec As New NewsReceiver()
        Dim rec2 As New NewsReceiver()
        SendNews?.Invoke("Just in: A newsworthy item...")
    End Sub

    Public Sub Register(client As Action(Of String))
        SendNews = SendNews.Combine({SendNews, client})
    End Sub
End Module

Public Class NewsReceiver
    Public Sub New()
        NewsBroadcaster.Register(AddressOf Me.DisplayNews)
    End Sub

    Public Sub DisplayNews(newsItem As String)
        Console.WriteLine(newsItem)
    End Sub
End Class

```

If there are no elements in the `SendNews` invocation list, the `SendNews` delegate throws a [NullReferenceException](#). Before null conditional operators, code like the following ensured that the delegate invocation list was not `Nothing`:

```

SendNews = SendNews.Combine({SendNews, client})
If SendNews IsNot Nothing Then
    SendNews("Just in...")
End If

```

The new way is much simpler:

```

SendNews = SendNews.Combine({SendNews, client})
SendNews?.Invoke("Just in...")

```

The new way is thread-safe because the compiler generates code to evaluate `SendNews` one time only, keeping the result in a temporary variable. You need to explicitly call the `Invoke` method because there is no null-conditional delegate invocation syntax `SendNews?(String)`.

See also

- [Operators \(Visual Basic\)](#)
- [Visual Basic Programming Guide](#)
- [Visual Basic Language Reference](#)

? . and ?() null-conditional operators (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Tests the value of the left-hand operand for null (`Nothing`) before performing a member access (`?.`) or index (`?()`) operation; returns `Nothing` if the left-hand operand evaluates to `Nothing`. Note that in expressions that ordinarily return value types, the null-conditional operator returns a `Nullable<T>`.

These operators help you write less code to handle null checks, especially when descending into data structures. For example:

```
' Nothing if customers is Nothing
Dim length As Integer? = customers?.Length

' Nothing if customers is Nothing
Dim first As Customer = customers?(0)

' Nothing if customers, the first customer, or Orders is Nothing
Dim count As Integer? = customers?(0)?.Orders?.Count()
```

For comparison, the alternative code for the first of these expressions without a null-conditional operator is:

```
Dim length As Integer
If customers IsNot Nothing Then
    length = customers.Length
End If
```

Sometimes you need to take an action on an object that may be null, based on the value of a Boolean member on that object (like the Boolean property `IsAllowedFreeShipping` in the following example):

```
Dim customer = FindCustomerByID(123) 'customer will be Nothing if not found.

If customer IsNot Nothing AndAlso customer.IsAllowedFreeShipping Then
    ApplyFreeShippingToOrders(customer)
End If
```

You can shorten your code and avoid manually checking for null by using the null-conditional operator as follows:

```
Dim customer = FindCustomerByID(123) 'customer will be Nothing if not found.

If customer?.IsAllowedFreeShipping Then ApplyFreeShippingToOrders(customer)
```

The null-conditional operators are short-circuiting. If one operation in a chain of conditional member access and index operations returns `Nothing`, the rest of the chain's execution stops. In the following example, `C(E)` isn't evaluated if `A`, `B`, or `C` evaluates to `Nothing`.

```
A?.B?.C?(E);
```

Another use for null-conditional member access is to invoke delegates in a thread-safe way with much less code. The following example defines two types, a `NewsBroadcaster` and a `NewsReceiver`. News items are sent to the receiver by the `NewsBroadcaster.SendNews` delegate.

```

Public Module NewsBroadcaster
    Dim SendNews As Action(Of String)

    Public Sub Main()
        Dim rec As New NewsReceiver()
        Dim rec2 As New NewsReceiver()
        SendNews?.Invoke("Just in: A newsworthy item...")
    End Sub

    Public Sub Register(client As Action(Of String))
        SendNews = SendNews.Combine({SendNews, client})
    End Sub
End Module

Public Class NewsReceiver
    Public Sub New()
        NewsBroadcaster.Register(AddressOf Me.DisplayNews)
    End Sub

    Public Sub DisplayNews(newsItem As String)
        Console.WriteLine(newsItem)
    End Sub
End Class

```

If there are no elements in the `SendNews` invocation list, the `SendNews` delegate throws a [NullReferenceException](#).

Before null conditional operators, code like the following ensured that the delegate invocation list was not `Nothing`:

```

SendNews = SendNews.Combine({SendNews, client})
If SendNews IsNot Nothing Then
    SendNews("Just in...")
End If

```

The new way is much simpler:

```

SendNews = SendNews.Combine({SendNews, client})
SendNews?.Invoke("Just in...")

```

The new way is thread-safe because the compiler generates code to evaluate `SendNews` one time only, keeping the result in a temporary variable. You need to explicitly call the `Invoke` method because there is no null-conditional delegate invocation syntax `SendNews?(String)`.

See also

- [Operators \(Visual Basic\)](#)
- [Visual Basic Programming Guide](#)
- [Visual Basic Language Reference](#)

AddressOf Operator (Visual Basic)

9/28/2019 • 2 minutes to read • [Edit Online](#)

Creates a delegate instance that references the specific procedure.

Syntax

```
AddressOf procedurename
```

Parts

`procedurename`

Required. Specifies the procedure to be referenced by the newly created delegate.

Remarks

The `AddressOf` operator creates a delegate that points to the sub or function specified by `procedurename`. When the specified procedure is an instance method then the delegate refers to both the instance and the method. Then, when the delegate is invoked the specified method of the specified instance is called.

The `AddressOf` operator can be used as the operand of a delegate constructor or it can be used in a context in which the type of the delegate can be determined by the compiler.

Example

This example uses the `AddressOf` operator to designate a delegate to handle the `Click` event of a button.

```
' Add the following line to Sub Form1_Load().
AddHandler Button1.Click, AddressOf Button1_Click
```

Example

The following example uses the `AddressOf` operator to designate the startup function for a thread.

```
Public Sub CountSheep()
    Dim i As Integer = 1 ' Sheep do not count from 0.
    Do While (True) ' Endless loop.
        Console.WriteLine("Sheep " & i & " Baah")
        i = i + 1
        System.Threading.Thread.Sleep(1000) 'Wait 1 second.
    Loop
End Sub

Sub UseThread()
    Dim t As New System.Threading.Thread(AddressOf CountSheep)
    t.Start()
End Sub
```

See also

- Declare Statement
- Function Statement
- Sub Statement
- Delegates

And Operator (Visual Basic)

9/28/2019 • 2 minutes to read • [Edit Online](#)

Performs a logical conjunction on two `Boolean` expressions, or a bitwise conjunction on two numeric expressions.

Syntax

```
result = expression1 And expression2
```

Parts

`result`

Required. Any `Boolean` or numeric expression. For Boolean comparison, `result` is the logical conjunction of two `Boolean` values. For bitwise operations, `result` is a numeric value representing the bitwise conjunction of two numeric bit patterns.

`expression1`

Required. Any `Boolean` or numeric expression.

`expression2`

Required. Any `Boolean` or numeric expression.

Remarks

For Boolean comparison, `result` is `True` if and only if both `expression1` and `expression2` evaluate to `True`. The following table illustrates how `result` is determined.

IF <code>EXPRESSION1</code> IS	AND <code>EXPRESSION2</code> IS	THE VALUE OF <code>RESULT</code> IS
<code>True</code>	<code>True</code>	<code>True</code>
<code>True</code>	<code>False</code>	<code>False</code>
<code>False</code>	<code>True</code>	<code>False</code>
<code>False</code>	<code>False</code>	<code>False</code>

NOTE

In a Boolean comparison, the `And` operator always evaluates both expressions, which could include making procedure calls. The [AndAlso Operator](#) performs *short-circuiting*, which means that if `expression1` is `False`, then `expression2` is not evaluated.

When applied to numeric values, the `And` operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in `result` according to the following table.

IF BIT IN <code>EXPRESSION1</code> IS	AND BIT IN <code>EXPRESSION2</code> IS	THE BIT IN <code>RESULT</code> IS
1	1	1
1	0	0
0	1	0
0	0	0

NOTE

Since the logical and bitwise operators have a lower precedence than other arithmetic and relational operators, any bitwise operations should be enclosed in parentheses to ensure accurate results.

Data Types

If the operands consist of one `Boolean` expression and one numeric expression, Visual Basic converts the `Boolean` expression to a numeric value (–1 for `True` and 0 for `False`) and performs a bitwise operation.

For a Boolean comparison, the data type of the result is `Boolean`. For a bitwise comparison, the result data type is a numeric type appropriate for the data types of `expression1` and `expression2`. See the "Relational and Bitwise Comparisons" table in [Data Types of Operator Results](#).

NOTE

The `And` 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](#).

Example

The following example uses the `And` operator to perform a logical conjunction on two expressions. The result is a `Boolean` value that represents whether both of the expressions are `True`.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstCheck, secondCheck As Boolean
firstCheck = a > b And b > c
secondCheck = b > a And b > c
```

The preceding example produces results of `True` and `False`, respectively.

Example

The following example uses the `And` operator to perform logical conjunction on the individual bits of two numeric expressions. The bit in the result pattern is set if the corresponding bits in the operands are both set to 1.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstPattern, secondPattern, thirdPattern As Integer
firstPattern = (a And b)
secondPattern = (a And c)
thirdPattern = (b And c)
```

The preceding example produces results of 8, 2, and 0, respectively.

See also

- [Logical/Bitwise Operators \(Visual Basic\)](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [AndAlso Operator](#)
- [Logical and Bitwise Operators in Visual Basic](#)

AndAlso Operator (Visual Basic)

10/3/2019 • 2 minutes to read • [Edit Online](#)

Performs short-circuiting logical conjunction on two expressions.

Syntax

```
result = expression1 AndAlso expression2
```

Parts

TERM	DEFINITION
result	Required. Any <code>Boolean</code> expression. The result is the <code>Boolean</code> result of comparison of the two expressions.
expression1	Required. Any <code>Boolean</code> expression.
expression2	Required. Any <code>Boolean</code> expression.

Remarks

A logical operation is said to be *short-circuiting* if the compiled code can bypass the evaluation of one expression depending on the result of another expression. If the result of the first expression evaluated determines the final result of the operation, there is no need to evaluate the second expression, because it cannot change the final result. Short-circuiting can improve performance if the bypassed expression is complex, or if it involves procedure calls.

If both expressions evaluate to `True`, `result` is `True`. The following table illustrates how `result` is determined.

IF <code>EXPRESSION1</code> IS	AND <code>EXPRESSION2</code> IS	THE VALUE OF <code>RESULT</code> IS
<code>True</code>	<code>True</code>	<code>True</code>
<code>True</code>	<code>False</code>	<code>False</code>
<code>False</code>	(not evaluated)	<code>False</code>

Data Types

The `AndAlso` operator is defined only for the [Boolean Data Type](#). Visual Basic converts each operand as necessary to `Boolean` before evaluating the expression. If you assign the result to a numeric type, Visual Basic converts it from `Boolean` to that type such that `False` becomes `0` and `True` becomes `-1`. For more information, see [Boolean Type Conversions](#).

Overloading

The [And Operator](#) and the [IsFalse Operator](#) 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. Overloading the `And` and `IsFalse` operators affects the behavior of the `AndAlso` operator. If your code uses `AndAlso` on a class or structure that overloads `And` and `IsFalse`, be sure you understand their redefined behavior. For more information, see [Operator Procedures](#).

Example

The following example uses the `AndAlso` operator to perform a logical conjunction on two expressions. The result is a `Boolean` value that represents whether the entire conjoined expression is true. If the first expression is `False`, the second is not evaluated.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstCheck, secondCheck, thirdCheck As Boolean
firstCheck = a > b AndAlso b > c
secondCheck = b > a AndAlso b > c
thirdCheck = a > b AndAlso c > b
```

The preceding example produces results of `True`, `False`, and `False`, respectively. In the calculation of `secondCheck`, the second expression is not evaluated because the first is already `False`. However, the second expression is evaluated in the calculation of `thirdCheck`.

Example

The following example shows a `Function` procedure that searches for a given value among the elements of an array. If the array is empty, or if the array length has been exceeded, the `While` statement does not test the array element against the search value.

```
Public Function findValue(ByVal arr() As Double,
    ByVal searchValue As Double) As Double
    Dim i As Integer = 0
    While i <= UBound(arr) AndAlso arr(i) <> searchValue
        ' If i is greater than UBound(arr), searchValue is not checked.
        i += 1
    End While
    If i > UBound(arr) Then i = -1
    Return i
End Function
```

See also

- [Logical/Bitwise Operators \(Visual Basic\)](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [And Operator](#)
- [IsFalse Operator](#)
- [Logical and Bitwise Operators in Visual Basic](#)

Await Operator (Visual Basic)

10/18/2019 • 4 minutes to read • [Edit Online](#)

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](#).

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<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.

```
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](#). 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.

```
' 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<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](#).

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.

```

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](#)
- [Walkthrough: Accessing the Web by Using Async and Await](#)
- [Async](#)

DirectCast Operator (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Introduces a type conversion operation based on inheritance or implementation.

Remarks

`DirectCast` does not use the Visual Basic run-time helper routines for conversion, so it can provide somewhat better performance than `CType` when converting to and from data type `Object`.

You use the `DirectCast` keyword similar to the way you use the [CType Function](#) and the [TryCast Operator](#) keyword. You supply an expression as the first argument and a type to convert it to as the second argument.

`DirectCast` requires an inheritance or implementation relationship between the data types of the two arguments. This means that one type must inherit from or implement the other.

Errors and Failures

`DirectCast` generates a compiler error if it detects that no inheritance or implementation relationship exists. But the lack of a compiler error does not guarantee a successful conversion. If the desired conversion is narrowing, it could fail at run time. If this happens, the runtime throws an [InvalidOperationException](#) error.

Conversion Keywords

A comparison of the type conversion keywords is as follows.

KEYWORD	DATA TYPES	ARGUMENT RELATIONSHIP	RUN-TIME FAILURE
CType Function	Any data types	Widening or narrowing conversion must be defined between the two data types	Throws InvalidOperationException
<code>DirectCast</code>	Any data types	One type must inherit from or implement the other type	Throws InvalidOperationException
TryCast Operator	Reference types only	One type must inherit from or implement the other type	Returns Nothing

Example

The following example demonstrates two uses of `DirectCast`, one that fails at run time and one that succeeds.

```
Dim q As Object = 2.37
Dim i As Integer = CType(q, Integer)
' The following conversion fails at run time
Dim j As Integer = DirectCast(q, Integer)
Dim f As New System.Windows.Forms.Form
Dim c As System.Windows.Forms.Control
' The following conversion succeeds.
c = DirectCast(f, System.Windows.Forms.Control)
```

In the preceding example, the run-time type of `q` is `Double`. `CType` succeeds because `Double` can be converted

to `Integer`. However, the first `DirectCast` fails at run time because the run-time type of `Double` has no inheritance relationship with `Integer`, even though a conversion exists. The second `DirectCast` succeeds because it converts from type `Form` to type `Control`, from which `Form` inherits.

See also

- [Convert.ChangeType](#)
- [Widening and Narrowing Conversions](#)
- [Implicit and Explicit Conversions](#)

Function Expression (Visual Basic)

9/28/2019 • 2 minutes to read • [Edit Online](#)

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 .
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 <code>Return</code> statement. (See Return Statement .) 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](#).

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.

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

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

Example

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

```
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](#).

```
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.

```
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](#). For more information about standard query operators, see [Standard Query Operators Overview](#).

See also

- [Function Statement](#)
- [Lambda Expressions](#)
- [Operators and Expressions](#)
- [Statements](#)
- [Value Comparisons](#)
- [Boolean Expressions](#)
- [If Operator](#)
- [Relaxed Delegate Conversion](#)

GetType Operator (Visual Basic)

9/28/2019 • 2 minutes to read • [Edit Online](#)

Returns a [Type](#) object for the specified type. The [Type](#) object provides information about the type such as its properties, methods, and events.

Syntax

```
GetType(typename)
```

Parameters

PARAMETER	DESCRIPTION
<code>typename</code>	The name of the type for which you desire information.

Remarks

The `GetType` operator returns the [Type](#) object for the specified `typename`. You can pass the name of any defined type in `typename`. This includes the following:

- Any Visual Basic data type, such as `Boolean` or `Date`.
- Any .NET Framework class, structure, module, or interface, such as [System.ArgumentException](#) or [System.Double](#).
- Any class, structure, module, or interface defined by your application.
- Any array defined by your application.
- Any delegate defined by your application.
- Any enumeration defined by Visual Basic, the .NET Framework, or your application.

If you want to get the type object of an object variable, use the [Type.GetType](#) method.

The `GetType` operator can be useful in the following circumstances:

- You must access the metadata for a type at run time. The [Type](#) object supplies metadata such as type members and deployment information. You need this, for example, to reflect over an assembly. For more information, see [System.Reflection](#).
- You want to compare two object references to see if they refer to instances of the same type. If they do, `GetType` returns references to the same [Type](#) object.

Example

The following examples show the `GetType` operator in use.

```
' The following statement returns the Type object for Integer.  
MsgBox(GetType(Integer).ToString())  
' The following statement returns the Type object for one-dimensional string arrays.  
MsgBox(GetType(String()).ToString())
```

See also

- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Operators and Expressions](#)

GetXmlNamespace Operator (Visual Basic)

9/28/2019 • 2 minutes to read • [Edit Online](#)

Gets the [XNamespace](#) object that corresponds to the specified XML namespace prefix.

Syntax

```
GetXmlNamespace(xmlNamespacePrefix)
```

Parts

`xmlNamespacePrefix`

Optional. The string that identifies the XML namespace prefix. If supplied, this string must be a valid XML identifier. For more information, see [Names of Declared XML Elements and Attributes](#). If no prefix is specified, the default namespace is returned. If no default namespace is specified, the empty namespace is returned.

Return Value

The [XNamespace](#) object that corresponds to the XML namespace prefix.

Remarks

The `GetXmlNamespace` operator gets the [XNamespace](#) object that corresponds to the XML namespace prefix `xmlNamespacePrefix`.

You can use XML namespace prefixes directly in XML literals and XML axis properties. However, you must use the `GetXmlNamespace` operator to convert a namespace prefix to an [XNamespace](#) object before you can use it in your code. You can append an unqualified element name to an [XNamespace](#) object to get a fully qualified [XName](#) object, which many LINQ to XML methods require.

Example

The following example imports `ns` as an XML namespace prefix. It then uses the prefix of the namespace to create an XML literal and access the first child node that has the qualified name `ns:phone`. It then passes that child node to the `ShowName` subroutine, which constructs a qualified name by using the `GetXmlNamespace` operator. The `ShowName` subroutine then passes the qualified name to the `Ancestors` method to get the parent `ns:contact` node.

```
' Place Imports statements at the top of your program.  
Imports <xmlns:ns="http://SomeNamespace">  
  
Module GetXmlNamespaceSample  
  
Sub RunSample()  
  
    ' Create test by using a global XML namespace prefix.  
  
    Dim contact =  
        <ns:contact>  
            <ns:name>Patrick Hines</ns:name>  
            <ns:phone ns:type="home">206-555-0144</ns:phone>  
            <ns:phone ns:type="work">425-555-0145</ns:phone>  
        </ns:contact>  
  
    ShowName(contact.<ns:phone>(0))  
End Sub  
  
Sub ShowName(ByVal phone As XElement)  
    Dim qualifiedName = GetXmlNamespace(ns) + "contact"  
    Dim contact = phone.Ancestors(qualifiedName)(0)  
    Console.WriteLine("Name: " & contact.<ns:name>.Value)  
End Sub  
  
End Module
```

When you call `TestGetXmlNamespace.RunSample()`, it displays a message box that contains the following text:

Name: Patrick Hines

See also

- [Imports Statement \(XML Namespace\)](#)
- [Accessing XML in Visual Basic](#)

If Operator (Visual Basic)

10/1/2019 • 3 minutes to read • [Edit Online](#)

Uses short-circuit evaluation to conditionally return one of two values. The `If` operator can be called with three arguments or with two arguments.

Syntax

```
If( [argument1,] argument2, argument3 )
```

If Operator Called with Three Arguments

When `If` is called by using three arguments, the first argument must evaluate to a value that can be cast as a `Boolean`. That `Boolean` value will determine which of the other two arguments is evaluated and returned. The following list applies only when the `If` operator is called by using three arguments.

Parts

TERM	DEFINITION
<code>argument1</code>	Required. <code>Boolean</code> . Determines which of the other arguments to evaluate and return.
<code>argument2</code>	Required. <code>Object</code> . Evaluated and returned if <code>argument1</code> evaluates to <code>True</code> .
<code>argument3</code>	Required. <code>Object</code> . Evaluated and returned if <code>argument1</code> evaluates to <code>False</code> or if <code>argument1</code> is a <code>Nullable Boolean</code> variable that evaluates to <code>Nothing</code> .

An `If` operator that is called with three arguments works like an `IIf` function except that it uses short-circuit evaluation. An `IIf` function always evaluates all three of its arguments, whereas an `If` operator that has three arguments evaluates only two of them. The first `If` argument is evaluated and the result is cast as a `Boolean` value, `True` or `False`. If the value is `True`, `argument2` is evaluated and its value is returned, but `argument3` is not evaluated. If the value of the `Boolean` expression is `False`, `argument3` is evaluated and its value is returned, but `argument2` is not evaluated. The following examples illustrate the use of `If` when three arguments are used:

```

' This statement prints TruePart, because the first argument is true.
Console.WriteLine(If(True, "TruePart", "FalsePart"))

' This statement prints FalsePart, because the first argument is false.
Console.WriteLine(If(False, "TruePart", "FalsePart"))

Dim number = 3
' With number set to 3, this statement prints Positive.
Console.WriteLine(If(number >= 0, "Positive", "Negative"))

number = -1
' With number set to -1, this statement prints Negative.
Console.WriteLine(If(number >= 0, "Positive", "Negative"))

```

The following example illustrates the value of short-circuit evaluation. The example shows two attempts to divide variable `number` by variable `divisor` except when `divisor` is zero. In that case, a 0 should be returned, and no attempt should be made to perform the division because a run-time error would result. Because the `If` expression uses short-circuit evaluation, it evaluates either the second or the third argument, depending on the value of the first argument. If the first argument is true, the divisor is not zero and it is safe to evaluate the second argument and perform the division. If the first argument is false, only the third argument is evaluated and a 0 is returned. Therefore, when the divisor is 0, no attempt is made to perform the division and no error results. However, because `IIf` does not use short-circuit evaluation, the second argument is evaluated even when the first argument is false. This causes a run-time divide-by-zero error.

```

number = 12

' When the divisor is not 0, both If and IIf return 4.
Dim divisor = 3
Console.WriteLine(If(divisor <> 0, number \ divisor, 0))
Console.WriteLine(IIf(divisor <> 0, number \ divisor, 0))

' When the divisor is 0, IIf causes a run-time error, but If does not.
divisor = 0
Console.WriteLine(If(divisor <> 0, number \ divisor, 0))
' Console.WriteLine(IIf(divisor <> 0, number \ divisor, 0))

```

If Operator Called with Two Arguments

The first argument to `If` can be omitted. This enables the operator to be called by using only two arguments. The following list applies only when the `If` operator is called with two arguments.

Parts

TERM	DEFINITION
<code>argument2</code>	Required. <code>Object</code> . Must be a reference or nullable type. Evaluated and returned when it evaluates to anything other than <code>Nothing</code> .
<code>argument3</code>	Required. <code>Object</code> . Evaluated and returned if <code>argument2</code> evaluates to <code>Nothing</code> .

When the `Boolean` argument is omitted, the first argument must be a reference or nullable type. If the first argument evaluates to `Nothing`, the value of the second argument is returned. In all other cases, the value of the first argument is returned. The following example illustrates how this evaluation works.

```
' Variable first is a nullable type.  
Dim first? As Integer = 3  
Dim second As Integer = 6  
  
' Variable first <> Nothing, so its value, 3, is returned.  
Console.WriteLine(If(first, second))  
  
second = Nothing  
' Variable first <> Nothing, so the value of first is returned again.  
Console.WriteLine(If(first, second))  
  
first = Nothing  
second = 6  
' Variable first = Nothing, so 6 is returned.  
Console.WriteLine(If(first, second))
```

See also

- [IIf](#)
- [Nullable Value Types](#)
- [Nothing](#)

Is Operator (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

Compares two object reference variables.

Syntax

```
result = object1 Is object2
```

Parts

`result`

Required. Any `Boolean` value.

`object1`

Required. Any `Object` name.

`object2`

Required. Any `Object` name.

Remarks

The `Is` operator determines if two object references refer to the same object. However, it does not perform value comparisons. If `object1` and `object2` both refer to the exact same object instance, `result` is `True`; if they do not, `result` is `False`.

`Is` can also be used with the `TypeOf` keyword to make a `TypeOf ... Is` expression, which tests whether an object variable is compatible with a data type.

NOTE

The `Is` keyword is also used in the `Select...Case Statement`.

Example

The following example uses the `Is` operator to compare pairs of object references. The results are assigned to a `Boolean` value representing whether the two objects are identical.

```
Dim myObject As New Object
Dim otherObject As New Object
Dim yourObject, thisObject, thatObject As Object
Dim myCheck As Boolean
yourObject = myObject
thisObject = myObject
thatObject = otherObject
' The following statement sets myCheck to True.
myCheck = yourObject Is thisObject
' The following statement sets myCheck to False.
myCheck = thatObject Is thisObject
' The following statement sets myCheck to False.
myCheck = myObject Is thatObject
thatObject = myObject
' The following statement sets myCheck to True.
myCheck = thisObject Is thatObject
```

As the preceding example demonstrates, you can use the `Is` operator to test both early bound and late bound objects.

See also

- [TypeOf Operator](#)
- [IsNot Operator](#)
- [Comparison Operators in Visual Basic](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Operators and Expressions](#)

IsFalse Operator (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

Determines whether an expression is `False`.

You cannot call `IsFalse` explicitly in your code, but the Visual Basic compiler can use it to generate code from `AndAlso` clauses. If you define a class or structure and then use a variable of that type in an `AndAlso` clause, you must define `IsFalse` on that class or structure.

The compiler considers the `IsFalse` and `IsTrue` operators as a *matched pair*. This means that if you define one of them, you must also define the other one.

NOTE

The `IsFalse` operator can be *overloaded*, which means that a class or structure can redefine its behavior when its 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](#).

Example

The following code example defines the outline of a structure that includes definitions for the `IsFalse` and `IsTrue` operators.

```
Public Structure p
    Dim a As Double
    Public Shared Operator IsFalse(ByVal w As p) As Boolean
        Dim b As Boolean
        ' Insert code to calculate IsFalse of w.
        Return b
    End Operator
    Public Shared Operator IsTrue(ByVal w As p) As Boolean
        Dim b As Boolean
        ' Insert code to calculate IsTrue of w.
        Return b
    End Operator
End Structure
```

See also

- [IsTrue Operator](#)
- [How to: Define an Operator](#)
- [AndAlso Operator](#)

IsNot Operator (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

Compares two object reference variables.

Syntax

```
result = object1 IsNot object2
```

Parts

`result` Required. A `Boolean` value.

`object1` Required. Any `Object` variable or expression.

`object2` Required. Any `Object` variable or expression.

Remarks

The `IsNot` operator determines if two object references refer to different objects. However, it does not perform value comparisons. If `object1` and `object2` both refer to the exact same object instance, `result` is `False`; if they do not, `result` is `True`.

`IsNot` is the opposite of the `Is` operator. The advantage of `IsNot` is that you can avoid awkward syntax with `Not` and `Is`, which can be difficult to read.

You can use the `Is` and `IsNot` operators to test both early-bound and late-bound objects.

Example

The following code example uses both the `Is` operator and the `IsNot` operator to accomplish the same comparison.

```
Dim o1, o2 As New Object
If Not o1 Is o2 Then MsgBox("o1 and o2 do not refer to the same instance.")
If o1 IsNot o2 Then MsgBox("o1 and o2 do not refer to the same instance.")
```

See also

- [Is Operator](#)
- [TypeOf Operator](#)
- [Operator Precedence in Visual Basic](#)
- [How to: Test Whether Two Objects Are the Same](#)

IsTrue Operator (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

Determines whether an expression is `True`.

You cannot call `IsTrue` explicitly in your code, but the Visual Basic compiler can use it to generate code from `OrElse` clauses. If you define a class or structure and then use a variable of that type in an `OrElse` clause, you must define `IsTrue` on that class or structure.

The compiler considers the `IsTrue` and `IsFalse` operators as a *matched pair*. This means that if you define one of them, you must also define the other one.

Compiler Use of IsTrue

When you have defined a class or structure, you can use a variable of that type in a `For`, `If`, `Else If`, or `While` statement, or in a `When` clause. If you do this, the compiler requires an operator that converts your type into a `Boolean` value so it can test a condition. It searches for a suitable operator in the following order:

1. A widening conversion operator from your class or structure to `Boolean`.
2. A widening conversion operator from your class or structure to `Boolean?`.
3. The `IsTrue` operator on your class or structure.
4. A narrowing conversion to `Boolean?` that does not involve a conversion from `Boolean` to `Boolean?`.
5. A narrowing conversion operator from your class or structure to `Boolean`.

If you have not defined any conversion to `Boolean` or an `IsTrue` operator, the compiler signals an error.

NOTE

The `IsTrue` operator can be *overloaded*, which means that a class or structure can redefine its behavior when its 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](#).

Example

The following code example defines the outline of a structure that includes definitions for the `IsFalse` and `IsTrue` operators.

```
Public Structure p
    Dim a As Double
    Public Shared Operator IsFalse(ByVal w As p) As Boolean
        Dim b As Boolean
        ' Insert code to calculate IsFalse of w.
        Return b
    End Operator
    Public Shared Operator IsTrue(ByVal w As p) As Boolean
        Dim b As Boolean
        ' Insert code to calculate IsTrue of w.
        Return b
    End Operator
End Structure
```

See also

- [IsFalse Operator](#)
- [How to: Define an Operator](#)
- [OrElse Operator](#)

Like Operator (Visual Basic)

10/1/2019 • 5 minutes to read • [Edit Online](#)

Compares a string against a pattern.

IMPORTANT

The `Like` operator is currently not supported in .NET Core and .NET Standard projects.

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

CHARACTERS IN PATTERN	MATCHES IN STRING
#	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](#).

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 È 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](#).

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.

```
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 "aBBBBa" have an "a" at the  
' beginning, an "a" at the end, and any number of characters in  
' between?)  
testCheck = "aBBBBa" 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](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Option Compare Statement](#)
- [Operators and Expressions](#)
- [How to: Match a String against a Pattern](#)

Mod operator (Visual Basic)

9/12/2019 • 3 minutes to read • [Edit Online](#)

Divides two numbers and returns only the remainder.

Syntax

```
result = number1 Mod number2
```

Parts

`result`

Required. Any numeric variable or property.

`number1`

Required. Any numeric expression.

`number2`

Required. Any numeric expression.

Supported types

All numeric types. This includes the unsigned and floating-point types and `Decimal`.

Result

The result is the remainder after `number1` is divided by `number2`. For example, the expression `14 Mod 4` evaluates to 2.

NOTE

There is a difference between *remainder* and *modulus* in mathematics, with different results for negative numbers. The `Mod` operator in Visual Basic, the .NET Framework `op_Modulus` operator, and the underlying `rem` IL instruction all perform a remainder operation.

The result of a `Mod` operation retains the sign of the dividend, `number1`, and so it may be positive or negative. The result is always in the range $(-\text{number2}, \text{number2})$, exclusive. For example:

```

Public Module Example
    Public Sub Main()
        Console.WriteLine($" 8 Mod 3 = {8 Mod 3}")
        Console.WriteLine($"-8 Mod 3 = {-8 Mod 3}")
        Console.WriteLine($" 8 Mod -3 = {8 Mod -3}")
        Console.WriteLine($"-8 Mod -3 = {-8 Mod -3}")
    End Sub
End Module
' The example displays the following output:
'     8 Mod 3 = 2
'    -8 Mod 3 = -2
'     8 Mod -3 = 2
'    -8 Mod -3 = -2

```

Remarks

If either `number1` or `number2` is a floating-point value, the floating-point remainder of the division is returned. The data type of the result is the smallest data type that can hold all possible values that result from division with the data types of `number1` and `number2`.

If `number1` or `number2` evaluates to [Nothing](#), it is treated as zero.

Related operators include the following:

- The [\ Operator \(Visual Basic\)](#) returns the integer quotient of a division. For example, the expression `14 \ 4` evaluates to 3.
- The [/ Operator \(Visual Basic\)](#) returns the full quotient, including the remainder, as a floating-point number. For example, the expression `14 / 4` evaluates to 3.5.

Attempted division by zero

If `number2` evaluates to zero, the behavior of the `Mod` operator depends on the data type of the operands:

- An integral division throws a [DivideByZeroException](#) exception if `number2` cannot be determined in compile-time and generates a compile-time error `BC30542 Division by zero occurred while evaluating this expression` if `number2` is evaluated to zero at compile-time.
- A floating-point division returns [Double.NaN](#).

Equivalent formula

The expression `a Mod b` is equivalent to either of the following formulas:

$$a - (b * (a \ b))$$

$$a - (b * \text{Fix}(a / b))$$

Floating-point imprecision

When you work with floating-point numbers, remember that they do not always have a precise decimal representation in memory. This can lead to unexpected results from certain operations, such as value comparison and the `Mod` operator. For more information, see [Troubleshooting Data Types](#).

Overloading

The `Mod` operator can be *overloaded*, which means that a class or structure can redefine its behavior. If your code applies `Mod` to an instance of a class or structure that includes such an overload, be sure you understand its

redefined behavior. For more information, see [Operator Procedures](#).

Example

The following example uses the `Mod` operator to divide two numbers and return only the remainder. If either number is a floating-point number, the result is a floating-point number that represents the remainder.

```
Debug.WriteLine(10 Mod 5)
' Output: 0
Debug.WriteLine(10 Mod 3)
' Output: 1
Debug.WriteLine(-10 Mod 3)
' Output: -1
Debug.WriteLine(12 Mod 4.3)
' Output: 3.4
Debug.WriteLine(12.6 Mod 5)
' Output: 2.6
Debug.WriteLine(47.9 Mod 9.35)
' Output: 1.15
```

Example

The following example demonstrates the potential imprecision of floating-point operands. In the first statement, the operands are `Double`, and 0.2 is an infinitely repeating binary fraction with a stored value of 0.20000000000000001. In the second statement, the literal type character `D` forces both operands to `Decimal`, and 0.2 has a precise representation.

```
firstResult = 2.0 Mod 0.2
' Double operation returns 0.2, not 0.
secondResult = 2D Mod 0.2D
' Decimal operation returns 0.
```

See also

- [Int](#)
- [Fix](#)
- [Arithmetic Operators](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Troubleshooting Data Types](#)
- [Arithmetic Operators in Visual Basic](#)
- [\ Operator \(Visual Basic\)](#)

New Operator (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

Introduces a `New` clause to create a new object instance, specifies a constructor constraint on a type parameter, or identifies a `Sub` procedure as a class constructor.

Remarks

In a declaration or assignment statement, a `New` clause must specify a defined class from which the instance can be created. This means that the class must expose one or more constructors that the calling code can access.

You can use a `New` clause in a declaration statement or an assignment statement. When the statement runs, it calls the appropriate constructor of the specified class, passing any arguments you have supplied. The following example demonstrates this by creating instances of a `Customer` class that has two constructors, one that takes no parameters and one that takes a string parameter.

```
' For customer1, call the constructor that takes no arguments.  
Dim customer1 As New Customer()  
  
' For customer2, call the constructor that takes the name of the  
' customer as an argument.  
Dim customer2 As New Customer("Blue Yonder Airlines")  
  
' For customer3, declare an instance of Customer in the first line  
' and instantiate it in the second.  
Dim customer3 As Customer  
customer3 = New Customer()  
  
' With Option Infer set to On, the following declaration declares  
' and instantiates a new instance of Customer.  
Dim customer4 = New Customer("Coho Winery")
```

Since arrays are classes, `New` can create a new array instance, as shown in the following examples.

```
Dim intArray1() As Integer  
intArray1 = New Integer() {1, 2, 3, 4}  
  
Dim intArray2() As Integer = {5, 6}  
  
' The following example requires that Option Infer be set to On.  
Dim intArray3() = New Integer() {6, 7, 8}
```

The common language runtime (CLR) throws an [OutOfMemoryException](#) error if there is insufficient memory to create the new instance.

NOTE

The `New` keyword is also used in type parameter lists to specify that the supplied type must expose an accessible parameterless constructor. For more information about type parameters and constraints, see [Type List](#).

To create a constructor procedure for a class, set the name of a `Sub` procedure to the `New` keyword. For more information, see [Object Lifetime: How Objects Are Created and Destroyed](#).

The `New` keyword can be used in these contexts:

[Dim Statement](#)

[Of](#)

[Sub Statement](#)

See also

- [OutOfMemoryException](#)
- [Keywords](#)
- [Type List](#)
- [Generic Types in Visual Basic](#)
- [Object Lifetime: How Objects Are Created and Destroyed](#)

Not Operator (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

Performs logical negation on a `Boolean` expression, or bitwise negation on a numeric expression.

Syntax

```
result = Not expression
```

Parts

`result`

Required. Any `Boolean` or numeric expression.

`expression`

Required. Any `Boolean` or numeric expression.

Remarks

For `Boolean` expressions, the following table illustrates how `result` is determined.

IF <code>EXPRESSION</code> IS	THE VALUE OF <code>RESULT</code> IS
<code>True</code>	<code>False</code>
<code>False</code>	<code>True</code>

For numeric expressions, the `Not` operator inverts the bit values of any numeric expression and sets the corresponding bit in `result` according to the following table.

IF BIT IN <code>EXPRESSION</code> IS	THE BIT IN <code>RESULT</code> IS
1	0
0	1

NOTE

Since the logical and bitwise operators have a lower precedence than other arithmetic and relational operators, any bitwise operations should be enclosed in parentheses to ensure accurate execution.

Data Types

For a Boolean negation, the data type of the result is `Boolean`. For a bitwise negation, the result data type is the same as that of `expression`. However, if `expression` is `Decimal`, the result is `Long`.

Overloading

The `Not` operator can be *overloaded*, which means that a class or structure can redefine its behavior when its 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](#).

Example

The following example uses the `Not` operator to perform logical negation on a `Boolean` expression. The result is a `Boolean` value that represents the reverse of the value of the expression.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstCheck, secondCheck As Boolean
firstCheck = Not (a > b)
secondCheck = Not (b > a)
```

The preceding example produces results of `False` and `True`, respectively.

Example

The following example uses the `Not` operator to perform logical negation of the individual bits of a numeric expression. The bit in the result pattern is set to the reverse of the corresponding bit in the operand pattern, including the sign bit.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstPattern, secondPattern, thirdPattern As Integer
firstPattern = (Not a)
secondPattern = (Not b)
thirdPattern = (Not c)
```

The preceding example produces results of `-11`, `-9`, and `-7`, respectively.

See also

- [Logical/Bitwise Operators \(Visual Basic\)](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Logical and Bitwise Operators in Visual Basic](#)

Or Operator (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

Performs a logical disjunction on two `Boolean` expressions, or a bitwise disjunction on two numeric expressions.

Syntax

```
result = expression1 Or expression2
```

Parts

`result`

Required. Any `Boolean` or numeric expression. For `Boolean` comparison, `result` is the inclusive logical disjunction of two `Boolean` values. For bitwise operations, `result` is a numeric value representing the inclusive bitwise disjunction of two numeric bit patterns.

`expression1`

Required. Any `Boolean` or numeric expression.

`expression2`

Required. Any `Boolean` or numeric expression.

Remarks

For `Boolean` comparison, `result` is `False` if and only if both `expression1` and `expression2` evaluate to `False`. The following table illustrates how `result` is determined.

IF <code>EXPRESSION1</code> IS	AND <code>EXPRESSION2</code> IS	THE VALUE OF <code>RESULT</code> IS
<code>True</code>	<code>True</code>	<code>True</code>
<code>True</code>	<code>False</code>	<code>True</code>
<code>False</code>	<code>True</code>	<code>True</code>
<code>False</code>	<code>False</code>	<code>False</code>

NOTE

In a `Boolean` comparison, the `or` operator always evaluates both expressions, which could include making procedure calls. The [OrElse Operator](#) performs *short-circuiting*, which means that if `expression1` is `True`, then `expression2` is not evaluated.

For bitwise operations, the `or` operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in `result` according to the following table.

IF BIT IN <code>EXPRESSION1</code> IS	AND BIT IN <code>EXPRESSION2</code> IS	THE BIT IN <code>RESULT</code> IS
1	1	1
1	0	1
0	1	1
0	0	0

NOTE

Since the logical and bitwise operators have a lower precedence than other arithmetic and relational operators, any bitwise operations should be enclosed in parentheses to ensure accurate execution.

Data Types

If the operands consist of one `Boolean` expression and one numeric expression, Visual Basic converts the `Boolean` expression to a numeric value (–1 for `True` and 0 for `False`) and performs a bitwise operation.

For a `Boolean` comparison, the data type of the result is `Boolean`. For a bitwise comparison, the result data type is a numeric type appropriate for the data types of `expression1` and `expression2`. See the "Relational and Bitwise Comparisons" table in [Data Types of Operator Results](#).

Overloading

The `or` 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](#).

Example

The following example uses the `or` operator to perform an inclusive logical disjunction on two expressions. The result is a `Boolean` value that represents whether either of the two expressions is `True`.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstCheck, secondCheck, thirdCheck As Boolean
firstCheck = a > b Or b > c
secondCheck = b > a Or b > c
thirdCheck = b > a Or c > b
```

The preceding example produces results of `True`, `True`, and `False`, respectively.

Example

The following example uses the `or` operator to perform inclusive logical disjunction on the individual bits of two numeric expressions. The bit in the result pattern is set if either of the corresponding bits in the operands is set to 1.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstPattern, secondPattern, thirdPattern As Integer
firstPattern = (a Or b)
secondPattern = (a Or c)
thirdPattern = (b Or c)
```

The preceding example produces results of 10, 14, and 14, respectively.

See also

- [Logical/Bitwise Operators \(Visual Basic\)](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [OrElse Operator](#)
- [Logical and Bitwise Operators in Visual Basic](#)

OrElse Operator (Visual Basic)

10/3/2019 • 2 minutes to read • [Edit Online](#)

Performs short-circuiting inclusive logical disjunction on two expressions.

Syntax

```
result = expression1 OrElse expression2
```

Parts

`result`

Required. Any `Boolean` expression.

`expression1`

Required. Any `Boolean` expression.

`expression2`

Required. Any `Boolean` expression.

Remarks

A logical operation is said to be *short-circuiting* if the compiled code can bypass the evaluation of one expression depending on the result of another expression. If the result of the first expression evaluated determines the final result of the operation, there is no need to evaluate the second expression, because it cannot change the final result. Short-circuiting can improve performance if the bypassed expression is complex, or if it involves procedure calls.

If either or both expressions evaluate to `True`, `result` is `True`. The following table illustrates how `result` is determined.

IF <code>EXPRESSION1</code> IS	AND <code>EXPRESSION2</code> IS	THE VALUE OF <code>RESULT</code> IS
<code>True</code>	(not evaluated)	<code>True</code>
<code>False</code>	<code>True</code>	<code>True</code>
<code>False</code>	<code>False</code>	<code>False</code>

Data Types

The `OrElse` operator is defined only for the [Boolean Data Type](#). Visual Basic converts each operand as necessary to `Boolean` before evaluating the expression. If you assign the result to a numeric type, Visual Basic converts it from `Boolean` to that type such that `False` becomes `0` and `True` becomes `-1`. For more information, see [Boolean Type Conversions](#).

Overloading

The [Or Operator](#) and the [IsTrue Operator](#) 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. Overloading the `Or` and `.IsTrue` operators affects the behavior of the `OrElse` operator. If your code uses `OrElse` on a class or structure that overloads `Or` and `.IsTrue`, be sure you understand their redefined behavior. For more information, see [Operator Procedures](#).

Example

The following example uses the `OrElse` operator to perform logical disjunction on two expressions. The result is a `Boolean` value that represents whether either of the two expressions is true. If the first expression is `True`, the second is not evaluated.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstCheck, secondCheck, thirdCheck As Boolean
firstCheck = a > b OrElse b > c
secondCheck = b > a OrElse b > c
thirdCheck = b > a OrElse c > b
```

The preceding example produces results of `True`, `True`, and `False` respectively. In the calculation of `firstCheck`, the second expression is not evaluated because the first is already `True`. However, the second expression is evaluated in the calculation of `secondCheck`.

Example

The following example shows an `If ... Then` statement containing two procedure calls. If the first call returns `True`, the second procedure is not called. This could produce unexpected results if the second procedure performs important tasks that should always be performed when this section of the code runs.

```
If testFunction(5) = True OrElse otherFunction(4) = True Then
    ' If testFunction(5) is True, otherFunction(4) is not called.
    ' Insert code to be executed.
End If
```

See also

- [Logical/Bitwise Operators \(Visual Basic\)](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Or Operator](#)
- [IsTrue Operator](#)
- [Logical and Bitwise Operators in Visual Basic](#)

Sub Expression (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

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

Syntax

```
Sub ( [ parameterlist ] ) statement
- or -
Sub ( [ parameterlist ] )
    [ statements ]
End Sub
```

Parts

TERM	DEFINITION
parameterlist	Optional. A list of local variable names that represent the parameters of the procedure. The parentheses must be present even when the list is empty. For more information, see Parameter List .
statement	Required. A single statement.
statements	Required. A list of statements.

Remarks

A *lambda expression* is a subroutine that does not have a name and that executes one or more statements. You can use a lambda expression anywhere that 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](#).

Lambda Expression Syntax

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

- A lambda expression does not have a name.
- A lambda expression cannot have a modifier, such as `overloads` or `Overrides`.
- The body of a single-line lambda expression must be a statement, not an expression. The body can consist of a call to a sub procedure, but not a call to a function procedure.
- In a lambda expression, either all parameters must have specified data types or all parameters must be inferred.
- Optional and `ParamArray` parameters are not permitted in lambda expressions.
- Generic parameters are not permitted in lambda expressions.

Example

Following is an example of a lambda expression that writes a value to the console. The example shows both the single-line and multiline lambda expression syntax for a subroutine. For more examples, see [Lambda Expressions](#).

```
Dim writeline1 = Sub(x) Console.WriteLine(x)
Dim writeline2 = Sub(x)
    Console.WriteLine(x)
End Sub

' Write "Hello".
writeline1("Hello")

' Write "World"
writeline2("World")
```

See also

- [Sub Statement](#)
- [Lambda Expressions](#)
- [Operators and Expressions](#)
- [Statements](#)
- [Relaxed Delegate Conversion](#)

TryCast Operator (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Introduces a type conversion operation that does not throw an exception.

Remarks

If an attempted conversion fails, `ctype` and `DirectCast` both throw an `InvalidCastException` error. This can adversely affect the performance of your application. `TryCast` returns `Nothing`, so that instead of having to handle a possible exception, you need only test the returned result against `Nothing`.

You use the `TryCast` keyword the same way you use the [CType Function](#) and the [DirectCast Operator](#) keyword. You supply an expression as the first argument and a type to convert it to as the second argument. `TryCast` operates only on reference types, such as classes and interfaces. It requires an inheritance or implementation relationship between the two types. This means that one type must inherit from or implement the other.

Errors and Failures

`TryCast` generates a compiler error if it detects that no inheritance or implementation relationship exists. But the lack of a compiler error does not guarantee a successful conversion. If the desired conversion is narrowing, it could fail at run time. If this happens, `TryCast` returns `Nothing`.

Conversion Keywords

A comparison of the type conversion keywords is as follows.

KEYWORD	DATA TYPES	ARGUMENT RELATIONSHIP	RUN-TIME FAILURE
CType Function	Any data types	Widening or narrowing conversion must be defined between the two data types	Throws <code>InvalidCastException</code>
DirectCast Operator	Any data types	One type must inherit from or implement the other type	Throws <code>InvalidCastException</code>
<code>TryCast</code>	Reference types only	One type must inherit from or implement the other type	Returns <code>Nothing</code>

Example

The following example shows how to use `TryCast`.

```
Function PrintTypeCode(ByVal obj As Object) As String
    Dim objAsConvertible As IConvertible = TryCast(obj, IConvertible)
    If objAsConvertible Is Nothing Then
        Return obj.ToString() & " does not implement IConvertible"
    Else
        Return "Type code is " & objAsConvertible.GetTypeCode()
    End If
End Function
```

See also

- [Widening and Narrowing Conversions](#)
- [Implicit and Explicit Conversions](#)

TypeOf Operator (Visual Basic)

10/1/2019 • 2 minutes to read • [Edit Online](#)

Checks whether the runtime type of an expression's result is type-compatible with the specified type.

Syntax

```
result = TypeOf objectexpression Is typename
```

```
result = TypeOf objectexpression IsNot typename
```

Parts

`result`

Returned. A `Boolean` value.

`objectexpression`

Required. Any expression that evaluates to a reference type.

`typename`

Required. Any data type name.

Remarks

The `TypeOf` operator determines whether the run-time type of `objectexpression` is compatible with `typename`.

The compatibility depends on the type category of `typename`. The following table shows how compatibility is determined.

TYPE CATEGORY OF <code>TYPENAME</code>	COMPATIBILITY CRITERION
Class	<code>objectexpression</code> is of type <code>typename</code> or inherits from <code>typename</code>
Structure	<code>objectexpression</code> is of type <code>typename</code>
Interface	<code>objectexpression</code> implements <code>typename</code> or inherits from a class that implements <code>typename</code>

If the run-time type of `objectexpression` satisfies the compatibility criterion, `result` is `True`. Otherwise, `result` is `False`. If `objectexpression` is null, then `TypeOf ... Is` returns `False`, and `... IsNot` returns `True`.

`TypeOf` is always used with the `Is` keyword to construct a `TypeOf ... Is` expression, or with the `IsNot` keyword to construct a `TypeOf ... IsNot` expression.

Example

The following example uses `TypeOf ... Is` expressions to test the type compatibility of two object reference variables with various data types.

```
Dim refInteger As Object = 2
MsgBox("TypeOf Object[Integer] Is Integer? " & TypeOf refInteger Is Integer)
MsgBox("TypeOf Object[Integer] Is Double? " & TypeOf refInteger Is Double)
Dim refForm As Object = New System.Windows.Forms.Form
MsgBox("TypeOf Object[Form] Is Form? " & TypeOf refForm Is System.Windows.Forms.Form)
MsgBox("TypeOf Object[Form] Is Label? " & TypeOf refForm Is System.Windows.Forms.Label)
MsgBox("TypeOf Object[Form] Is Control? " & TypeOf refForm Is System.Windows.Forms.Control)
MsgBox("TypeOf Object[Form] Is IComponent? " & TypeOf refForm Is System.ComponentModel.IComponent)
```

The variable `refInteger` has a run-time type of `Integer`. It is compatible with `Integer` but not with `Double`. The variable `refForm` has a run-time type of `Form`. It is compatible with `Form` because that is its type, with `Control` because `Form` inherits from `Control`, and with `IComponent` because `Form` inherits from `Component`, which implements `IComponent`. However, `refForm` is not compatible with `Label`.

See also

- [Is Operator](#)
- [IsNot Operator](#)
- [Comparison Operators in Visual Basic](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Operators and Expressions](#)

Xor Operator (Visual Basic)

10/1/2019 • 3 minutes to read • [Edit Online](#)

Performs a logical exclusion on two `Boolean` expressions, or a bitwise exclusion on two numeric expressions.

Syntax

```
result = expression1 Xor expression2
```

Parts

`result`

Required. Any `Boolean` or numeric variable. For Boolean comparison, `result` is the logical exclusion (exclusive logical disjunction) of two `Boolean` values. For bitwise operations, `result` is a numeric value that represents the bitwise exclusion (exclusive bitwise disjunction) of two numeric bit patterns.

`expression1`

Required. Any `Boolean` or numeric expression.

`expression2`

Required. Any `Boolean` or numeric expression.

Remarks

For Boolean comparison, `result` is `True` if and only if exactly one of `expression1` and `expression2` evaluates to `True`. That is, if and only if `expression1` and `expression2` evaluate to opposite `Boolean` values. The following table illustrates how `result` is determined.

IF <code>EXPRESSION1</code> IS	AND <code>EXPRESSION2</code> IS	THE VALUE OF <code>RESULT</code> IS
<code>True</code>	<code>True</code>	<code>False</code>
<code>True</code>	<code>False</code>	<code>True</code>
<code>False</code>	<code>True</code>	<code>True</code>
<code>False</code>	<code>False</code>	<code>False</code>

NOTE

In a Boolean comparison, the `Xor` operator always evaluates both expressions, which could include making procedure calls. There is no short-circuiting counterpart to `Xor`, because the result always depends on both operands. For *short-circuiting* logical operators, see [AndAlso Operator](#) and [OrElse Operator](#).

For bitwise operations, the `Xor` operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in `result` according to the following table.

IF BIT IN <code>expression1</code> IS	AND BIT IN <code>expression2</code> IS	THE BIT IN <code>result</code> IS
1	1	0
1	0	1
0	1	1
0	0	0

NOTE

Since the logical and bitwise operators have a lower precedence than other arithmetic and relational operators, any bitwise operations should be enclosed in parentheses to ensure accurate execution.

For example, `5 xor 3` is 6. To see why this is so, convert 5 and 3 to their binary representations, 101 and 011. Then use the previous table to determine that 101 Xor 011 is 110, which is the binary representation of the decimal number 6.

Data Types

If the operands consist of one `Boolean` expression and one numeric expression, Visual Basic converts the `Boolean` expression to a numeric value (-1 for `True` and 0 for `False`) and performs a bitwise operation.

For a `Boolean` comparison, the data type of the result is `Boolean`. For a bitwise comparison, the result data type is a numeric type appropriate for the data types of `expression1` and `expression2`. See the "Relational and Bitwise Comparisons" table in [Data Types of Operator Results](#).

Overloading

The `xor` 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, make sure you understand its redefined behavior. For more information, see [Operator Procedures](#).

Example

The following example uses the `xor` operator to perform logical exclusion (exclusive logical disjunction) on two expressions. The result is a `Boolean` value that represents whether exactly one of the expressions is `True`.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstCheck, secondCheck, thirdCheck As Boolean
firstCheck = a > b Xor b > c
secondCheck = b > a Xor b > c
thirdCheck = b > a Xor c > b
```

The previous example produces results of `False`, `True`, and `False`, respectively.

Example

The following example uses the `xor` operator to perform logical exclusion (exclusive logical disjunction) on the individual bits of two numeric expressions. The bit in the result pattern is set if exactly one of the corresponding

bits in the operands is set to 1.

```
Dim a As Integer = 10 ' 1010 in binary
Dim b As Integer = 8  ' 1000 in binary
Dim c As Integer = 6  ' 0110 in binary
Dim firstPattern, secondPattern, thirdPattern As Integer
firstPattern = (a Xor b) ' 2, 0010 in binary
secondPattern = (a Xor c) ' 12, 1100 in binary
thirdPattern = (b Xor c) ' 14, 1110 in binary
```

The previous example produces results of 2, 12, and 14, respectively.

See also

- [Logical/Bitwise Operators \(Visual Basic\)](#)
- [Operator Precedence in Visual Basic](#)
- [Operators Listed by Functionality](#)
- [Logical and Bitwise Operators in Visual Basic](#)

Properties (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

This page lists the properties that are members of Visual Basic modules. Other properties that are members of specific Visual Basic objects are listed in [Objects](#).

Visual Basic Properties

DateString	Returns or sets a <code>String</code> value representing the current date according to your system.
Now	Returns a <code>Date</code> value containing the current date and time according to your system.
ScriptEngine	Returns a <code>String</code> representing the runtime currently in use.
ScriptEngineBuildVersion	Returns an <code>Integer</code> containing the build version number of the runtime currently in use.
ScriptEngineMajorVersion	Returns an <code>Integer</code> containing the major version number of the runtime currently in use.
ScriptEngineMinorVersion	Returns an <code>Integer</code> containing the minor version number of the runtime currently in use.
TimeOfDay	Returns or sets a <code>Date</code> value containing the current time of day according to your system.
Timer	Returns a <code>Double</code> value representing the number of seconds elapsed since midnight.
TimeString	Returns or sets a <code>String</code> value representing the current time of day according to your system.
Today	Returns or sets a <code>Date</code> value containing the current date according to your system.

See also

- [Visual Basic Language Reference](#)
- [Visual Basic](#)

Queries (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic enables you to create Language-Integrated Query (LINQ) expressions in your code.

In This Section

[Aggregate Clause](#)

Describes the `Aggregate` clause, which applies one or more aggregate functions to a collection.

[Distinct Clause](#)

Describes the `Distinct` clause, which restricts the values of the current range variable to eliminate duplicate values in query results.

[From Clause](#)

Describes the `From` clause, which specifies a collection and a range variable for a query.

[Group By Clause](#)

Describes the `Group By` clause, which groups the elements of a query result and can be used to apply aggregate functions to each group.

[Group Join Clause](#)

Describes the `Group Join` clause, which combines two collections into a single hierarchical collection.

[Join Clause](#)

Describes the `Join` clause, which combines two collections into a single collection.

[Let Clause](#)

Describes the `Let` clause, which computes a value and assigns it to a new variable in the query.

[Order By Clause](#)

Describes the `Order By` clause, which specifies the sort order for columns in a query.

[Select Clause](#)

Describes the `Select` clause, which declares a set of range variables for a query.

[Skip Clause](#)

Describes the `Skip` clause, which bypasses a specified number of elements in a collection and then returns the remaining elements.

[Skip While Clause](#)

Describes the `Skip While` clause, which bypasses elements in a collection as long as a specified condition is `true` and then returns the remaining elements.

[Take Clause](#)

Describes the `Take` clause, which returns a specified number of contiguous elements from the start of a collection.

[Take While Clause](#)

Describes the `Take While` clause, which includes elements in a collection as long as a specified condition is `true` and bypasses the remaining elements.

[Where Clause](#)

Describes the `Where` clause, which specifies a filtering condition for a query.

See also

- [LINQ](#)
- [Introduction to LINQ in Visual Basic](#)

Aggregate Clause (Visual Basic)

10/7/2019 • 6 minutes to read • [Edit Online](#)

Applies one or more aggregate functions to a collection.

Syntax

```
Aggregate element [As type] In collection _  
[, element2 [As type2] In collection2, [...]]  
[ clause ]  
Into expressionList
```

Parts

TERM	DEFINITION
<code>element</code>	Required. Variable used to iterate through the elements of the collection.
<code>type</code>	Optional. The type of <code>element</code> . If no type is specified, the type of <code>element</code> is inferred from <code>collection</code> .
<code>collection</code>	Required. Refers to the collection to operate on.
<code>clause</code>	Optional. One or more query clauses, such as a <code>Where</code> clause, to refine the query result to apply the aggregate clause or clauses to.
<code>expressionList</code>	Required. One or more comma-delimited expressions that identify an aggregate function to apply to the collection. You can apply an alias to an aggregate function to specify a member name for the query result. If no alias is supplied, the name of the aggregate function is used. For examples, see the section about aggregate functions later in this topic.

Remarks

The `Aggregate` clause can be used to include aggregate functions in your queries. Aggregate functions perform checks and computations over a set of values and return a single value. You can access the computed value by using a member of the query result type. The standard aggregate functions that you can use are the `All`, `Any`, `Average`, `Count`, `LongCount`, `Max`, `Min`, and `Sum` functions. These functions are familiar to developers who are familiar with aggregates in SQL. They are described in the following section of this topic.

The result of an aggregate function is included in the query result as a field of the query result type. You can supply an alias for the aggregate function result to specify the name of the member of the query result type that will hold the aggregate value. If no alias is supplied, the name of the aggregate function is used.

The `Aggregate` clause can begin a query, or it can be included as an additional clause in a query. If the `Aggregate` clause begins a query, the result is a single value that is the result of the aggregate function specified in the `Into` clause. If more than one aggregate function is specified in the `Into` clause, the query returns a single type with a

separate property to reference the result of each aggregate function in the `Into` clause. If the `Aggregate` clause is included as an additional clause in a query, the type returned in the query collection will have a separate property to reference the result of each aggregate function in the `Into` clause.

Aggregate Functions

The following are the standard aggregate functions that can be used with the `Aggregate` clause.

All

Returns `true` if all elements in the collection satisfy a specified condition; otherwise returns `false`. The following is an example:

```
Dim customerList1 = Aggregate order In orders
    Into AllOrdersOver100 = All(order.Total >= 100)
```

Any

Returns `true` if any element in the collection satisfies a specified condition; otherwise returns `false`. The following is an example:

```
Dim customerList2 = From cust In customers
    Aggregate order In cust.Orders
    Into AnyOrderOver500 = Any(order.Total >= 500)
```

Average

Computes the average of all elements in the collection, or computes a supplied expression for all elements in the collection. The following is an example:

```
Dim customerOrderAverage = Aggregate order In orders
    Into Average(order.Total)
```

Count

Counts the number of elements in the collection. You can supply an optional `Boolean` expression to count only the number of elements in the collection that satisfy a condition. The following is an example:

```
Dim customerOrderAfter1996 = From cust In customers
    Aggregate order In cust.Orders
    Into Count(order.OrderDate > #12/31/1996#)
```

Group

Refers to query results that are grouped as a result of a `Group By` or `Group Join` clause. The `Group` function is valid only in the `Into` clause of a `Group By` or `Group Join` clause. For more information and examples, see [Group By Clause](#) and [Group Join Clause](#).

LongCount

Counts the number of elements in the collection. You can supply an optional `Boolean` expression to count only the number of elements in the collection that satisfy a condition. Returns the result as a `Long`. For an example, see the `Count` aggregate function.

Max

Computes the maximum value from the collection, or computes a supplied expression for all elements in the collection. The following is an example:

```
Dim customerMaxOrder = Aggregate order In orders
    Into MaxOrder = Max(order.Total)
```

Min

Computes the minimum value from the collection, or computes a supplied expression for all elements in the collection. The following is an example:

```
Dim customerMinOrder = From cust In customers
    Aggregate order In cust.Orders
    Into MinOrder = Min(order.Total)
```

Sum

Computes the sum of all elements in the collection, or computes a supplied expression for all elements in the collection. The following is an example:

```
Dim customerTotals = From cust In customers
    Aggregate order In cust.Orders
    Into Sum(order.Total)
```

Example

The following example shows how to use the `Aggregate` clause to apply aggregate functions to a query result.

```
Public Sub AggregateSample()
    Dim customers = GetCustomerList()

    Dim customerOrderTotal =
        From cust In customers
        Aggregate order In cust.Orders
        Into Sum(order.Total), MaxOrder = Max(order.Total),
        MinOrder = Min(order.Total), Avg = Average(order.Total)

    For Each customer In customerOrderTotal
        Console.WriteLine(customer.cust.CompanyName & vbCrLf &
            vbTab & "Sum = " & customer.Sum & vbCrLf &
            vbTab & "Min = " & customer.MinOrder & vbCrLf &
            vbTab & "Max = " & customer.MaxOrder & vbCrLf &
            vbTab & "Avg = " & customer.Avg.ToString("#.##"))

    Next
End Sub
```

Creating User-Defined Aggregate Functions

You can include your own custom aggregate functions in a query expression by adding extension methods to the `IEnumerable<T>` type. Your custom method can then perform a calculation or operation on the enumerable collection that has referenced your aggregate function. For more information about extension methods, see [Extension Methods](#).

For example, the following example shows a custom aggregate function that calculates the median value of a collection of numbers. There are two overloads of the `Median` extension method. The first overload accepts, as input, a collection of type `IEnumerable(Of Double)`. If the `Median` aggregate function is called for a query field of type `Double`, this method will be called. The second overload of the `Median` method can be passed any generic type. The generic overload of the `Median` method takes a second parameter that references the `Func(Of T, Double)` lambda expression to project a value for a type (from a collection) as the corresponding

value of type `Double`. It then delegates the calculation of the median value to the other overload of the `Median` method. For more information about lambda expressions, see [Lambda Expressions](#).

```
Imports System.Runtime.CompilerServices

Module UserDefinedAggregates

    ' Calculate the median value for a collection of type Double.
    <Extension()
    Function Median(ByVal values As IEnumerable(Of Double)) As Double
        If values.Count = 0 Then
            Throw New InvalidOperationException("Cannot compute median for an empty set.")
        End If

        Dim sortedList = From number In values
                        Order By number

        Dim medianValue As Double

        Dim itemIndex = CInt(Int(sortedList.Count / 2))

        If sortedList.Count Mod 2 = 0 Then
            ' Even number of items in list.
            medianValue = ((sortedList(itemIndex) + sortedList(itemIndex - 1)) / 2)
        Else
            ' Odd number of items in list.
            medianValue = sortedList(itemIndex)
        End If

        Return medianValue
    End Function

    ' "Cast" the collection of generic items as type Double and call the
    ' Median() method to calculate the median value.
    <Extension()
    Function Median(Of T)(ByVal values As IEnumerable(Of T),
                           ByVal selector As Func(Of T, Double)) As Double
        Return (From element In values Select selector(element)).Median()
    End Function

End Module
```

The following example shows sample queries that call the `Median` aggregate function on a collection of type `Integer`, and a collection of type `Double`. The query that calls the `Median` aggregate function on the collection of type `Double` calls the overload of the `Median` method that accepts, as input, a collection of type `Double`. The query that calls the `Median` aggregate function on the collection of type `Integer` calls the generic overload of the `Median` method.

```
Module Module1

Sub Main()
    Dim numbers1 = {1, 2, 3, 4, 5}

    Dim query1 = Aggregate num In numbers1 Into Median(num)

    Console.WriteLine("Median = " & query1)

    Dim numbers2 = {1.9, 2, 8, 4, 5.7, 6, 7.2, 0}

    Dim query2 = Aggregate num In numbers2 Into Median()

    Console.WriteLine("Median = " & query2)
End Sub

End Module
```

See also

- [Introduction to LINQ in Visual Basic](#)
- [Queries](#)
- [Select Clause](#)
- [From Clause](#)
- [Where Clause](#)
- [Group By Clause](#)

Distinct Clause (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Restricts the values of the current range variable to eliminate duplicate values in subsequent query clauses.

Syntax

```
Distinct
```

Remarks

You can use the `Distinct` clause to return a list of unique items. The `Distinct` clause causes the query to ignore duplicate query results. The `Distinct` clause applies to duplicate values for all return fields specified by the `Select` clause. If no `Select` clause is specified, the `Distinct` clause is applied to the range variable for the query identified in the `From` clause. If the range variable is not an immutable type, the query will only ignore a query result if all members of the type match an existing query result.

Example

The following query expression joins a list of customers and a list of customer orders. The `Distinct` clause is included to return a list of unique customer names and order dates.

```
Dim customerOrders = From cust In customers, ord In orders  
    Where cust.CustomerID = ord.CustomerID  
    Select cust.CompanyName, ord.OrderDate  
    Distinct
```

See also

- [Introduction to LINQ in Visual Basic](#)
- [Queries](#)
- [From Clause](#)
- [Select Clause](#)
- [Where Clause](#)

Equals Clause (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Compares keys from collections being joined.

Remarks

The `Equals` keyword is used in the following contexts:

[Group Join Clause](#)

[Join Clause](#)

See also

- [Keywords](#)

From Clause (Visual Basic)

10/7/2019 • 3 minutes to read • [Edit Online](#)

Specifies one or more range variables and a collection to query.

Syntax

```
From element [ As type ] In collection [ _ ]
[, element2 [ As type2 ] In collection2 [, ... ] ]
```

Parts

TERM	DEFINITION
<code>element</code>	Required. A <i>range variable</i> used to iterate through the elements of the collection. A range variable is used to refer to each member of the <code>collection</code> as the query iterates through the <code>collection</code> . Must be an enumerable type.
<code>type</code>	Optional. The type of <code>element</code> . If no <code>type</code> is specified, the type of <code>element</code> is inferred from <code>collection</code> .
<code>collection</code>	Required. Refers to the collection to be queried. Must be an enumerable type.

Remarks

The `From` clause is used to identify the source data for a query and the variables that are used to refer to an element from the source collection. These variables are called *range variables*. The `From` clause is required for a query, except when the `Aggregate` clause is used to identify a query that returns only aggregated results. For more information, see [Aggregate Clause](#).

You can specify multiple `From` clauses in a query to identify multiple collections to be joined. When multiple collections are specified, they are iterated over independently, or you can join them if they are related. You can join collections implicitly by using the `Select` clause, or explicitly by using the `Join` or `Group Join` clauses. As an alternative, you can specify multiple range variables and collections in a single `From` clause, with each related range variable and collection separated from the others by a comma. The following code example shows both syntax options for the `From` clause.

```
' Multiple From clauses in a query.
Dim result = From var1 In collection1, var2 In collection2

' Equivalent syntax with a single From clause.
Dim result2 = From var1 In collection1
              From var2 In collection2
```

The `From` clause defines the scope of a query, which is similar to the scope of a `For` loop. Therefore, each `element` range variable in the scope of a query must have a unique name. Because you can specify multiple `From` clauses for a query, subsequent `From` clauses can refer to range variables in the `From` clause, or they can

refer to range variables in a previous `From` clause. For example, the following example shows a nested `From` clause where the collection in the second clause is based on a property of the range variable in the first clause.

```
Dim allOrders = From cust In GetCustomerList()
    From ord In cust.Orders
    Select ord
```

Each `From` clause can be followed by any combination of additional query clauses to refine the query. You can refine the query in the following ways:

- Combine multiple collections implicitly by using the `From` and `Select` clauses, or explicitly by using the `Join` or `Group Join` clauses.
- Use the `Where` clause to filter the query result.
- Sort the result by using the `Order By` clause.
- Group similar results together by using the `Group By` clause.
- Use the `Aggregate` clause to identify aggregate functions to evaluate for the whole query result.
- Use the `Let` clause to introduce an iteration variable whose value is determined by an expression instead of a collection.
- Use the `Distinct` clause to ignore duplicate query results.
- Identify parts of the result to return by using the `Skip`, `Take`, `Skip While`, and `Take While` clauses.

Example

The following query expression uses a `From` clause to declare a range variable `cust` for each `Customer` object in the `customers` collection. The `Where` clause uses the range variable to restrict the output to customers from the specified region. The `For Each` loop displays the company name for each customer in the query result.

```
Sub DisplayCustomersForRegion(ByVal customers As List(Of Customer),
    ByVal region As String)

    Dim customersForRegion = From cust In customers
        Where cust.Region = region

    For Each cust In customersForRegion
        Console.WriteLine(cust.CompanyName)
    Next
End Sub
```

See also

- [Queries](#)
- [Introduction to LINQ in Visual Basic](#)
- [For Each...Next Statement](#)
- [For...Next Statement](#)
- [Select Clause](#)
- [Where Clause](#)
- [Aggregate Clause](#)
- [Distinct Clause](#)
- [Join Clause](#)

- [Group Join Clause](#)
- [Order By Clause](#)
- [Let Clause](#)
- [Skip Clause](#)
- [Take Clause](#)
- [Skip While Clause](#)
- [Take While Clause](#)

Group By Clause (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Groups the elements of a query result. Can also be used to apply aggregate functions to each group. The grouping operation is based on one or more keys.

Syntax

```
Group [ listField1 [, listField2 [...] ] By keyExp1 [, keyExp2 [...] ]  
    Into aggregateList
```

Parts

- `listField1`, `listField2`

Optional. One or more fields of the query variable or variables that explicitly identify the fields to be included in the grouped result. If no fields are specified, all fields of the query variable or variables are included in the grouped result.

- `keyExp1`

Required. An expression that identifies the key to use to determine the groups of elements. You can specify more than one key to specify a composite key.

- `keyExp2`

Optional. One or more additional keys that are combined with `keyExp1` to create a composite key.

- `aggregateList`

Required. One or more expressions that identify how the groups are aggregated. To identify a member name for the grouped results, use the `Group` keyword, which can be in either of the following forms:

```
    Into Group
```

-or-

```
    Into <alias> = Group
```

You can also include aggregate functions to apply to the group.

Remarks

You can use the `Group By` clause to break the results of a query into groups. The grouping is based on a key or a composite key consisting of multiple keys. Elements that are associated with matching key values are included in the same group.

You use the `aggregateList` parameter of the `Into` clause and the `Group` keyword to identify the member name that is used to reference the group. You can also include aggregate functions in the `Into` clause to compute values for the grouped elements. For a list of standard aggregate functions, see [Aggregate Clause](#).

Example

The following code example groups a list of customers based on their location (country/region) and provides a count of the customers in each group. The results are ordered by country/region name. The grouped results are ordered by city name.

```
Public Sub GroupBySample()
    Dim customers = GetCustomerList()

    Dim customersByCountry = From cust In customers
        Order By cust.City
        Group By CountryName = cust.Country
        Into RegionalCustomers = Group, Count()
        Order By CountryName

    For Each country In customersByCountry
        Console.WriteLine(country.CountryName &
            " (" & country.Count & ")" & vbCrLf)

        For Each customer In country.RegionalCustomers
            Console.WriteLine(vbTab & customer.CompanyName &
                " (" & customer.City & ")")
        Next
    Next
End Sub
```

See also

- [Introduction to LINQ in Visual Basic](#)
- [Queries](#)
- [Select Clause](#)
- [From Clause](#)
- [Order By Clause](#)
- [Aggregate Clause](#)
- [Group Join Clause](#)

Group Join Clause (Visual Basic)

10/7/2019 • 3 minutes to read • [Edit Online](#)

Combines two collections into a single hierarchical collection. The join operation is based on matching keys.

Syntax

```
Group Join element [As type] In collection _  
    On key1 Equals key2 [ And key3 Equals key4 [ ... ] ] _  
    Into expressionList
```

Parts

TERM	DEFINITION
<code>element</code>	Required. The control variable for the collection being joined.
<code>type</code>	Optional. The type of <code>element</code> . If no <code>type</code> is specified, the type of <code>element</code> is inferred from <code>collection</code> .
<code>collection</code>	Required. The collection to combine with the collection that is on the left side of the <code>Group Join</code> operator. A <code>Group Join</code> clause can be nested in a <code>Join</code> clause or in another <code>Group Join</code> clause.
<code>key1 Equals key2</code>	Required. Identifies keys for the collections being joined. You must use the <code>Equals</code> operator to compare keys from the collections being joined. You can combine join conditions by using the <code>And</code> operator to identify multiple keys. The <code>key1</code> parameter must be from the collection on the left side of the <code>Join</code> operator. The <code>key2</code> parameter must be from the collection on the right side of the <code>Join</code> operator. The keys used in the join condition can be expressions that include more than one item from the collection. However, each key expression can contain only items from its respective collection.
<code>expressionList</code>	Required. One or more expressions that identify how the groups of elements from the collection are aggregated. To identify a member name for the grouped results, use the <code>Group</code> keyword (<code><alias> = Group</code>). You can also include aggregate functions to apply to the group.

Remarks

The `Group Join` clause combines two collections based on matching key values from the collections being joined. The resulting collection can contain a member that references a collection of elements from the second collection that match the key value from the first collection. You can also specify aggregate functions to apply to the grouped elements from the second collection. For information about aggregate functions, see [Aggregate](#)

Clause.

Consider, for example, a collection of managers and a collection of employees. Elements from both collections have a ManagerID property that identifies the employees that report to a particular manager. The results from a join operation would contain a result for each manager and employee with a matching ManagerID value. The results from a `Group Join` operation would contain the complete list of managers. Each manager result would have a member that referenced the list of employees that were a match for the specific manager.

The collection resulting from a `Group Join` operation can contain any combination of values from the collection identified in the `From` clause and the expressions identified in the `Into` clause of the `Group Join` clause. For more information about valid expressions for the `Into` clause, see [Aggregate Clause](#).

A `Group Join` operation will return all results from the collection identified on the left side of the `Group Join` operator. This is true even if there are no matches in the collection being joined. This is like a `LEFT OUTER JOIN` in SQL.

You can use the `Join` clause to combine collections into a single collection. This is equivalent to an `INNER JOIN` in SQL.

Example

The following code example joins two collections by using the `Group Join` clause.

```
Dim customerList = From cust In customers
    Group Join ord In orders On
        cust.CustomerID Equals ord.CustomerID
    Into CustomerOrders = Group,
        OrderTotal = Sum(ord.Total)
    Select cust.CompanyName, cust.CustomerID,
        CustomerOrders, OrderTotal

For Each customer In customerList
    Console.WriteLine(customer.CompanyName &
        " (" & customer.OrderTotal & ")")

    For Each order In customer.CustomerOrders
        Console.WriteLine(vbTab & order.OrderID & ": " & order.Total)
    Next
Next
```

See also

- [Introduction to LINQ in Visual Basic](#)
- [Queries](#)
- [Select Clause](#)
- [From Clause](#)
- [Join Clause](#)
- [Where Clause](#)
- [Group By Clause](#)

Join Clause (Visual Basic)

10/18/2019 • 3 minutes to read • [Edit Online](#)

Combines two collections into a single collection. The join operation is based on matching keys and uses the `Equals` operator.

Syntax

```
Join element In collection _  
[ joinClause _ ]  
[ groupJoinClause ... _ ]  
On key1 Equals key2 [ And key3 Equals key4 [ ... ]]
```

Parts

`element` Required. The control variable for the collection being joined.

`collection`

Required. The collection to combine with the collection identified on the left side of the `Join` operator. A `Join` clause can be nested in another `Join` clause, or in a `Group Join` clause.

`joinClause`

Optional. One or more additional `Join` clauses to further refine the query.

`groupJoinClause`

Optional. One or more additional `Group Join` clauses to further refine the query.

`key1 Equals key2`

Required. Identifies keys for the collections being joined. You must use the `Equals` operator to compare keys from the collections being joined. You can combine join conditions by using the `And` operator to identify multiple keys. `key1` must be from the collection on the left side of the `Join` operator. `key2` must be from the collection on the right side of the `Join` operator.

The keys used in the join condition can be expressions that include more than one item from the collection. However, each key expression can contain only items from its respective collection.

Remarks

The `Join` clause combines two collections based on matching key values from the collections being joined. The resulting collection can contain any combination of values from the collection identified on the left side of the `Join` operator and the collection identified in the `Join` clause. The query will return only results for which the condition specified by the `Equals` operator is met. This is equivalent to an `INNER JOIN` in SQL.

You can use multiple `Join` clauses in a query to join two or more collections into a single collection.

You can perform an implicit join to combine collections without the `Join` clause. To do this, include multiple `In` clauses in your `From` clause and specify a `Where` clause that identifies the keys that you want to use for the join.

You can use the `Group Join` clause to combine collections into a single hierarchical collection. This is like a `LEFT OUTER JOIN` in SQL.

Example

The following code example performs an implicit join to combine a list of customers with their orders.

```
Dim customerIDs() = {"ALFKI", "VICTE", "BLAUS", "TRAIH"}  
  
Dim customerList = From cust In customers, custID In customerIDs  
    Where cust.CustomerID = custID  
    Select cust.CompanyName  
  
For Each companyName In customerList  
    Console.WriteLine(companyName)  
Next
```

Example

The following code example joins two collections by using the `Join` clause.

```
Imports System.Diagnostics  
Imports System.Security.Permissions  
  
Public Class JoinSample  
  
<SecurityPermission(SecurityAction.Demand)>  
Public Sub ListProcesses()  
    Dim processDescriptions As New List(Of ProcessDescription)  
    processDescriptions.Add(New ProcessDescription With {  
        .ProcessName = "explorer",  
        .Description = "Windows Explorer"})  
    processDescriptions.Add(New ProcessDescription With {  
        .ProcessName = "winlogon",  
        .Description = "Windows Logon"})  
    processDescriptions.Add(New ProcessDescription With {  
        .ProcessName = "cmd",  
        .Description = "Command Window"})  
    processDescriptions.Add(New ProcessDescription With {  
        .ProcessName = "iexplore",  
        .Description = "Internet Explorer"})  
  
    Dim processes = From proc In Process.GetProcesses  
        Join desc In processDescriptions  
        On proc.ProcessName Equals desc.ProcessName  
        Select proc.ProcessName, proc.Id, desc.Description  
  
    For Each proc In processes  
        Console.WriteLine("{0} ({1}), {2}",  
            proc.ProcessName, proc.Id, proc.Description)  
    Next  
End Sub  
  
End Class  
  
Public Class ProcessDescription  
    Public ProcessName As String  
    Public Description As String  
End Class
```

This example will produce output similar to the following:

```
winlogon (968), Windows Logon
```

```
explorer (2424), File Explorer
```

Example

The following code example joins two collections by using the `Join` clause with two key columns.

```

Imports System.Diagnostics
Imports System.Security.Permissions

Public Class JoinSample2

    <SecurityPermission(SecurityAction.Demand)>
    Public Sub ListProcesses()
        Dim processDescriptions As New List(Of ProcessDescription2)

        ' 8 = Normal priority, 13 = High priority
        processDescriptions.Add(New ProcessDescription2 With {
            .ProcessName = "explorer",
            .Description = "Windows Explorer",
            .Priority = 8})
        processDescriptions.Add(New ProcessDescription2 With {
            .ProcessName = "winlogon",
            .Description = "Windows Logon",
            .Priority = 13})
        processDescriptions.Add(New ProcessDescription2 With {
            .ProcessName = "cmd",
            .Description = "Command Window",
            .Priority = 8})
        processDescriptions.Add(New ProcessDescription2 With {
            .ProcessName = "iexplore",
            .Description = "Internet Explorer",
            .Priority = 8})

        Dim processes = From proc In Process.GetProcesses
                        Join desc In processDescriptions
                        On proc.ProcessName Equals desc.ProcessName And
                            proc.BasePriority Equals desc.Priority
                        Select proc.ProcessName, proc.Id, desc.Description,
                            desc.Priority

        For Each proc In processes
            Console.WriteLine("{0} ({1}), {2}, Priority = {3}",
                proc.ProcessName,
                proc.Id,
                proc.Description,
                proc.Priority)
        Next
    End Sub

End Class

Public Class ProcessDescription2
    Public ProcessName As String
    Public Description As String
    Public Priority As Integer
End Class

```

The example will produce output similar to the following:

```
winlogon (968), Windows Logon, Priority = 13
```

```
cmd (700), Command Window, Priority = 8
```

```
explorer (2424), File Explorer, Priority = 8
```

See also

- [Introduction to LINQ in Visual Basic](#)
- [Queries](#)
- [Select Clause](#)
- [From Clause](#)
- [Group Join Clause](#)
- [Where Clause](#)

Let Clause (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Computes a value and assigns it to a new variable within the query.

Syntax

```
Let variable = expression [, ...]
```

Parts

TERM	DEFINITION
variable	Required. An alias that can be used to reference the results of the supplied expression.
expression	Required. An expression that will be evaluated and assigned to the specified variable.

Remarks

The `Let` clause enables you to compute values for each query result and reference them by using an alias. The alias can be used in other clauses, such as the `Where` clause. The `Let` clause enables you to create a query statement that is easier to read because you can specify an alias for an expression clause included in the query and substitute the alias each time the expression clause is used.

You can include any number of `variable` and `expression` assignments in the `Let` clause. Separate each assignment with a comma (,).

Example

The following code example uses the `Let` clause to compute a 10 percent discount on products.

```
Dim discountedProducts = From prod In products
    Let Discount = prod.UnitPrice * 0.1
    Where Discount >= 50
    Select prod.ProductName, prod.UnitPrice, Discount

For Each prod In discountedProducts
    Console.WriteLine("Product: {0}, Price: {1}, Discounted Price: {2}",
        prod.ProductName, prod.UnitPrice.ToString("#.00"),
        (prod.UnitPrice - prod.Discount).ToString("#.00"))
Next
```

See also

- [Introduction to LINQ in Visual Basic](#)
- [Queries](#)
- [Select Clause](#)

- From Clause
- Where Clause

Order By Clause (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies the sort order for a query result.

Syntax

```
Order By orderExp1 [ Ascending | Descending ] [, orderExp2 [...] ]
```

Parts

`orderExp1`

Required. One or more fields from the current query result that identify how to order the returned values. The field names must be separated by commas (,). You can identify each field as sorted in ascending or descending order by using the `Ascending` or `Descending` keywords. If no `Ascending` or `Descending` keyword is specified, the default sort order is ascending. The sort order fields are given precedence from left to right.

Remarks

You can use the `Order By` clause to sort the results of a query. The `Order By` clause can only sort a result based on the range variable for the current scope. For example, the `Select` clause introduces a new scope in a query expression with new iteration variables for that scope. Range variables defined before a `Select` clause in a query are not available after the `Select` clause. Therefore, if you want to order your results by a field that is not available in the `Select` clause, you must put the `Order By` clause before the `Select` clause. One example of when you would have to do this is when you want to sort your query by fields that are not returned as part of the result.

Ascending and descending order for a field is determined by the implementation of the [IComparable](#) interface for the data type of the field. If the data type does not implement the [IComparable](#) interface, the sort order is ignored.

Example

The following query expression uses a `From` clause to declare a range variable `book` for the `books` collection. The `Order By` clause sorts the query result by price in ascending order (the default). Books with the same price are sorted by title in ascending order. The `Select` clause selects the `Title` and `Price` properties as the values returned by the query.

```
Dim titlesAscendingPrice = From book In books
                           Order By book.Price, book.Title
                           Select book.Title, book.Price
```

Example

The following query expression uses the `Order By` clause to sort the query result by price in descending order. Books with the same price are sorted by title in ascending order.

```
Dim titlesDescendingPrice = From book In books
                            Order By book.Price Descending, book.Title
                            Select book.Title, book.Price
```

Example

The following query expression uses a `Select` clause to select the book title, price, publish date, and author. It then populates the `Title`, `Price`, `PublishDate`, and `Author` fields of the range variable for the new scope. The `Order By` clause orders the new range variable by author name, book title, and then price. Each column is sorted in the default order (ascending).

```
Dim bookOrders =
    From book In books
    Select book.Title, book.Price, book.PublishDate, book.Author
    Order By Author, Title, Price
```

See also

- [Introduction to LINQ in Visual Basic](#)
- [Queries](#)
- [Select Clause](#)
- [From Clause](#)

Select Clause (Visual Basic)

10/7/2019 • 3 minutes to read • [Edit Online](#)

Defines the result of a query.

Syntax

```
Select [ var1 = ] fieldName1 [, [ var2 = ] fieldName2 [...] ]
```

Parts

`var1`

Optional. An alias that can be used to reference the results of the column expression.

`fieldName1`

Required. The name of the field to return in the query result.

Remarks

You can use the `Select` clause to define the results to return from a query. This enables you to either define the members of a new anonymous type that is created by a query, or to target the members of a named type that is returned by a query. The `Select` clause is not required for a query. If no `Select` clause is specified, the query will return a type based on all members of the range variables identified for the current scope. For more information, see [Anonymous Types](#). When a query creates a named type, it will return a result of type `IEnumerable<T>` where `T` is the created type.

The `Select` clause can reference any variables in the current scope. This includes range variables identified in the `From` clause (or `From` clauses). It also includes any new variables created with an alias by the `Aggregate`, `Let`, `Group By`, or `Group Join` clauses, or variables from a previous `Select` clause in the query expression.

The `Select` clause can also include static values. For example, the following code example shows a query expression in which the `Select` clause defines the query result as a new anonymous type with four members: `ProductName`, `Price`, `Discount`, and `DiscountedPrice`. The `ProductName` and `Price` member values are taken from the product range variable that is defined in the `From` clause. The `DiscountedPrice` member value is calculated in the `Let` clause. The `Discount` member is a static value.

```
' 10% discount
Dim discount_10 = 0.1
Dim priceList =
    From product In products
    Let DiscountedPrice = product.UnitPrice * (1 - discount_10)
    Select product.ProductName, Price = product.UnitPrice,
        Discount = discount_10, DiscountedPrice
```

The `Select` clause introduces a new set of range variables for subsequent query clauses, and previous range variables are no longer in scope. The last `Select` clause in a query expression determines the return value of the query. For example, the following query returns the company name and order ID for every customer order for which the total exceeds 500. The first `Select` clause identifies the range variables for the `Where` clause and the second `Select` clause. The second `Select` clause identifies the values returned by the query as a new anonymous type.

```
Dim customerList = From cust In customers, ord In cust.Orders
    Select Name = cust.CompanyName,
          Total = ord.Total, ord.OrderID
    Where Total > 500
    Select Name, OrderID
```

If the `Select` clause identifies a single item to return, the query expression returns a collection of the type of that single item. If the `Select` clause identifies multiple items to return, the query expression returns a collection of a new anonymous type, based on the selected items. For example, the following two queries return collections of two different types based on the `Select` clause. The first query returns a collection of company names as strings. The second query returns a collection of `Customer` objects populated with the company names and address information.

```
Dim customerNames = From cust In customers
    Select cust.CompanyName

Dim customerInfo As IEnumerable(Of Customer) =
    From cust In customers
    Select New Customer With {.CompanyName = cust.CompanyName,
        .Address = cust.Address,
        .City = cust.City,
        .Region = cust.Region,
        .Country = cust.Country}
```

Example

The following query expression uses a `From` clause to declare a range variable `cust` for the `customers` collection. The `Select` clause selects the customer name and ID value and populates the `CompanyName` and `CustomerID` columns of the new range variable. The `For Each` statement loops over each returned object and displays the `CompanyName` and `CustomerID` columns for each record.

```
Sub SelectCustomerNameAndId(ByVal customers() As Customer)
    Dim nameIds = From cust In customers
        Select cust.CompanyName, cust.CustomerID
    For Each nameId In nameIds
        Console.WriteLine(nameId.CompanyName & ":" & nameId.CustomerID)
    Next
End Sub
```

See also

- [Introduction to LINQ in Visual Basic](#)
- [Queries](#)
- [From Clause](#)
- [Where Clause](#)
- [Order By Clause](#)
- [Anonymous Types](#)

Skip Clause (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Bypasses a specified number of elements in a collection and then returns the remaining elements.

Syntax

```
Skip count
```

Parts

count

Required. A value or an expression that evaluates to the number of elements of the sequence to skip.

Remarks

The `skip` clause causes a query to bypass elements at the beginning of a results list and return the remaining elements. The number of elements to skip is identified by the `count` parameter.

You can use the `skip` clause with the `Take` clause to return a range of data from any segment of a query. To do this, pass the index of the first element of the range to the `skip` clause and the size of the range to the `Take` clause.

When you use the `skip` clause in a query, you may also need to ensure that the results are returned in an order that will enable the `skip` clause to bypass the intended results. For more information about ordering query results, see [Order By Clause](#).

You can use the `SkipWhile` clause to specify that only certain elements are ignored, depending on a supplied condition.

Example

The following code example uses the `skip` clause together with the `Take` clause to return data from a query in pages. The `GetCustomers` function uses the `skip` clause to bypass the customers in the list until the supplied starting index value, and uses the `Take` clause to return a page of customers starting from that index value.

```

Public Sub PagingSample()
    Dim pageNumber As Integer = 0
    Dim pageSize As Integer = 10

    Dim customersPage = GetCustomers(pageNumber * pageSize, pageSize)

    Do While customersPage IsNot Nothing
        Console.WriteLine(vbCrLf & "Page: " & pageNumber + 1 & vbCrLf)

        For Each cust In customersPage
            Console.WriteLine(cust.CustomerID & ", " & cust.CompanyName)
        Next

        Console.WriteLine(vbCrLf)

        pageNumber += 1
        customersPage = GetCustomers(pageNumber * pageSize, pageSize)
    Loop
End Sub

Public Function GetCustomers(ByVal startIndex As Integer,
                            ByVal pageSize As Integer) As List(Of Customer)

    Dim customers = GetCustomerList()

    Dim returnCustomers = From cust In customers
                           Skip startIndex Take pageSize

    If returnCustomers.Count = 0 Then Return Nothing

    Return returnCustomers
End Function

```

See also

- [Introduction to LINQ in Visual Basic](#)
- [Queries](#)
- [Select Clause](#)
- [From Clause](#)
- [Order By Clause](#)
- [Skip While Clause](#)
- [Take Clause](#)

Skip While Clause (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Bypasses elements in a collection as long as a specified condition is `true` and then returns the remaining elements.

Syntax

```
Skip While expression
```

Parts

TERM	DEFINITION
<code>expression</code>	Required. An expression that represents a condition to test elements for. The expression must return a <code>Boolean</code> value or a functional equivalent, such as an <code>Integer</code> to be evaluated as a <code>Boolean</code> .

Remarks

The `skip While` clause bypasses elements from the beginning of a query result until the supplied `expression` returns `false`. After `expression` returns `false`, the query returns all the remaining elements. The `expression` is ignored for the remaining results.

The `skip While` clause differs from the `Where` clause in that the `Where` clause can be used to exclude all elements from a query that do not meet a particular condition. The `Skip While` clause excludes elements only until the first time that the condition is not satisfied. The `skip While` clause is most useful when you are working with an ordered query result.

You can bypass a specific number of results from the beginning of a query result by using the `skip` clause.

Example

The following code example uses the `Skip While` clause to bypass results until the first customer from the United States is found.

```
Public Sub SkipWhileSample()
    Dim customers = GetCustomerList()

    ' Return customers starting from the first U.S. customer encountered.
    Dim customerList = From cust In customers
        Order By cust.Country
        Skip While IsInternationalCustomer(cust)

    For Each cust In customerList
        Console.WriteLine(cust.CompanyName & vbTab & cust.Country)
    Next
End Sub

Public Function IsInternationalCustomer(ByVal cust As Customer) As Boolean
    If cust.Country = "USA" Then Return False

    Return True
End Function
```

See also

- [Introduction to LINQ in Visual Basic](#)
- [Queries](#)
- [Select Clause](#)
- [From Clause](#)
- [Skip Clause](#)
- [Take While Clause](#)
- [Where Clause](#)

Take Clause (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Returns a specified number of contiguous elements from the start of a collection.

Syntax

```
Take count
```

Parts

count

Required. A value or an expression that evaluates to the number of elements of the sequence to return.

Remarks

The `Take` clause causes a query to include a specified number of contiguous elements from the start of a results list. The number of elements to include is specified by the `count` parameter.

You can use the `Take` clause with the `skip` clause to return a range of data from any segment of a query. To do this, pass the index of the first element of the range to the `skip` clause and the size of the range to the `Take` clause. In this case, the `Take` clause must be specified after the `skip` clause.

When you use the `Take` clause in a query, you may also need to ensure that the results are returned in an order that will enable the `Take` clause to include the intended results. For more information about ordering query results, see [Order By Clause](#).

You can use the `TakeWhile` clause to specify that only certain elements be returned, depending on a supplied condition.

Example

The following code example uses the `Take` clause together with the `skip` clause to return data from a query in pages. The `GetCustomers` function uses the `skip` clause to bypass the customers in the list until the supplied starting index value, and uses the `Take` clause to return a page of customers starting from that index value.

```

Public Sub PagingSample()
    Dim pageNumber As Integer = 0
    Dim pageSize As Integer = 10

    Dim customersPage = GetCustomers(pageNumber * pageSize, pageSize)

    Do While customersPage IsNot Nothing
        Console.WriteLine(vbCrLf & "Page: " & pageNumber + 1 & vbCrLf)

        For Each cust In customersPage
            Console.WriteLine(cust.CustomerID & ", " & cust.CompanyName)
        Next

        Console.WriteLine(vbCrLf)

        pageNumber += 1
        customersPage = GetCustomers(pageNumber * pageSize, pageSize)
    Loop
End Sub

Public Function GetCustomers(ByVal startIndex As Integer,
                            ByVal pageSize As Integer) As List(Of Customer)

    Dim customers = GetCustomerList()

    Dim returnCustomers = From cust In customers
                           Skip startIndex Take pageSize

    If returnCustomers.Count = 0 Then Return Nothing

    Return returnCustomers
End Function

```

See also

- [Introduction to LINQ in Visual Basic](#)
- [Queries](#)
- [Select Clause](#)
- [From Clause](#)
- [Order By Clause](#)
- [Take While Clause](#)
- [Skip Clause](#)

Take While Clause (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Includes elements in a collection as long as a specified condition is `true` and bypasses the remaining elements.

Syntax

```
Take While expression
```

Parts

TERM	DEFINITION
<code>expression</code>	Required. An expression that represents a condition to test elements for. The expression must return a <code>Boolean</code> value or a functional equivalent, such as an <code>Integer</code> to be evaluated as a <code>Boolean</code> .

Remarks

The `Take While` clause includes elements from the start of a query result until the supplied `expression` returns `false`. After the `expression` returns `false`, the query will bypass all remaining elements. The `expression` is ignored for the remaining results.

The `Take While` clause differs from the `Where` clause in that the `Where` clause can be used to include all elements from a query that meet a particular condition. The `Take While` clause includes elements only until the first time that the condition is not satisfied. The `Take While` clause is most useful when you are working with an ordered query result.

Example

The following code example uses the `Take While` clause to retrieve results until the first customer without any orders is found.

```
Public Sub TakeWhileSample()
    Dim customers = GetCustomerList()

    ' Return customers until the first customer with no orders is found.
    Dim customersWithOrders = From cust In customers
                                Order By cust.Orders.Count Descending
                                Take While HasOrders(cust)

    For Each cust In customersWithOrders
        Console.WriteLine(cust.CompanyName & " (" & cust.Orders.Length & ")")
    Next
End Sub

Public Function HasOrders(ByVal cust As Customer) As Boolean
    If cust.Orders.Length > 0 Then Return True

    Return False
End Function
```

See also

- [Introduction to LINQ in Visual Basic](#)
- [Queries](#)
- [Select Clause](#)
- [From Clause](#)
- [Take Clause](#)
- [Skip While Clause](#)
- [Where Clause](#)

Where Clause (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies the filtering condition for a query.

Syntax

```
Where condition
```

Parts

`condition`

Required. An expression that determines whether the values for the current item in the collection are included in the output collection. The expression must evaluate to a `Boolean` value or the equivalent of a `Boolean` value. If the condition evaluates to `True`, the element is included in the query result; otherwise, the element is excluded from the query result.

Remarks

The `Where` clause enables you to filter query data by selecting only elements that meet certain criteria. Elements whose values cause the `Where` clause to evaluate to `True` are included in the query result; other elements are excluded. The expression that is used in a `Where` clause must evaluate to a `Boolean` or the equivalent of a `Boolean`, such as an `Integer` that evaluates to `False` when its value is zero. You can combine multiple expressions in a `Where` clause by using logical operators such as `And`, `Or`, `AndAlso`, `OrElse`, `Is`, and `IsNot`.

By default, query expressions are not evaluated until they are accessed—for example, when they are data-bound or iterated through in a `For` loop. As a result, the `Where` clause is not evaluated until the query is accessed. If you have values external to the query that are used in the `Where` clause, ensure that the appropriate value is used in the `Where` clause at the time the query is executed. For more information about query execution, see [Writing Your First LINQ Query](#).

You can call functions within a `Where` clause to perform a calculation or operation on a value from the current element in the collection. Calling a function in a `Where` clause can cause the query to be executed immediately when it is defined instead of when it is accessed. For more information about query execution, see [Writing Your First LINQ Query](#).

Example

The following query expression uses a `From` clause to declare a range variable `cust` for each `Customer` object in the `customers` collection. The `Where` clause uses the range variable to restrict the output to customers from the specified region. The `For Each` loop displays the company name for each customer in the query result.

```

Sub DisplayCustomersForRegion(ByVal customers As List(Of Customer),
                             ByVal region As String)

    Dim customersForRegion = From cust In customers
                             Where cust.Region = region

    For Each cust In customersForRegion
        Console.WriteLine(cust.CompanyName)
    Next
End Sub

```

Example

The following example uses `And` and `or` logical operators in the `Where` clause.

```

Private Sub DisplayElements()
    Dim elements As List(Of Element) = BuildList()

    ' Get a list of elements that have an atomic number from 12 to 14,
    ' or that have a name that ends in "r".
    Dim subset = From theElement In elements
                 Where (theElement.AtomicNumber >= 12 And theElement.AtomicNumber < 15) _
                 Or theElement.Name.EndsWith("r")
                 Order By theElement.Name

    For Each theElement In subset
        Console.WriteLine(theElement.Name & " " & theElement.AtomicNumber)
    Next

    ' Output:
    ' Aluminum 13
    ' Magnesium 12
    ' Silicon 14
    ' Sulfur 16
End Sub

Private Function BuildList() As List(Of Element)
    Return New List(Of Element) From
    {
        {New Element With {.Name = "Sodium", .AtomicNumber = 11}},
        {New Element With {.Name = "Magnesium", .AtomicNumber = 12}},
        {New Element With {.Name = "Aluminum", .AtomicNumber = 13}},
        {New Element With {.Name = "Silicon", .AtomicNumber = 14}},
        {New Element With {.Name = "Phosphorous", .AtomicNumber = 15}},
        {New Element With {.Name = "Sulfur", .AtomicNumber = 16}}
    }
End Function

Public Class Element
    Public Property Name As String
    Public Property AtomicNumber As Integer
End Class

```

See also

- [Introduction to LINQ in Visual Basic](#)
- [Queries](#)
- [From Clause](#)
- [Select Clause](#)
- [For Each...Next Statement](#)

Statements (Visual Basic)

5/4/2018 • 2 minutes to read • [Edit Online](#)

The topics in this section contain tables of the Visual Basic declaration and executable statements, and of important lists that apply to many statements.

In This Section

[A-E Statements](#)

[F-P Statements](#)

[Q-Z Statements](#)

[Clauses](#)

[Declaration Contexts and Default Access Levels](#)

[Attribute List](#)

[Parameter List](#)

[Type List](#)

Related Sections

[Visual Basic Language Reference](#)

[Visual Basic](#)

A-E Statements

9/28/2019 • 2 minutes to read • [Edit Online](#)

The following table contains a listing of Visual Basic language statements.

AddHandler	Call	Class	Const
Continue	Declare	Delegate	Dim
Do...Loop	Else	End	End <keyword>
Enum	Erase	Error	Event
Exit			

See also

- [F-P Statements](#)
- [Q-Z Statements](#)
- [Visual Basic Language Reference](#)

AddHandler Statement

10/7/2019 • 2 minutes to read • [Edit Online](#)

Associates an event with an event handler at run time.

Syntax

```
AddHandler event, AddressOf eventhandler
```

Parts

event	The name of the event to handle.
eventhandler	The name of a procedure that handles the event.

Remarks

The `AddHandler` and `RemoveHandler` statements allow you to start and stop event handling at any time during program execution.

The signature of the `eventhandler` procedure must match the signature of the event `event`.

The `Handles` keyword and the `AddHandler` statement both allow you to specify that particular procedures handle particular events, but there are differences. The `AddHandler` statement connects procedures to events at run time. Use the `Handles` keyword when defining a procedure to specify that it handles a particular event. For more information, see [Handles](#).

NOTE

For custom events, the `AddHandler` statement invokes the event's `AddHandler` accessor. For more information on custom events, see [Event Statement](#).

Example

```
Sub TestEvents()
    Dim Obj As New Class1
    ' Associate an event handler with an event.
    AddHandler Obj.Ev_Event, AddressOf EventHandler
    ' Call the method to raise the event.
    Obj.CauseSomeEvent()
    ' Stop handling events.
    RemoveHandler Obj.Ev_Event, AddressOf EventHandler
    ' This event will not be handled.
    Obj.CauseSomeEvent()
End Sub

Sub EventHandler()
    ' Handle the event.
    MsgBox("EventHandler caught event.")
End Sub

Public Class Class1
    ' Declare an event.
    Public Event Ev_Event()
    Sub CauseSomeEvent()
        ' Raise an event.
        RaiseEvent Ev_Event()
    End Sub
End Class
```

See also

- [RemoveHandler Statement](#)
- [Handles](#)
- [Event Statement](#)
- [Events](#)

Call Statement (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Transfers control to a `Function`, `Sub`, or dynamic-link library (DLL) procedure.

Syntax

```
[ Call ] procedureName [ (argumentList) ]
```

Parts

<code>procedureName</code>	Required. Name of the procedure to call.
<code>argumentList</code>	Optional. List of variables or expressions representing arguments that are passed to the procedure when it is called. Multiple arguments are separated by commas. If you include <code>argumentList</code> , you must enclose it in parentheses.

Remarks

You can use the `Call` keyword when you call a procedure. For most procedure calls, you aren't required to use this keyword.

You typically use the `Call` keyword when the called expression doesn't start with an identifier. Use of the `Call` keyword for other uses isn't recommended.

If the procedure returns a value, the `Call` statement discards it.

Example

The following code shows two examples where the `Call` keyword is necessary to call a procedure. In both examples, the called expression doesn't start with an identifier.

```
Sub TestCall()
    Call (Sub() Console.WriteLine("Hello"))()

    Call New TheClass().ShowText()
End Sub

Class TheClass
    Public Sub ShowText()
        Console.WriteLine(" World")
    End Sub
End Class
```

See also

- Function Statement
- Sub Statement
- Declare Statement
- Lambda Expressions

Class Statement (Visual Basic)

10/7/2019 • 4 minutes to read • [Edit Online](#)

Declares the name of a class and introduces the definition of the variables, properties, events, and procedures that the class comprises.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ] [ MustInherit | NotInheritable ] [ Partial ] _  
Class name [ ( Of typelist ) ]  
    [ Inherits classname ]  
    [ Implements interfacenames ]  
    [ statements ]  
End Class
```

Parts

TERM	DEFINITION
<code>attributelist</code>	Optional. See Attribute List .
<code>accessmodifier</code>	Optional. Can be one of the following: <ul style="list-style-type: none">- Public- Protected- Friend- Private- Protected Friend- Private Protected See Access levels in Visual Basic .
<code>Shadows</code>	Optional. See Shadows .
<code>MustInherit</code>	Optional. See MustInherit .
<code>NotInheritable</code>	Optional. See NotInheritable .
<code>Partial</code>	Optional. Indicates a partial definition of the class. See Partial .
<code>name</code>	Required. Name of this class. See Declared Element Names .
<code>Of</code>	Optional. Specifies that this is a generic class.
<code>typelist</code>	Required if you use the <code>Of</code> keyword. List of type parameters for this class. See Type List .
<code>Inherits</code>	Optional. Indicates that this class inherits the members of another class. See Inherits Statement .

TERM	DEFINITION
<code>classname</code>	Required if you use the Inherits Statement . The name of the class from which this class derives.
<code>Implements</code>	Optional. Indicates that this class implements the members of one or more interfaces. See Implements Statement .
<code>interfacenames</code>	Required if you use the Implements Statement . The names of the interfaces this class implements.
<code>statements</code>	Optional. Statements which define the members of this class.
<code>End Class</code>	Required. Terminates the Class definition.

Remarks

A `Class` statement defines a new data type. A *class* is a fundamental building block of object-oriented programming (OOP). For more information, see [Objects and Classes](#).

You can use `Class` only at namespace or module level. This means the *declaration context* for a class must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure or block. For more information, see [Declaration Contexts and Default Access Levels](#).

Each instance of a class has a lifetime independent of all other instances. This lifetime begins when it is created by a [New Operator](#) clause or by a function such as [CreateObject](#). It ends when all variables pointing to the instance have been set to [Nothing](#) or to instances of other classes.

Classes default to [Friend](#) access. You can adjust their access levels with the access modifiers. For more information, see [Access levels in Visual Basic](#).

Rules

- **Nesting.** You can define one class within another. The outer class is called the *containing class*, and the inner class is called a *nested class*.
- **Inheritance.** If the class uses the [Inherits Statement](#), you can specify only one base class or interface. A class cannot inherit from more than one element.

A class cannot inherit from another class with a more restrictive access level. For example, a `Public` class cannot inherit from a `Friend` class.

A class cannot inherit from a class nested within it.

- **Implementation.** If the class uses the [Implements Statement](#), you must implement every member defined by every interface you specify in `interfacenames`. An exception to this is reimplementing of a base class member. For more information, see "Reimplementation" in [Implements](#).
- **Default Property.** A class can specify at most one property as its *default property*. For more information, see [Default](#).

Behavior

- **Access Level.** Within a class, you can declare each member with its own access level. Class members default to [Public](#) access, except variables and constants, which default to [Private](#) access. When a class

has more restricted access than one of its members, the class access level takes precedence.

- **Scope.** A class is in scope throughout its containing namespace, class, structure, or module.

The scope of every class member is the entire class.

Lifetime. Visual Basic does not support static classes. The functional equivalent of a static class is provided by a module. For more information, see [Module Statement](#).

Class members have lifetimes depending on how and where they are declared. For more information, see [Lifetime in Visual Basic](#).

- **Qualification.** Code outside a class must qualify a member's name with the name of that class.

If code inside a nested class makes an unqualified reference to a programming element, Visual Basic searches for the element first in the nested class, then in its containing class, and so on out to the outermost containing element.

Classes and Modules

These elements have many similarities, but there are some important differences as well.

- **Terminology.** Previous versions of Visual Basic recognize two types of modules: *class modules* (.cls files) and *standard modules* (.bas files). The current version calls these *classes* and *modules*, respectively.
- **Shared Members.** You can control whether a member of a class is a shared or instance member.
- **Object Orientation.** Classes are object-oriented, but modules are not. You can create one or more instances of a class. For more information, see [Objects and Classes](#).

Example

The following example uses a `Class` statement to define a class and several members.

```
Class BankAccount
    Shared interestRate As Decimal
    Private accountNumber As String
    Private accountBalance As Decimal
    Public holdOnAccount As Boolean = False

    Public ReadOnly Property Balance() As Decimal
        Get
            Return accountBalance
        End Get
    End Property

    Public Sub PostInterest()
        accountBalance = accountBalance * (1 + interestRate)
    End Sub

    Public Sub PostDeposit(ByVal amountIn As Decimal)
        accountBalance = accountBalance + amountIn
    End Sub

    Public Sub PostWithdrawal(ByVal amountOut As Decimal)
        accountBalance = accountBalance - amountOut
    End Sub
End Class
```

See also

- [Objects and Classes](#)
- [Structures and Classes](#)
- [Interface Statement](#)
- [Module Statement](#)
- [Property Statement](#)
- [Object Lifetime: How Objects Are Created and Destroyed](#)
- [Generic Types in Visual Basic](#)
- [How to: Use a Generic Class](#)

Const Statement (Visual Basic)

10/18/2019 • 4 minutes to read • [Edit Online](#)

Declares and defines one or more constants.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ]
Const constantlist
```

Parts

attributelist

Optional. List of attributes that apply to all the constants declared in this statement. See [Attribute List](#) in angle brackets ("<" and ">").

accessmodifier

Optional. Use this to specify what code can access these constants. Can be [Public](#), [Protected](#), [Friend](#), [Protected Friend](#), [Private](#), or [Private Protected](#).

Shadows

Optional. Use this to redeclare and hide a programming element in a base class. See [Shadows](#).

constantlist

Required. List of constants being declared in this statement.

constant [, constant ...]

Each `constant` has the following syntax and parts:

constantname [As datatype] = initializer

PART	DESCRIPTION
<code>constantname</code>	Required. Name of the constant. See Declared Element Names .
<code>datatype</code>	Required if <code>Option Strict</code> is <code>on</code> . Data type of the constant.
<code>initializer</code>	Required. Expression that is evaluated at compile time and assigned to the constant.

Remarks

If you have a value that never changes in your application, you can define a named constant and use it in place of a literal value. A name is easier to remember than a value. You can define the constant just once and use it in many places in your code. If in a later version you need to redefine the value, the `Const` statement is the only place you need to make a change.

You can use `Const` only at module or procedure level. This means the *declaration context* for a variable must

be a class, structure, module, procedure, or block, and cannot be a source file, namespace, or interface. For more information, see [Declaration Contexts and Default Access Levels](#).

Local constants (inside a procedure) default to public access, and you cannot use any access modifiers on them. Class and module member constants (outside any procedure) default to private access, and structure member constants default to public access. You can adjust their access levels with the access modifiers.

Rules

- **Declaration Context.** A constant declared at module level, outside any procedure, is a *member constant*; it is a member of the class, structure, or module that declares it.
A constant declared at procedure level is a *local constant*; it is local to the procedure or block that declares it.
- **Attributes.** You can apply attributes only to member constants, not to local constants. An attribute contributes information to the assembly's metadata, which is not meaningful for temporary storage such as local constants.
- **Modifiers.** By default, all constants are `Shared`, `Static`, and `ReadOnly`. You cannot use any of these keywords when declaring a constant.

At procedure level, you cannot use `Shadows` or any access modifiers to declare local constants.

- **Multiple Constants.** You can declare several constants in the same declaration statement, specifying the `constantname` part for each one. Multiple constants are separated by commas.

Data Type Rules

- **Data Types.** The `Const` statement can declare the data type of a variable. You can specify any data type or the name of an enumeration.
- **Default Type.** If you do not specify `datatype`, the constant takes the data type of `initializer`. If you specify both `datatype` and `initializer`, the data type of `initializer` must be convertible to `datatype`. If neither `datatype` nor `initializer` is present, the data type defaults to `Object`.
- **Different Types.** You can specify different data types for different constants by using a separate `As` clause for each variable you declare. However, you cannot declare several constants to be of the same type by using a common `As` clause.
- **Initialization.** You must initialize the value of every constant in `constantlist`. You use `initializer` to supply an expression to be assigned to the constant. The expression can be any combination of literals, other constants that are already defined, and enumeration members that are already defined. You can use arithmetic and logical operators to combine such elements.

You cannot use variables or functions in `initializer`. However, you can use conversion keywords such as `CByte` and `CShort`. You can also use `AscW` if you call it with a constant `String` or `Char` argument, since that can be evaluated at compile time.

Behavior

- **Scope.** Local constants are accessible only from within their procedure or block. Member constants are accessible from anywhere within their class, structure, or module.
- **Qualification.** Code outside a class, structure, or module must qualify a member constant's name with the name of that class, structure, or module. Code outside a procedure or block cannot refer to any local constants within that procedure or block.

Example

The following example uses the `Const` statement to declare constants for use in place of literal values.

```
' The following statements declare constants.  
Const maximum As Long = 459  
Public Const helpString As String = "HELP"  
Private Const startValue As Integer = 5
```

Example

If you define a constant with data type `Object`, the Visual Basic compiler gives it the type of `initializer`, instead of `Object`. In the following example, the constant `naturalLogBase` has the run-time type `Decimal`.

```
Const naturalLogBase As Object = CDec(2.7182818284)  
MsgBox("Run-time type of constant naturalLogBase is " &  
    naturalLogBase.GetType.ToString())
```

The preceding example uses the `ToString` method on the `Type` object returned by the `GetType Operator`, because `Type` cannot be converted to `String` using `cStr`.

See also

- [Asc](#)
- [AscW](#)
- [Enum Statement](#)
- [#Const Directive](#)
- [Dim Statement](#)
- [ReDim Statement](#)
- [Implicit and Explicit Conversions](#)
- [Constants and Enumerations](#)
- [Constants and Enumerations](#)
- [Type Conversion Functions](#)

Continue Statement (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Transfers control immediately to the next iteration of a loop.

Syntax

```
Continue { Do | For | While }
```

Remarks

You can transfer from inside a `Do`, `For`, or `While` loop to the next iteration of that loop. Control passes immediately to the loop condition test, which is equivalent to transferring to the `For` or `While` statement, or to the `Do` or `Loop` statement that contains the `Until` or `While` clause.

You can use `Continue` at any location in the loop that allows transfers. The rules allowing transfer of control are the same as with the [GoTo Statement](#).

For example, if a loop is totally contained within a `Try` block, a `Catch` block, or a `Finally` block, you can use `Continue` to transfer out of the loop. If, on the other hand, the `Try ... End Try` structure is contained within the loop, you cannot use `Continue` to transfer control out of the `Finally` block, and you can use it to transfer out of a `Try` or `Catch` block only if you transfer completely out of the `Try ... End Try` structure.

If you have nested loops of the same type, for example a `Do` loop within another `Do` loop, a `Continue Do` statement skips to the next iteration of the innermost `Do` loop that contains it. You cannot use `Continue` to skip to the next iteration of a containing loop of the same type.

If you have nested loops of different types, for example a `Do` loop within a `For` loop, you can skip to the next iteration of either loop by using either `Continue Do` or `Continue For`.

Example

The following code example uses the `Continue While` statement to skip to the next column of an array if a divisor is zero. The `Continue While` is inside a `For` loop. It transfers to the `While col < lastcol` statement, which is the next iteration of the innermost `While` loop that contains the `For` loop.

```
Dim row, col As Integer
Dim lastrow As Integer = 6
Dim lastcol As Integer = 10
Dim a(,) As Double = New Double(lastrow, lastcol) {}
Dim b(7) As Double
row = -1
While row < lastrow
    row += 1
    col = -1
    While col < lastcol
        col += 1
        a(row, col) = 0
        For i As Integer = 0 To b.GetUpperBound(0)
            If b(i) = col Then
                Continue While
            Else
                a(row, col) += (row + b(i)) / (col - b(i))
            End If
        Next i
    End While
End While
```

See also

- [Do...Loop Statement](#)
- [For...Next Statement](#)
- [While...End While Statement](#)
- [Try...Catch...Finally Statement](#)

Declare Statement

10/7/2019 • 8 minutes to read • [Edit Online](#)

Declares a reference to a procedure implemented in an external file.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ] [ Overloads ] _
Declare [ charsetmodifier ] [ Sub ] name Lib "libname" _
[ Alias "aliasname" ] [ ([ parameterlist ]) ]
' -or-
[ <attributelist> ] [ accessmodifier ] [ Shadows ] [ Overloads ] _
Declare [ charsetmodifier ] [ Function ] name Lib "libname" _
[ Alias "aliasname" ] [ ([ parameterlist ]) ] [ As returntype ]
```

Parts

TERM	DEFINITION
<code>attributelist</code>	Optional. See Attribute List .
<code>accessmodifier</code>	Optional. Can be one of the following: <ul style="list-style-type: none">- Public- Protected- Friend- Private- Protected Friend- Private Protected See Access levels in Visual Basic .
<code>Shadows</code>	Optional. See Shadows .
<code>charsetmodifier</code>	Optional. Specifies character set and file search information. Can be one of the following: <ul style="list-style-type: none">- Ansi (default)- Unicode- Auto
<code>Sub</code>	Optional, but either <code>Sub</code> or <code>Function</code> must appear. Indicates that the external procedure does not return a value.
<code>Function</code>	Optional, but either <code>Sub</code> or <code>Function</code> must appear. Indicates that the external procedure returns a value.
<code>name</code>	Required. Name of this external reference. For more information, see Declared Element Names .

TERM	DEFINITION
<code>Lib</code>	Required. Introduces a <code>Lib</code> clause, which identifies the external file (DLL or code resource) that contains an external procedure.
<code>libname</code>	Required. Name of the file that contains the declared procedure.
<code>Alias</code>	Optional. Indicates that the procedure being declared cannot be identified within its file by the name specified in <code>name</code> . You specify its identification in <code>aliasname</code> .
<code>aliasname</code>	Required if you use the <code>Alias</code> keyword. String that identifies the procedure in one of two ways: The entry point name of the procedure within its file, within quotes (<code>" "</code>) -or- A number sign (<code>#</code>) followed by an integer specifying the ordinal number of the procedure's entry point within its file
<code>parameterlist</code>	Required if the procedure takes parameters. See Parameter List .
<code>returntype</code>	Required if <code>Function</code> is specified and <code>Option Strict</code> is <code>On</code> . Data type of the value returned by the procedure.

Remarks

Sometimes you need to call a procedure defined in a file (such as a DLL or code resource) outside your project. When you do this, the Visual Basic compiler does not have access to the information it needs to call the procedure correctly, such as where the procedure is located, how it is identified, its calling sequence and return type, and the string character set it uses. The `Declare` statement creates a reference to an external procedure and supplies this necessary information.

You can use `Declare` only at module level. This means the *declaration context* for an external reference must be a class, structure, or module, and cannot be a source file, namespace, interface, procedure, or block. For more information, see [Declaration Contexts and Default Access Levels](#).

External references default to `Public` access. You can adjust their access levels with the access modifiers.

Rules

- **Attributes.** You can apply attributes to an external reference. Any attribute you apply has effect only in your project, not in the external file.
- **Modifiers.** External procedures are implicitly `Shared`. You cannot use the `Shared` keyword when declaring an external reference, and you cannot alter its shared status.

An external procedure cannot participate in overriding, implement interface members, or handle events. Accordingly, you cannot use the `Overrides`, `Overridable`, `NotOverridable`, `MustOverride`, `Implements`, or `Handles` keyword in a `Declare` statement.

- **External Procedure Name.** You do not have to give this external reference the same name (in `name`) as the procedure's entry-point name within its external file (`aliasname`). You can use an `Alias` clause to specify the entry-point name. This can be useful if the external procedure has the same name as a Visual Basic reserved modifier or a variable, procedure, or any other programming element in the same scope.

NOTE

Entry-point names in most DLLs are case-sensitive.

- **External Procedure Number.** Alternatively, you can use an `Alias` clause to specify the ordinal number of the entry point within the export table of the external file. To do this, you begin `aliasname` with a number sign (`#`). This can be useful if any character in the external procedure name is not allowed in Visual Basic, or if the external file exports the procedure without a name.

Data Type Rules

- **Parameter Data Types.** If `Option Strict` is `On`, you must specify the data type of each parameter in `parameterlist`. This can be any data type or the name of an enumeration, structure, class, or interface. Within `parameterlist`, you use an `As` clause to specify the data type of the argument to be passed to each parameter.

NOTE

If the external procedure was not written for the .NET Framework, you must take care that the data types correspond. For example, if you declare an external reference to a Visual Basic 6.0 procedure with an `Integer` parameter (16 bits in Visual Basic 6.0), you must identify the corresponding argument as `Short` in the `Declare` statement, because that is the 16-bit integer type in Visual Basic. Similarly, `Long` has a different data width in Visual Basic 6.0, and `Date` is implemented differently.

- **Return Data Type.** If the external procedure is a `Function` and `Option Strict` is `On`, you must specify the data type of the value returned to the calling code. This can be any data type or the name of an enumeration, structure, class, or interface.

NOTE

The Visual Basic compiler does not verify that your data types are compatible with those of the external procedure. If there is a mismatch, the common language runtime generates a `MarshalDirectiveException` exception at run time.

- **Default Data Types.** If `option Strict` is `off` and you do not specify the data type of a parameter in `parameterlist`, the Visual Basic compiler converts the corresponding argument to the `Object` **Data Type**. Similarly, if you do not specify `returntype`, the compiler takes the return data type to be `Object`.

NOTE

Because you are dealing with an external procedure that might have been written on a different platform, it is dangerous to make any assumptions about data types or to allow them to default. It is much safer to specify the data type of every parameter and of the return value, if any. This also improves the readability of your code.

Behavior

- **Scope.** An external reference is in scope throughout its class, structure, or module.
- **Lifetime.** An external reference has the same lifetime as the class, structure, or module in which it is declared.
- **Calling an External Procedure.** You call an external procedure the same way you call a [Function](#) or [Sub](#) procedure—by using it in an expression if it returns a value, or by specifying it in a [Call Statement](#) if it does not return a value.

You pass arguments to the external procedure exactly as specified by [parameterlist](#) in the [Declare](#) statement. Do not take into account how the parameters were originally declared in the external file. Similarly, if there is a return value, use it exactly as specified by [returntype](#) in the [Declare](#) statement.

- **Character Sets.** You can specify in [charsetmodifier](#) how Visual Basic should marshal strings when it calls the external procedure. The [Ansi](#) modifier directs Visual Basic to marshal all strings to ANSI values, and the [Unicode](#) modifier directs it to marshal all strings to Unicode values. The [Auto](#) modifier directs Visual Basic to marshal strings according to .NET Framework rules based on the external reference [name](#), or [aliasname](#) if specified. The default value is [Ansi](#).

[charsetmodifier](#) also specifies how Visual Basic should look up the external procedure within its external file. [Ansi](#) and [Unicode](#) both direct Visual Basic to look it up without modifying its name during the search. [Auto](#) directs Visual Basic to determine the base character set of the run-time platform and possibly modify the external procedure name, as follows:

- On an ANSI platform, such as Windows 95, Windows 98, or Windows Millennium Edition, first look up the external procedure with no name modification. If that fails, append "A" to the end of the external procedure name and look it up again.
 - On a Unicode platform, such as Windows NT, Windows 2000, or Windows XP, first look up the external procedure with no name modification. If that fails, append "W" to the end of the external procedure name and look it up again.
- **Mechanism.** Visual Basic uses the .NET Framework *platform invoke* (PInvoke) mechanism to resolve and access external procedures. The [Declare](#) statement and the [DllImportAttribute](#) class both use this mechanism automatically, and you do not need any knowledge of PInvoke. For more information, see [Walkthrough: Calling Windows APIs](#).

IMPORTANT

If the external procedure runs outside the common language runtime (CLR), it is *unmanaged code*. When you call such a procedure, for example a Windows API function or a COM method, you might expose your application to security risks. For more information, see [Secure Coding Guidelines for Unmanaged Code](#).

Example

The following example declares an external reference to a [Function](#) procedure that returns the current user name. It then calls the external procedure [GetUserNameA](#) as part of the [getUser](#) procedure.

```

Declare Function GetUserName Lib "advapi32.dll" Alias "GetUserNameA" (
    ByVal lpBuffer As String, ByRef nSize As Integer) As Integer
Sub GetUser()
    Dim buffer As String = New String(CChar(" "), 25)
    Dim retVal As Integer = GetUserName(buffer, 25)
    Dim userName As String = Strings.Left(buffer, InStr(buffer, Chr(0)) - 1)
    MsgBox(userName)
End Sub

```

Example

The [DllImportAttribute](#) provides an alternative way of using functions in unmanaged code. The following example declares an imported function without using a `Declare` statement.

```

' Add an Imports statement at the top of the class, structure, or
' module that uses the DllImport attribute.
Imports System.Runtime.InteropServices

```

```

<DllImportAttribute("kernel32.dll", EntryPoint:="MoveFileW",
    SetLastError:=True, CharSet:=CharSet.Unicode,
    ExactSpelling:=True,
    CallingConvention:=CallingConvention.StdCall)>
Public Shared Function MoveFile(ByVal src As String,
    ByVal dst As String) As Boolean
    ' This function copies a file from the path src to the path dst.
    ' Leave this function empty. The DllImport attribute forces calls
    ' to MoveFile to be forwarded to MoveFileW in KERNEL32.DLL.
End Function

```

See also

- [LastDIIError](#)
- [Imports Statement \(.NET Namespace and Type\)](#)
- [AddressOf Operator](#)
- [Function Statement](#)
- [Sub Statement](#)
- [Parameter List](#)
- [Call Statement](#)
- [Walkthrough: Calling Windows APIs](#)

Delegate Statement

10/18/2019 • 4 minutes to read • [Edit Online](#)

Used to declare a delegate. A delegate is a reference type that refers to a `Shared` method of a type or to an instance method of an object. Any procedure with matching parameter and return types can be used to create an instance of this delegate class. The procedure can then later be invoked by means of the delegate instance.

Syntax

```
[ <attrlist> ] [ accessmodifier ] _  
[ Shadows ] Delegate [ Sub | Function ] name [( Of typeparamlist )] [([ parameterlist ]) ] [ As type ]
```

Parts

TERM	DEFINITION
<code>attrlist</code>	Optional. List of attributes that apply to this delegate. Multiple attributes are separated by commas. You must enclose the Attribute List in angle brackets ("`<`" and "`>`").
<code>accessmodifier</code>	Optional. Specifies what code can access the delegate. Can be one of the following: <ul style="list-style-type: none">- Public. Any code that can access the element that declares the delegate can access it.- Protected. Only code within the delegate's class or a derived class can access it.- Friend. Only code within the same assembly can access the delegate.- Private. Only code within the element that declares the delegate can access it.- Protected Friend Only code within the delegate's class, a derived class, or the same assembly can access the delegate.- Private Protected Only code within the delegate's class or in a derived class in the same assembly can access the delegate.
<code>Shadows</code>	Optional. Indicates that this delegate redeclares and hides an identically named programming element, or set of overloaded elements, in a base class. You can shadow any kind of declared element with any other kind. A shadowed element is unavailable from within the derived class that shadows it, except from where the shadowing element is inaccessible. For example, if a <code>Private</code> element shadows a base class element, code that does not have permission to access the <code>Private</code> element accesses the base class element instead.
<code>Sub</code>	Optional, but either <code>Sub</code> or <code>Function</code> must appear. Declares this procedure as a delegate <code>Sub</code> procedure that does not return a value.

TERM	DEFINITION
Function	Optional, but either Sub or Function must appear. Declares this procedure as a delegate Function procedure that returns a value.
name	Required. Name of the delegate type; follows standard variable naming conventions.
typeparamlist	Optional. List of type parameters for this delegate. Multiple type parameters are separated by commas. Optionally, each type parameter can be declared variant by using In and Out generic modifiers. You must enclose the Type List in parentheses and introduce it with the Of keyword.
parameterlist	Optional. List of parameters that are passed to the procedure when it is called. You must enclose the Parameter List in parentheses.
type	Required if you specify a Function procedure. Data type of the return value.

Remarks

The Delegate statement defines the parameter and return types of a delegate class. Any procedure with matching parameters and return types can be used to create an instance of this delegate class. The procedure can then later be invoked by means of the delegate instance, by calling the delegate's Invoke method.

Delegates can be declared at the namespace, module, class, or structure level, but not within a procedure.

Each delegate class defines a constructor that is passed the specification of an object method. An argument to a delegate constructor must be a reference to a method, or a lambda expression.

To specify a reference to a method, use the following syntax:

```
AddressOf [ expression ].methodname
```

The compile-time type of the expression must be the name of a class or an interface that contains a method of the specified name whose signature matches the signature of the delegate class. The methodname can be either a shared method or an instance method. The methodname is not optional, even if you create a delegate for the default method of the class.

To specify a lambda expression, use the following syntax:

```
Function ([ parm As type , parm2 As type2 , ...]) expression
```

The signature of the function must match that of the delegate type. For more information about lambda expressions, see [Lambda Expressions](#).

For more information about delegates, see [Delegates](#).

Example

The following example uses the Delegate statement to declare a delegate for operating on two numbers and returning a number. The DelegateTest method takes an instance of a delegate of this type and uses it to operate on pairs of numbers.

```

Delegate Function MathOperator(
    ByVal x As Double,
    ByVal y As Double
) As Double

Function AddNumbers(
    ByVal x As Double,
    ByVal y As Double
) As Double
    Return x + y
End Function

Function SubtractNumbers(
    ByVal x As Double,
    ByVal y As Double
) As Double
    Return x - y
End Function

Sub DelegateTest(
    ByVal x As Double,
    ByVal op As MathOperator,
    ByVal y As Double
)
    Dim ret As Double
    ret = op.Invoke(x, y) ' Call the method.
    MsgBox(ret)
End Sub

Protected Sub Test()
    DelegateTest(5, AddressOf AddNumbers, 3)
    DelegateTest(9, AddressOf SubtractNumbers, 3)
End Sub

```

See also

- [AddressOf Operator](#)
- [Of](#)
- [Delegates](#)
- [How to: Use a Generic Class](#)
- [Generic Types in Visual Basic](#)
- [Covariance and Contravariance](#)
- [In](#)
- [Out](#)

Dim Statement (Visual Basic)

10/18/2019 • 12 minutes to read • [Edit Online](#)

Declares and allocates storage space for one or more variables.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [[ Shared ] [ Shadows ] | [ Static ]] [ ReadOnly ]
Dim [ WithEvents ] variablelist
```

Parts

- `attributelist`

Optional. See [Attribute List](#).

- `accessmodifier`

Optional. Can be one of the following:

- [Public](#)
- [Protected](#)
- [Friend](#)
- [Private](#)
- [Protected Friend](#)
- [Private Protected](#)

See [Access levels in Visual Basic](#).

- `Shared`

Optional. See [Shared](#).

- `Shadows`

Optional. See [Shadows](#).

- `Static`

Optional. See [Static](#).

- `ReadOnly`

Optional. See [ReadOnly](#).

- `WithEvents`

Optional. Specifies that these are object variables that refer to instances of a class that can raise events.
See [WithEvents](#).

- `variablelist`

Required. List of variables being declared in this statement.

```
variable [ , variable ... ]
```

Each `variable` has the following syntax and parts:

```
variablename [ ( [ boundslist ] ) ] [ As [ New ] datatype [ With {  
[ .propertynname = propinitializer [ , ... ] } ] ] [ = initializer ] ]
```

PART	DESCRIPTION
<code>variablename</code>	Required. Name of the variable. See Declared Element Names .
<code>boundslist</code>	Optional. List of bounds of each dimension of an array variable.
<code>New</code>	Optional. Creates a new instance of the class when the <code>Dim</code> statement runs.
<code>datatype</code>	Optional. Data type of the variable.
<code>With</code>	Optional. Introduces the object initializer list.
<code>propertynname</code>	Optional. The name of a property in the class you are making an instance of.
<code>propinitializer</code>	Required after <code>propertynname</code> =. The expression that is evaluated and assigned to the property name.
<code>initializer</code>	Optional if <code>New</code> is not specified. Expression that is evaluated and assigned to the variable when it is created.

Remarks

The Visual Basic compiler uses the `Dim` statement to determine the variable's data type and other information, such as what code can access the variable. The following example declares a variable to hold an `Integer` value.

```
Dim numberOfStudents As Integer
```

You can specify any data type or the name of an enumeration, structure, class, or interface.

```
Dim finished As Boolean  
Dim monitorBox As System.Windows.Forms.Form
```

For a reference type, you use the `New` keyword to create a new instance of the class or structure that is specified by the data type. If you use `New`, you do not use an initializer expression. Instead, you supply arguments, if they are required, to the constructor of the class from which you are creating the variable.

```
Dim bottomLabel As New System.Windows.Forms.Label
```

You can declare a variable in a procedure, block, class, structure, or module. You cannot declare a variable in a source file, namespace, or interface. For more information, see [Declaration Contexts and Default Access Levels](#).

A variable that is declared at module level, outside any procedure, is a *member variable* or *field*. Member variables are in scope throughout their class, structure, or module. A variable that is declared at procedure level is a *local variable*. Local variables are in scope only within their procedure or block.

The following access modifiers are used to declare variables outside a procedure: `Public`, `Protected`, `Friend`, `Protected Friend`, and `Private`. For more information, see [Access levels in Visual Basic](#).

The `Dim` keyword is optional and usually omitted if you specify any of the following modifiers: `Public`, `Protected`, `Friend`, `Protected Friend`, `Private`, `Shared`, `Shadows`, `Static`, `ReadOnly`, or `WithEvents`.

```
Public maximumAllowed As Double
Protected Friend currentUser As String
Private salary As Decimal
Static runningTotal As Integer
```

If `Option Explicit` is on (the default), the compiler requires a declaration for every variable you use. For more information, see [Option Explicit Statement](#).

Specifying an Initial Value

You can assign a value to a variable when it is created. For a value type, you use an *initializer* to supply an expression to be assigned to the variable. The expression must evaluate to a constant that can be calculated at compile time.

```
Dim quantity As Integer = 10
Dim message As String = "Just started"
```

If an initializer is specified and a data type is not specified in an `As` clause, *type inference* is used to infer the data type from the initializer. In the following example, both `num1` and `num2` are strongly typed as integers. In the second declaration, type inference infers the type from the value 3.

```
' Use explicit typing.
Dim num1 As Integer = 3

' Use local type inference.
Dim num2 = 3
```

Type inference applies at the procedure level. It does not apply outside a procedure in a class, structure, module, or interface. For more information about type inference, see [Option Infer Statement](#) and [Local Type Inference](#).

For information about what happens when a data type or initializer is not specified, see [Default Data Types and Values](#) later in this topic.

You can use an *object initializer* to declare instances of named and anonymous types. The following code creates an instance of a `Student` class and uses an object initializer to initialize properties.

```
Dim student1 As New Student With {.First = "Michael",
                                  .Last = "Tucker"}
```

For more information about object initializers, see [How to: Declare an Object by Using an Object](#)

Declaring Multiple Variables

You can declare several variables in one declaration statement, specifying the variable name for each one, and following each array name with parentheses. Multiple variables are separated by commas.

```
Dim lastTime, nextTime, allTimes() As Date
```

If you declare more than one variable with one `As` clause, you cannot supply an initializer for that group of variables.

You can specify different data types for different variables by using a separate `As` clause for each variable you declare. Each variable takes the data type specified in the first `As` clause encountered after its `variablename` part.

```
Dim a, b, c As Single, x, y As Double, i As Integer  
' a, b, and c are all Single; x and y are both Double
```

Arrays

You can declare a variable to hold an *array*, which can hold multiple values. To specify that a variable holds an array, follow its `variablename` immediately with parentheses. For more information about arrays, see [Arrays](#).

You can specify the lower and upper bound of each dimension of an array. To do this, include a `boundslist` inside the parentheses. For each dimension, the `boundslist` specifies the upper bound and optionally the lower bound. The lower bound is always zero, whether you specify it or not. Each index can vary from zero through its upper bound value.

The following two statements are equivalent. Each statement declares an array of 21 `Integer` elements. When you access the array, the index can vary from 0 through 20.

```
Dim totals(20) As Integer  
Dim totals(0 To 20) As Integer
```

The following statement declares a two-dimensional array of type `Double`. The array has 4 rows (3 + 1) of 6 columns (5 + 1) each. Note that an upper bound represents the highest possible value for the index, not the length of the dimension. The length of the dimension is the upper bound plus one.

```
Dim matrix2(3, 5) As Double
```

An array can have from 1 to 32 dimensions.

You can leave all the bounds blank in an array declaration. If you do this, the array has the number of dimensions you specify, but it is uninitialized. It has a value of `Nothing` until you initialize at least some of its elements. The `Dim` statement must specify bounds either for all dimensions or for no dimensions.

```
' Declare an array with blank array bounds.  
Dim messages() As String  
' Initialize the array.  
ReDim messages(4)
```

If the array has more than one dimension, you must include commas between the parentheses to indicate the number of dimensions.

```
Dim oneDimension(), twoDimensions(,), threeDimensions(,,) As Byte
```

You can declare a *zero-length array* by declaring one of the array's dimensions to be -1. A variable that holds a zero-length array does not have the value `Nothing`. Zero-length arrays are required by certain common language runtime functions. If you try to access such an array, a runtime exception occurs. For more information, see [Arrays](#).

You can initialize the values of an array by using an array literal. To do this, surround the initialization values with braces (`{}`).

```
Dim longArray() As Long = {0, 1, 2, 3}
```

For multidimensional arrays, the initialization for each separate dimension is enclosed in braces in the outer dimension. The elements are specified in row-major order.

```
Dim twoDimensions(,) As Integer = {{0, 1, 2}, {10, 11, 12}}
```

For more information about array literals, see [Arrays](#).

Default Data Types and Values

The following table describes the results of various combinations of specifying the data type and initializer in a `Dim` statement.

DATA TYPE SPECIFIED?	INITIALIZER SPECIFIED?	EXAMPLE	RESULT
No	No	<code>Dim qty</code>	If <code>Option Strict</code> is off (the default), the variable is set to <code>Nothing</code> . If <code>Option Strict</code> is on, a compile-time error occurs.
No	Yes	<code>Dim qty = 5</code>	If <code>Option Infer</code> is on (the default), the variable takes the data type of the initializer. See Local Type Inference . If <code>Option Infer</code> is off and <code>Option Strict</code> is off, the variable takes the data type of <code>Object</code> . If <code>Option Infer</code> is off and <code>Option Strict</code> is on, a compile-time error occurs.

DATA TYPE SPECIFIED?	INITIALIZER SPECIFIED?	EXAMPLE	RESULT
Yes	No	<code>Dim qty As Integer</code>	The variable is initialized to the default value for the data type. See the table later in this section.
Yes	Yes	<code>Dim qty As Integer = 5</code>	If the data type of the initializer is not convertible to the specified data type, a compile-time error occurs.

If you specify a data type but do not specify an initializer, Visual Basic initializes the variable to the default value for its data type. The following table shows the default initialization values.

DATA TYPE	DEFAULT VALUE
All numeric types (including <code>Byte</code> and <code>SByte</code>)	0
<code>Char</code>	Binary 0
All reference types (including <code>Object</code> , <code>String</code> , and all arrays)	<code>Nothing</code>
<code>Boolean</code>	<code>False</code>
<code>Date</code>	12:00 AM of January 1 of the year 1 (01/01/0001 12:00:00 AM)

Each element of a structure is initialized as if it were a separate variable. If you declare the length of an array but do not initialize its elements, each element is initialized as if it were a separate variable.

Static Local Variable Lifetime

A `Static` local variable has a longer lifetime than that of the procedure in which it is declared. The boundaries of the variable's lifetime depend on where the procedure is declared and whether it is `Shared`.

PROCEDURE DECLARATION	VARIABLE INITIALIZED	VARIABLE STOPS EXISTING
In a module	The first time the procedure is called	When your program stops execution
In a class or structure, procedure is <code>Shared</code>	The first time the procedure is called either on a specific instance or on the class or structure itself	When your program stops execution
In a class or structure, procedure isn't <code>Shared</code>	The first time the procedure is called on a specific instance	When the instance is released for garbage collection (GC)

Attributes and Modifiers

You can apply attributes only to member variables, not to local variables. An attribute contributes information to the assembly's metadata, which is not meaningful for temporary storage such as local variables.

At module level, you cannot use the `Static` modifier to declare member variables. At procedure level, you cannot use `Shared`, `Shadows`, `ReadOnly`, `WithEvents`, or any access modifiers to declare local variables.

You can specify what code can access a variable by supplying an `accessmodifier`. Class and module member variables (outside any procedure) default to private access, and structure member variables default to public access. You can adjust their access levels with the access modifiers. You cannot use access modifiers on local variables (inside a procedure).

You can specify `WithEvents` only on member variables, not on local variables inside a procedure. If you specify `WithEvents`, the data type of the variable must be a specific class type, not `Object`. You cannot declare an array with `WithEvents`. For more information about events, see [Events](#).

NOTE

Code outside a class, structure, or module must qualify a member variable's name with the name of that class, structure, or module. Code outside a procedure or block cannot refer to any local variables within that procedure or block.

Releasing Managed Resources

The .NET Framework garbage collector disposes of managed resources without any extra coding on your part. However, you can force the disposal of a managed resource instead of waiting for the garbage collector.

If a class holds onto a particularly valuable and scarce resource (such as a database connection or file handle), you might not want to wait until the next garbage collection to clean up a class instance that's no longer in use. A class may implement the [IDisposable](#) interface to provide a way to release resources before a garbage collection. A class that implements that interface exposes a `Dispose` method that can be called to force valuable resources to be released immediately.

The `Using` statement automates the process of acquiring a resource, executing a set of statements, and then disposing of the resource. However, the resource must implement the [IDisposable](#) interface. For more information, see [Using Statement](#).

Example

The following example declares variables by using the `Dim` statement with various options.

```
' Declare and initialize a Long variable.  
Dim startingAmount As Long = 500  
  
' Declare a variable that refers to a Button object,  
' create a Button object, and assign the Button object  
' to the variable.  
Dim switchButton As New System.Windows.Forms.Button  
  
' Declare a local variable that always retains its value,  
' even after its procedure returns to the calling code.  
Static totalSales As Double  
  
' Declare a variable that refers to an array.  
Dim highTemperature(31) As Integer  
  
' Declare and initialize an array variable that  
' holds four Boolean check values.  
Dim checkValues() As Boolean = {False, False, True, False}
```

Example

The following example lists the prime numbers between 1 and 30. The scope of local variables is described in code comments.

```
Public Sub ListPrimes()
    ' The sb variable can be accessed only
    ' within the ListPrimes procedure.
    Dim sb As New System.Text.StringBuilder()

    ' The number variable can be accessed only
    ' within the For...Next block. A different
    ' variable with the same name could be declared
    ' outside of the For...Next block.
    For number As Integer = 1 To 30
        If CheckIfPrime(number) = True Then
            sb.Append(number.ToString & " ")
        End If
    Next

    Debug.WriteLine(sb.ToString)
    ' Output: 2 3 5 7 11 13 17 19 23 29
End Sub

Private Function CheckIfPrime(ByVal number As Integer) As Boolean
    If number < 2 Then
        Return False
    Else
        ' The root and highCheck variables can be accessed
        ' only within the Else block. Different variables
        ' with the same names could be declared outside of
        ' the Else block.
        Dim root As Double = Math.Sqrt(number)
        Dim highCheck As Integer = Convert.ToInt32(Math.Truncate(root))

        ' The div variable can be accessed only within
        ' the For...Next block.
        For div As Integer = 2 To highCheck
            If number Mod div = 0 Then
                Return False
            End If
        Next

        Return True
    End If
End Function
```

Example

In the following example, the `speedValue` variable is declared at the class level. The `Private` keyword is used to declare the variable. The variable can be accessed by any procedure in the `Car` class.

```
' Create a new instance of a Car.
Dim theCar As New Car()
theCar.Accelerate(30)
theCar.Accelerate(20)
theCar.Accelerate(-5)

Debug.WriteLine(theCar.Speed.ToString)
' Output: 45
```

```
Public Class Car
    ' The speedValue variable can be accessed by
    ' any procedure in the Car class.
    Private speedValue As Integer = 0

    Public ReadOnly Property Speed() As Integer
        Get
            Return speedValue
        End Get
    End Property

    Public Sub Accelerate(ByVal speedIncrease As Integer)
        speedValue += speedIncrease
    End Sub
End Class
```

See also

- [Const Statement](#)
- [ReDim Statement](#)
- [Option Explicit Statement](#)
- [Option Infer Statement](#)
- [Option Strict Statement](#)
- [Compile Page, Project Designer \(Visual Basic\)](#)
- [Variable Declaration](#)
- [Arrays](#)
- [Object Initializers: Named and Anonymous Types](#)
- [Anonymous Types](#)
- [Object Initializers: Named and Anonymous Types](#)
- [How to: Declare an Object by Using an Object Initializer](#)
- [Local Type Inference](#)

Do...Loop Statement (Visual Basic)

10/18/2019 • 4 minutes to read • [Edit Online](#)

Repeats a block of statements while a `Boolean` condition is `True` or until the condition becomes `True`.

Syntax

```
Do { While | Until } condition
    [ statements ]
    [ Continue Do ]
    [ statements ]
    [ Exit Do ]
    [ statements ]
Loop
' -or-
Do
    [ statements ]
    [ Continue Do ]
    [ statements ]
    [ Exit Do ]
    [ statements ]
Loop { While | Until } condition
```

Parts

TERM	DEFINITION
<code>Do</code>	Required. Starts the definition of the <code>Do</code> loop.
<code>While</code>	Required unless <code>Until</code> is used. Repeat the loop until <code>condition</code> is <code>False</code> .
<code>Until</code>	Required unless <code>While</code> is used. Repeat the loop until <code>condition</code> is <code>True</code> .
<code>condition</code>	Optional. <code>Boolean</code> expression. If <code>condition</code> is <code>Nothing</code> , Visual Basic treats it as <code>False</code> .
<code>statements</code>	Optional. One or more statements that are repeated while, or until, <code>condition</code> is <code>True</code> .
<code>Continue Do</code>	Optional. Transfers control to the next iteration of the <code>Do</code> loop.
<code>Exit Do</code>	Optional. Transfers control out of the <code>Do</code> loop.
<code>Loop</code>	Required. Terminates the definition of the <code>Do</code> loop.

Remarks

Use a `Do...Loop` structure when you want to repeat a set of statements an indefinite number of times, until a

condition is satisfied. If you want to repeat the statements a set number of times, the [For...Next Statement](#) is usually a better choice.

You can use either `While` or `Until` to specify `condition`, but not both.

You can test `condition` only one time, at either the start or the end of the loop. If you test `condition` at the start of the loop (in the `Do` statement), the loop might not run even one time. If you test at the end of the loop (in the `Loop` statement), the loop always runs at least one time.

The condition usually results from a comparison of two values, but it can be any expression that evaluates to a [Boolean Data Type](#) value (`True` or `False`). This includes values of other data types, such as numeric types, that have been converted to `Boolean`.

You can nest `Do` loops by putting one loop within another. You can also nest different kinds of control structures within each other. For more information, see [Nested Control Structures](#).

NOTE

The `Do...Loop` structure gives you more flexibility than the [While...End While Statement](#) because it enables you to decide whether to end the loop when `condition` stops being `True` or when it first becomes `True`. It also enables you to test `condition` at either the start or the end of the loop.

Exit Do

The [Exit Do](#) statement can provide an alternative way to exit a `Do...Loop`. `Exit Do` transfers control immediately to the statement that follows the `Loop` statement.

`Exit Do` is often used after some condition is evaluated, for example in an `If...Then...Else` structure. You might want to exit a loop if you detect a condition that makes it unnecessary or impossible to continue iterating, such as an erroneous value or a termination request. One use of `Exit Do` is to test for a condition that could cause an *endless loop*, which is a loop that could run a large or even infinite number of times. You can use `Exit Do` to escape the loop.

You can include any number of `Exit Do` statements anywhere in a `Do...Loop`.

When used within nested `Do` loops, `Exit Do` transfers control out of the innermost loop and into the next higher level of nesting.

Example

In the following example, the statements in the loop continue to run until the `index` variable is greater than 10. The `Until` clause is at the end of the loop.

```
Dim index As Integer = 0
Do
    Debug.WriteLine(index.ToString & " ")
    index += 1
Loop Until index > 10

Debug.WriteLine("")
' Output: 0 1 2 3 4 5 6 7 8 9 10
```

Example

The following example uses a `While` clause instead of an `Until` clause, and `condition` is tested at the start of

the loop instead of at the end.

```
Dim index As Integer = 0
Do While index <= 10
    Debug.WriteLine(index.ToString & " ")
    index += 1
Loop

Debug.WriteLine("")
' Output: 0 1 2 3 4 5 6 7 8 9 10
```

Example

In the following example, `condition` stops the loop when the `index` variable is greater than 100. The `If` statement in the loop, however, causes the `Exit Do` statement to stop the loop when the `index` variable is greater than 10.

```
Dim index As Integer = 0
Do While index <= 100
    If index > 10 Then
        Exit Do
    End If

    Debug.WriteLine(index.ToString & " ")
    index += 1
Loop

Debug.WriteLine("")
' Output: 0 1 2 3 4 5 6 7 8 9 10
```

Example

The following example reads all lines in a text file. The `OpenText` method opens the file and returns a `StreamReader` that reads the characters. In the `Do...Loop` condition, the `Peek` method of the `StreamReader` determines whether there are any additional characters.

```
Private Sub ShowText(ByVal textFilePath As String)
    If System.IO.File.Exists(textFilePath) = False Then
        Debug.WriteLine("File Not Found: " & textFilePath)
    Else
        Dim sr As System.IO.StreamReader = System.IO.File.OpenText(textFilePath)

        Do While sr.Peek() >= 0
            Debug.WriteLine(sr.ReadLine())
        Loop

        sr.Close()
    End If
End Sub
```

See also

- [Loop Structures](#)
- [For...Next Statement](#)
- [Boolean Data Type](#)
- [Nested Control Structures](#)

- Exit Statement
- While...End While Statement

Else Statement (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Introduces a group of statements to be run or compiled if no other conditional group of statements has been run or compiled.

Remarks

The `Else` keyword can be used in these contexts:

[If...Then...Else Statement](#)

[Select...Case Statement](#)

[#If...Then...#Else Directive](#)

See also

- [Keywords](#)

End Statement

10/18/2019 • 2 minutes to read • [Edit Online](#)

Terminates execution immediately.

Syntax

```
End
```

Remarks

You can place the `End` statement anywhere in a procedure to force the entire application to stop running. `End` closes any files opened with an `Open` statement and clears all the application's variables. The application closes as soon as there are no other programs holding references to its objects and none of its code is running.

NOTE

The `End` statement stops code execution abruptly, and does not invoke the `Dispose` or `Finalize` method, or any other Visual Basic code. Object references held by other programs are invalidated. If an `End` statement is encountered within a `Try` or `Catch` block, control does not pass to the corresponding `Finally` block.

The `Stop` statement suspends execution, but unlike `End`, it does not close any files or clear any variables, unless it is encountered in a compiled executable (.exe) file.

Because `End` terminates your application without attending to any resources that might be open, you should try to close down cleanly before using it. For example, if your application has any forms open, you should close them before control reaches the `End` statement.

You should use `End` sparingly, and only when you need to stop immediately. The normal ways to terminate a procedure ([Return Statement](#) and [Exit Statement](#)) not only close down the procedure cleanly but also give the calling code the opportunity to close down cleanly. A console application, for example, can simply `Return` from the `Main` procedure.

IMPORTANT

The `End` statement calls the `Exit` method of the `Environment` class in the `System` namespace. `Exit` requires that you have `UnmanagedCode` permission. If you do not, a `SecurityException` error occurs.

When followed by an additional keyword, `End <keyword> Statement` delineates the end of the definition of the appropriate procedure or block. For example, `End Function` terminates the definition of a `Function` procedure.

Example

The following example uses the `End` statement to terminate code execution if the user requests it.

```
Sub Form_Load()
    Dim answer As MsgBoxResult
    answer = MsgBox("Do you want to quit now?", MsgBoxStyle.YesNo)
    If answer = MsgBoxResult.Yes Then
        MsgBox("Terminating program")
        End
    End If
End Sub
```

Smart Device Developer Notes

This statement is not supported.

See also

- [SecurityPermissionFlag](#)
- [Stop Statement](#)
- [End <keyword> Statement](#)

End <keyword> Statement (Visual Basic)

1/30/2019 • 2 minutes to read • [Edit Online](#)

When followed by an additional keyword, terminates the definition of the statement block introduced by that keyword.

Syntax

```
End AddHandler  
End Class  
End Enum  
End Event  
End Function  
End Get  
End If  
End Interface  
End Module  
End Namespace  
End Operator  
End Property  
End RaiseEvent  
End RemoveHandler  
End Select  
End Set  
End Structure  
End Sub  
End SyncLock  
End Try  
End While  
End With
```

Parts

PART	DESCRIPTION
<code>End</code>	Required. Terminates the definition of the programming element.
<code>AddHandler</code>	Required to terminate an <code>AddHandler</code> accessor begun by a matching <code>AddHandler</code> statement in a custom Event Statement .
<code>Class</code>	Required to terminate a class definition begun by a matching Class Statement .
<code>Enum</code>	Required to terminate an enumeration definition begun by a matching Enum Statement .
<code>Event</code>	Required to terminate a <code>Custom</code> event definition begun by a matching Event Statement .

PART	DESCRIPTION
Function	Required to terminate a <code>Function</code> procedure definition begun by a matching Function Statement . If execution encounters an <code>End Function</code> statement, control returns to the calling code.
Get	Required to terminate a <code>Property</code> procedure definition begun by a matching Get Statement . If execution encounters an <code>End Get</code> statement, control returns to the statement requesting the property's value.
If	Required to terminate an <code>If ... Then ... Else</code> block definition begun by a matching <code>If</code> statement. See If...Then...Else Statement .
Interface	Required to terminate an interface definition begun by a matching Interface Statement .
Module	Required to terminate a module definition begun by a matching Module Statement .
Namespace	Required to terminate a namespace definition begun by a matching Namespace Statement .
Operator	Required to terminate an operator definition begun by a matching Operator Statement .
Property	Required to terminate a property definition begun by a matching Property Statement .
RaiseEvent	Required to terminate a <code>RaiseEvent</code> accessor begun by a matching <code>RaiseEvent</code> statement in a custom Event Statement .
RemoveHandler	Required to terminate a <code>RemoveHandler</code> accessor begun by a matching <code>RemoveHandler</code> statement in a custom Event Statement .
Select	Required to terminate a <code>Select ... Case</code> block definition begun by a matching <code>Select</code> statement. See Select...Case Statement .
Set	Required to terminate a <code>Property</code> procedure definition begun by a matching Set Statement . If execution encounters an <code>End Set</code> statement, control returns to the statement setting the property's value.
Structure	Required to terminate a structure definition begun by a matching Structure Statement .
Sub	Required to terminate a <code>Sub</code> procedure definition begun by a matching Sub Statement . If execution encounters an <code>End Sub</code> statement, control returns to the calling code.

PART	DESCRIPTION
SyncLock	Required to terminate a <code>SyncLock</code> block definition begun by a matching <code>SyncLock</code> statement. See SyncLock Statement .
Try	Required to terminate a <code>Try ... Catch ... Finally</code> block definition begun by a matching <code>Try</code> statement. See Try...Catch...Finally Statement .
While	Required to terminate a <code>While</code> loop definition begun by a matching <code>While</code> statement. See While..End While Statement .
With	Required to terminate a <code>With</code> block definition begun by a matching <code>With</code> statement. See With...End With Statement .

Directives

When preceded by a number sign (#), the `End` keyword terminates a preprocessing block introduced by the corresponding directive.

```
#End ExternalSource
#End If
#End Region
```

PART	DESCRIPTION
#End	Required. Terminates the definition of the preprocessing block.
ExternalSource	Required to terminate an external source block begun by a matching #ExternalSource Directive .
If	Required to terminate a conditional compilation block begun by a matching <code>#If</code> directive. See #If...Then...#Else Directives .
Region	Required to terminate a source region block begun by a matching #Region Directive .

Remarks

The [End Statement](#), without an additional keyword, terminates execution immediately.

Smart Device Developer Notes

The `End` statement, without an additional keyword, is not supported.

See also

- [End Statement](#)

Enum Statement (Visual Basic)

10/18/2019 • 7 minutes to read • [Edit Online](#)

Declares an enumeration and defines the values of its members.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ]
  Enum enumerationname [ As datatype ]
    memberlist
  End Enum
```

Parts

- **attributelist**

Optional. List of attributes that apply to this enumeration. You must enclose the [attribute list](#) in angle brackets ("<" and ">").

The [FlagsAttribute](#) attribute indicates that the value of an instance of the enumeration can include multiple enumeration members, and that each member represents a bit field in the enumeration value.

- **accessmodifier**

Optional. Specifies what code can access this enumeration. Can be one of the following:

- [Public](#)
- [Protected](#)
- [Friend](#)
- [Private](#)
- [Protected Friend](#)
- [Private Protected](#)

- **Shadows**

Optional. Specifies that this enumeration redeclares and hides an identically named programming element, or set of overloaded elements, in a base class. You can specify [Shadows](#) only on the enumeration itself, not on any of its members.

- **enumerationname**

Required. Name of the enumeration. For information on valid names, see [Declared Element Names](#).

- **datatype**

Optional. Data type of the enumeration and all its members.

- **memberlist**

Required. List of member constants being declared in this statement. Multiple members appear on individual source code lines.

Each `member` has the following syntax and parts: `[<attribute list>] member name [= initializer]`

PART	DESCRIPTION
<code>membername</code>	Required. Name of this member.
<code>initializer</code>	Optional. Expression that is evaluated at compile time and assigned to this member.

- `End` `Enum`

Terminates the `Enum` block.

Remarks

If you have a set of unchanging values that are logically related to each other, you can define them together in an enumeration. This provides meaningful names for the enumeration and its members, which are easier to remember than their values. You can then use the enumeration members in many places in your code.

The benefits of using enumerations include the following:

- Reduces errors caused by transposing or mistyping numbers.
- Makes it easy to change values in the future.
- Makes code easier to read, which means it is less likely that errors will be introduced.
- Ensures forward compatibility. If you use enumerations, your code is less likely to fail if in the future someone changes the values corresponding to the member names.

An enumeration has a name, an underlying data type, and a set of members. Each member represents a constant.

An enumeration declared at class, structure, module, or interface level, outside any procedure, is a *member enumeration*. It is a member of the class, structure, module, or interface that declares it.

Member enumerations can be accessed from anywhere within their class, structure, module, or interface. Code outside a class, structure, or module must qualify a member enumeration's name with the name of that class, structure, or module. You can avoid the need to use fully qualified names by adding an [Imports](#) statement to the source file.

An enumeration declared at namespace level, outside any class, structure, module, or interface, is a member of the namespace in which it appears.

The *declaration context* for an enumeration must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure. For more information, see [Declaration Contexts and Default Access Levels](#).

You can apply attributes to an enumeration as a whole, but not to its members individually. An attribute contributes information to the assembly's metadata.

Data Type

The `Enum` statement can declare the data type of an enumeration. Each member takes the enumeration's data type. You can specify `Byte`, `Integer`, `Long`, `SByte`, `Short`, `UInteger`, `ULong`, or `UShort`.

If you do not specify `datatype` for the enumeration, each member takes the data type of its `initializer`. If you specify both `datatype` and `initializer`, the data type of `initializer` must be convertible to `datatype`. If

neither `datatype` nor `initializer` is present, the data type defaults to `Integer`.

Initializing Members

The `Enum` statement can initialize the contents of selected members in `memberlist`. You use `initializer` to supply an expression to be assigned to the member.

If you do not specify `initializer` for a member, Visual Basic initializes it either to zero (if it is the first `member` in `memberlist`), or to a value greater by one than that of the immediately preceding `member`.

The expression supplied in each `initializer` can be any combination of literals, other constants that are already defined, and enumeration members that are already defined, including a previous member of this enumeration. You can use arithmetic and logical operators to combine such elements.

You cannot use variables or functions in `initializer`. However, you can use conversion keywords such as `CByte` and `CShort`. You can also use `AscW` if you call it with a constant `String` or `Char` argument, since that can be evaluated at compile time.

Enumerations cannot have floating-point values. If a member is assigned a floating-point value and `Option Strict` is set to on, a compiler error occurs. If `Option Strict` is off, the value is automatically converted to the `Enum` type.

If the value of a member exceeds the allowable range for the underlying data type, or if you initialize any member to the maximum value allowed by the underlying data type, the compiler reports an error.

Modifiers

Class, structure, module, and interface member enumerations default to public access. You can adjust their access levels with the access modifiers. Namespace member enumerations default to friend access. You can adjust their access levels to public, but not to private or protected. For more information, see [Access levels in Visual Basic](#).

All enumeration members have public access, and you cannot use any access modifiers on them. However, if the enumeration itself has a more restricted access level, the specified enumeration access level takes precedence.

By default, all enumerations are types and their fields are constants. Therefore the `Shared`, `Static`, and `ReadOnly` keywords cannot be used when declaring an enumeration or its members.

Assigning Multiple Values

Enumerations typically represent mutually exclusive values. By including the `FlagsAttribute` attribute in the `Enum` declaration, you can instead assign multiple values to an instance of the enumeration. The `FlagsAttribute` attribute specifies that the enumeration be treated as a bit field, that is, a set of flags. These are called *bitwise* enumerations.

When you declare an enumeration by using the `FlagsAttribute` attribute, we recommend that you use powers of 2, that is, 1, 2, 4, 8, 16, and so on, for the values. We also recommend that "None" be the name of a member whose value is 0. For additional guidelines, see [FlagsAttribute](#) and [Enum](#).

Example

The following example shows how to use the `Enum` statement. Note that the member is referred to as `EggSizeEnum.Medium`, and not as `Medium`.

```

Public Class Egg
    Enum EggSizeEnum
        Jumbo
        ExtraLarge
        Large
        Medium
        Small
    End Enum

    Public Sub Poach()
        Dim size As EggSizeEnum

        size = EggSizeEnum.Medium
        ' Continue processing...
    End Sub
End Class

```

Example

The method in the following example is outside the `Egg` class. Therefore, `EggSizeEnum` is fully qualified as `Egg.EggSizeEnum`.

```

Public Sub Scramble(ByVal size As Egg.EggSizeEnum)
    ' Process for the three largest sizes.
    ' Throw an exception for any other size.
    Select Case size
        Case Egg.EggSizeEnum.Jumbo
            ' Process.
        Case Egg.EggSizeEnum.ExtraLarge
            ' Process.
        Case Egg.EggSizeEnum.Large
            ' Process.
        Case Else
            Throw New ApplicationException("size is invalid: " & size.ToString)
    End Select
End Sub

```

Example

The following example uses the `Enum` statement to define a related set of named constant values. In this case, the values are colors you might choose to design data entry forms for a database.

```

Public Enum InterfaceColors
    MistyRose = &HE1E4FF&
    SlateGray = &H908070&
    DodgerBlue = &HFF901E&
    DeepSkyBlue = &HFFBF00&
    SpringGreen = &H7FFF00&
    ForestGreen = &H228B22&
    Goldenrod = &H20A5DA&
    Firebrick = &H2222B2&
End Enum

```

Example

The following example shows values that include both positive and negative numbers.

```
Enum SecurityLevel
    IllegalEntry = -1
    MinimumSecurity = 0
    MaximumSecurity = 1
End Enum
```

Example

In the following example, an `As` clause is used to specify the `datatype` of an enumeration.

```
Public Enum MyEnum As Byte
    Zero
    One
    Two
End Enum
```

Example

The following example shows how to use a bitwise enumeration. Multiple values can be assigned to an instance of a bitwise enumeration. The `Enum` declaration includes the [FlagsAttribute](#) attribute, which indicates that the enumeration can be treated as a set of flags.

```
' Apply the Flags attribute, which allows an instance
' of the enumeration to have multiple values.
<Flags()> Public Enum FilePermissions As Integer
    None = 0
    Create = 1
    Read = 2
    Update = 4
    Delete = 8
End Enum

Public Sub ShowBitwiseEnum()

    ' Declare the non-exclusive enumeration object and
    ' set it to multiple values.
    Dim perm As FilePermissions
    perm = FilePermissions.Read Or FilePermissions.Update

    ' Show the values in the enumeration object.
    Console.WriteLine(perm.ToString)
    ' Output: Read, Update

    ' Show the total integer value of all values
    ' in the enumeration object.
    Console.WriteLine(CInt(perm))
    ' Output: 6

    ' Show whether the enumeration object contains
    ' the specified flag.
    Console.WriteLine(perm.HasFlag(FilePermissions.Update))
    ' Output: True
End Sub
```

Example

The following example iterates through an enumeration. It uses the [GetNames](#) method to retrieve an array of member names from the enumeration, and [GetValues](#) to retrieve an array of member values.

```
Enum EggSizeEnum
    Jumbo
    ExtraLarge
    Large
    Medium
    Small
End Enum

Public Sub Iterate()
    Dim names = [Enum].GetNames(GetType(EggSizeEnum))
    For Each name In names
        Console.WriteLine(name & " ")
    Next
    Console.WriteLine()
    ' Output: Jumbo ExtraLarge Large Medium Small

    Dim values = [Enum].GetValues(GetType(EggSizeEnum))
    For Each value In values
        Console.WriteLine(value & " ")
    Next
    Console.WriteLine()
    ' Output: 0 1 2 3 4
End Sub
```

See also

- [Enum](#)
- [AscW](#)
- [Const Statement](#)
- [Dim Statement](#)
- [Implicit and Explicit Conversions](#)
- [Type Conversion Functions](#)
- [Constants and Enumerations](#)

Erase Statement (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Used to release array variables and deallocate the memory used for their elements.

Syntax

```
Erase arraylist
```

Parts

arraylist

Required. List of array variables to be erased. Multiple variables are separated by commas.

Remarks

The `Erase` statement can appear only at procedure level. This means you can release arrays inside a procedure but not at class or module level.

The `Erase` statement is equivalent to assigning `Nothing` to each array variable.

Example

The following example uses the `Erase` statement to clear two arrays and free their memory (1000 and 100 storage elements, respectively). The `ReDim` statement then assigns a new array instance to the three-dimensional array.

```
Dim threeDimArray(9, 9, 9), twoDimArray(9, 9) As Integer
Erase threeDimArray, twoDimArray
ReDim threeDimArray(4, 4, 9)
```

See also

- [Nothing](#)
- [ReDim Statement](#)

Error Statement

10/18/2019 • 2 minutes to read • [Edit Online](#)

Simulates the occurrence of an error.

Syntax

```
Error errornumber
```

Parts

`errornumber`

Required. Can be any valid error number.

Remarks

The `Error` statement is supported for backward compatibility. In new code, especially when creating objects, use the `Err` object's `Raise` method to generate run-time errors.

If `errornumber` is defined, the `Error` statement calls the error handler after the properties of the `Err` object are assigned the following default values:

PROPERTY	VALUE
<code>Number</code>	Value specified as argument to <code>Error</code> statement. Can be any valid error number.
<code>Source</code>	Name of the current Visual Basic project.
<code>Description</code>	String expression corresponding to the return value of the <code>Error</code> function for the specified <code>Number</code> , if this string exists. If the string does not exist, <code>Description</code> contains a zero-length string ("").
<code>HelpFile</code>	The fully qualified drive, path, and file name of the appropriate Visual Basic Help file.
<code>HelpContext</code>	The appropriate Visual Basic Help file context ID for the error corresponding to the <code>Number</code> property.
<code>LastDLLError</code>	Zero.

If no error handler exists, or if none is enabled, an error message is created and displayed from the `Err` object properties.

NOTE

Some Visual Basic host applications cannot create objects. See your host application's documentation to determine whether it can create classes and objects.

Example

This example uses the `Error` statement to generate error number 11.

```
On Error Resume Next    ' Defer error handling.  
Error 11    ' Simulate the "Division by zero" error.
```

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- [Clear](#)
- [Err](#)
- [Raise](#)
- [On Error Statement](#)
- [Resume Statement](#)
- [Error Messages](#)

Event Statement

10/18/2019 • 6 minutes to read • [Edit Online](#)

Declares a user-defined event.

Syntax

```
[ <attrlist> ] [ accessmodifier ] _
[ Shared ] [ Shadows ] Event eventname[(parameterlist)] _
[ Implements implementslist ]
' -or-
[ <attrlist> ] [ accessmodifier ] _
[ Shared ] [ Shadows ] Event eventname As delegatename _
[ Implements implementslist ]
' -or-
[ <attrlist> ] [ accessmodifier ] _
[ Shared ] [ Shadows ] Custom Event eventname As delegatename _
[ Implements implementslist ]
    [ <attrlist> ] AddHandler(ByVal value As delegatename)
        [ statements ]
    End AddHandler
    [ <attrlist> ] RemoveHandler(ByVal value As delegatename)
        [ statements ]
    End RemoveHandler
    [ <attrlist> ] RaiseEvent(delegatesignature)
        [ statements ]
    End RaiseEvent
End Event
```

Parts

PART	DESCRIPTION
<code><attrlist></code>	Optional. List of attributes that apply to this event. Multiple attributes are separated by commas. You must enclose the Attribute List in angle brackets ("<" and ">").
<code>accessmodifier</code>	Optional. Specifies what code can access the event. Can be one of the following: <ul style="list-style-type: none">- Public—any code that can access the element that declares it can access it.- Protected—only code within its class or a derived class can access it.- Friend—only code in the same assembly can access it.- Private—only code in the element that declares it can access it.- Protected Friend—only code in the event's class, a derived class, or the same assembly can access it.- Private Protected—only code in the event's class or a derived class in the same assembly can access it.
<code>Shared</code>	Optional. Specifies that this event is not associated with a specific instance of a class or structure.

PART	DESCRIPTION
Shadows	<p>Optional. Indicates that this event redeclares and hides an identically named programming element, or set of overloaded elements, in a base class. You can shadow any kind of declared element with any other kind.</p> <p>A shadowed element is unavailable from within the derived class that shadows it, except from where the shadowing element is inaccessible. For example, if a <code>Private</code> element shadows a base-class element, code that does not have permission to access the <code>Private</code> element accesses the base-class element instead.</p>
eventname	Required. Name of the event; follows standard variable naming conventions.
parameterlist	Optional. List of local variables that represent the parameters of this event. You must enclose the Parameter List in parentheses.
Implements	Optional. Indicates that this event implements an event of an interface.
implementslist	<p>Required if <code>Implements</code> is supplied. List of <code>Sub</code> procedures being implemented. Multiple procedures are separated by commas:</p> <p><i>implementedprocedure</i> [, <i>implementedprocedure</i> ...]</p> <p>Each <code>implementedprocedure</code> has the following syntax and parts:</p> <p style="text-align: center;"><code>interface</code> . <code>definedname</code></p> <ul style="list-style-type: none"> - <code>interface</code> - Required. Name of an interface that this procedure's containing class or structure is implementing. - <code>Definedname</code> - Required. Name by which the procedure is defined in <code>interface</code>. This does not have to be the same as <code>name</code>, the name that this procedure is using to implement the defined procedure.
Custom	Required. Events declared as <code>Custom</code> must define custom <code>AddHandler</code> , <code>RemoveHandler</code> , and <code>RaiseEvent</code> accessors.
delegatename	Optional. The name of a delegate that specifies the event-handler signature.
AddHandler	Required. Declares an <code>AddHandler</code> accessor, which specifies the statements to execute when an event handler is added, either explicitly by using the <code>AddHandler</code> statement or implicitly by using the <code>Handles</code> clause.
End AddHandler	Required. Terminates the <code>AddHandler</code> block.
value	Required. Parameter name.

PART	DESCRIPTION
<code>RemoveHandler</code>	Required. Declares a <code>RemoveHandler</code> accessor, which specifies the statements to execute when an event handler is removed using the <code>RemoveHandler</code> statement.
<code>End RemoveHandler</code>	Required. Terminates the <code>RemoveHandler</code> block.
<code>RaiseEvent</code>	Required. Declares a <code>RaiseEvent</code> accessor, which specifies the statements to execute when the event is raised using the <code>RaiseEvent</code> statement. Typically, this invokes a list of delegates maintained by the <code>AddHandler</code> and <code>RemoveHandler</code> accessors.
<code>End RaiseEvent</code>	Required. Terminates the <code>RaiseEvent</code> block.
<code>delegatesignature</code>	Required. List of parameters that matches the parameters required by the <code>delegatename</code> delegate. You must enclose the Parameter List in parentheses.
<code>statements</code>	Optional. Statements that contain the bodies of the <code>AddHandler</code> , <code>RemoveHandler</code> , and <code>RaiseEvent</code> methods.
<code>End Event</code>	Required. Terminates the <code>Event</code> block.

Remarks

Once the event has been declared, use the `RaiseEvent` statement to raise the event. A typical event might be declared and raised as shown in the following fragments:

```
Public Class EventSource
    ' Declare an event.
    Public Event LogonCompleted(ByVal UserName As String)
    Sub CauseEvent()
        ' Raise an event on successful logon.
        RaiseEvent LogonCompleted("AustinSteele")
    End Sub
End Class
```

NOTE

You can declare event arguments just as you do arguments of procedures, with the following exceptions: events cannot have named arguments, `ParamArray` arguments, or `Optional` arguments. Events do not have return values.

To handle an event, you must associate it with an event handler subroutine using either the `Handles` or `AddHandler` statement. The signatures of the subroutine and the event must match. To handle a shared event, you must use the `AddHandler` statement.

You can use `Event` only at module level. This means the *declaration context* for an event must be a class, structure, module, or interface, and cannot be a source file, namespace, procedure, or block. For more information, see [Declaration Contexts and Default Access Levels](#).

In most circumstances, you can use the first syntax in the Syntax section of this topic for declaring events. However, some scenarios require that you have more control over the detailed behavior of the event. The last syntax in the Syntax section of this topic, which uses the `custom` keyword, provides that control by enabling you to define custom events. In a custom event, you specify exactly what occurs when code adds or removes an event handler to or from the event, or when code raises the event. For examples, see [How to: Declare Custom Events To Conserve Memory](#) and [How to: Declare Custom Events To Avoid Blocking](#).

Example

The following example uses events to count down seconds from 10 to 0. The code illustrates several of the event-related methods, properties, and statements. This includes the `RaiseEvent` statement.

The class that raises an event is the event source, and the methods that process the event are the event handlers. An event source can have multiple handlers for the events it generates. When the class raises the event, that event is raised on every class that has elected to handle events for that instance of the object.

The example also uses a form (`Form1`) with a button (`Button1`) and a text box (`TextBox1`). When you click the button, the first text box displays a countdown from 10 to 0 seconds. When the full time (10 seconds) has elapsed, the first text box displays "Done".

The code for `Form1` specifies the initial and terminal states of the form. It also contains the code executed when events are raised.

To use this example, open a new Windows Forms project. Then add a button named `Button1` and a text box named `TextBox1` to the main form, named `Form1`. Then right-click the form and click **View Code** to open the code editor.

Add a `WithEvents` variable to the declarations section of the `Form1` class:

```
Private WithEvents mText As TimerState
```

Add the following code to the code for `Form1`. Replace any duplicate procedures that may exist, such as `Form_Load` or `Button_Click`.

```

Private Sub Form1_Load() Handles MyBase.Load
    Button1.Text = "Start"
    mText = New TimerState
End Sub
Private Sub Button1_Click() Handles Button1.Click
    mText.StartCountdown(10.0, 0.1)
End Sub

Private Sub mText_ChangeText() Handles mText.Finished
    TextBox1.Text = "Done"
End Sub

Private Sub mTextUpdateTime(ByVal Countdown As Double
) Handles mText.UpdateTime

    TextBox1.Text = Format(Countdown, "##0.0")
    ' Use DoEvents to allow the display to refresh.
    My.Application.DoEvents()
End Sub

Class TimerState
    Public Event UpdateTime(ByVal Countdown As Double)
    Public Event Finished()
    Public Sub StartCountdown(ByVal Duration As Double,
        ByVal Increment As Double)
        Dim Start As Double = DateAndTime.Timer
        Dim ElapsedTime As Double = 0

        Dim SoFar As Double = 0
        Do While ElapsedTime < Duration
            If ElapsedTime > SoFar + Increment Then
                SoFar += Increment
                RaiseEvent UpdateTime(Duration - SoFar)
            End If
            ElapsedTime = DateAndTime.Timer - Start
        Loop
        RaiseEvent Finished()
    End Sub
End Class

```

Press F5 to run the previous example, and click the button labeled **Start**. The first text box starts to count down the seconds. When the full time (10 seconds) has elapsed, the first text box displays "Done".

NOTE

The `My.Application.DoEvents` method does not process events in the same way the form does. To enable the form to handle the events directly, you can use multithreading. For more information, see [Managed Threading](#).

See also

- [RaiseEvent Statement](#)
- [Implements Statement](#)
- [Events](#)
- [AddHandler Statement](#)
- [RemoveHandler Statement](#)
- [Handles](#)
- [Delegate Statement](#)
- [How to: Declare Custom Events To Conserve Memory](#)
- [How to: Declare Custom Events To Avoid Blocking](#)

- Shared
- Shadows

Exit Statement (Visual Basic)

10/4/2019 • 3 minutes to read • [Edit Online](#)

Exits a procedure or block and transfers control immediately to the statement following the procedure call or the block definition.

Syntax

```
Exit { Do | For | Function | Property | Select | Sub | Try | While }
```

Statements

Exit Do

Immediately exits the `Do` loop in which it appears. Execution continues with the statement following the `Loop` statement. `Exit Do` can be used only inside a `Do` loop. When used within nested `Do` loops, `Exit Do` exits the innermost loop and transfers control to the next higher level of nesting.

Exit For

Immediately exits the `For` loop in which it appears. Execution continues with the statement following the `Next` statement. `Exit For` can be used only inside a `For ... Next` or `For Each ... Next` loop. When used within nested `For` loops, `Exit For` exits the innermost loop and transfers control to the next higher level of nesting.

Exit Function

Immediately exits the `Function` procedure in which it appears. Execution continues with the statement following the statement that called the `Function` procedure. `Exit Function` can be used only inside a `Function` procedure.

To specify a return value, you can assign the value to the function name on a line before the `Exit Function` statement. To assign the return value and exit the function in one statement, you can instead use the [Return Statement](#).

Exit Property

Immediately exits the `Property` procedure in which it appears. Execution continues with the statement that called the `Property` procedure, that is, with the statement requesting or setting the property's value.

`Exit Property` can be used only inside a property's `Get` or `Set` procedure.

To specify a return value in a `Get` procedure, you can assign the value to the function name on a line before the `Exit Property` statement. To assign the return value and exit the `Get` procedure in one statement, you can instead use the `Return` statement.

In a `Set` procedure, the `Exit Property` statement is equivalent to the `Return` statement.

Exit Select

Immediately exits the `Select Case` block in which it appears. Execution continues with the statement following the `End Select` statement. `Exit Select` can be used only inside a `Select Case` statement.

Exit Sub

Immediately exits the `Sub` procedure in which it appears. Execution continues with the statement following the statement that called the `Sub` procedure. `Exit Sub` can be used only inside a `Sub` procedure.

In a `Sub` procedure, the `Exit Sub` statement is equivalent to the `Return` statement.

`Exit Try`

Immediately exits the `Try` or `Catch` block in which it appears. Execution continues with the `Finally` block if there is one, or with the statement following the `End Try` statement otherwise. `Exit Try` can be used only inside a `Try` or `Catch` block, and not inside a `Finally` block.

`Exit While`

Immediately exits the `While` loop in which it appears. Execution continues with the statement following the `End While` statement. `Exit While` can be used only inside a `While` loop. When used within nested `While` loops, `Exit While` transfers control to the loop that is one nested level above the loop where `Exit While` occurs.

Remarks

Do not confuse `Exit` statements with `End` statements. `Exit` does not define the end of a statement.

Example

In the following example, the loop condition stops the loop when the `index` variable is greater than 100. The `If` statement in the loop, however, causes the `Exit Do` statement to stop the loop when the `index` variable is greater than 10.

```
Dim index As Integer = 0
Do While index <= 100
    If index > 10 Then
        Exit Do
    End If

    Debug.WriteLine(index.ToString & " ")
    index += 1
Loop

Debug.WriteLine("")
' Output: 0 1 2 3 4 5 6 7 8 9 10
```

Example

The following example assigns the return value to the function name `myFunction`, and then uses `Exit Function` to return from the function:

```
Function MyFunction(ByVal j As Integer) As Double
    MyFunction = 3.87 * j
    Exit Function
End Function
```

Example

The following example uses the [Return Statement](#) to assign the return value and exit the function:

```
Function MyFunction(ByVal j As Integer) As Double
    Return 3.87 * j
End Function
```

See also

- [Continue Statement](#)
- [Do...Loop Statement](#)
- [End Statement](#)
- [For Each...Next Statement](#)
- [For...Next Statement](#)
- [Function Statement](#)
- [Return Statement](#)
- [Stop Statement](#)
- [Sub Statement](#)
- [Try...Catch...Finally Statement](#)

F-P Statements

4/2/2019 • 2 minutes to read • [Edit Online](#)

The following table contains a listing of Visual Basic language statements.

For Each...Next	For...Next	Function	Get
GoTo	If...Then...Else	Implements	Imports (.NET Namespace and Type)
Imports (XML Namespace)	Inherits	Interface	Mid
Module	Namespace	On Error	Operator
Option <keyword>	Option Compare	Option Explicit	Option Infer
Option Strict	Property		

See also

- [A-E Statements](#)
- [Q-Z Statements](#)
- [Visual Basic Language Reference](#)

For Each...Next Statement (Visual Basic)

9/27/2019 • 13 minutes to read • [Edit Online](#)

Repeats a group of statements for each element in a collection.

Syntax

```
For Each element [ As datatype ] In group
    [ statements ]
    [ Continue For ]
    [ statements ]
    [ Exit For ]
    [ statements ]
Next [ element ]
```

Parts

TERM	DEFINITION
<code>element</code>	Required in the <code>For Each</code> statement. Optional in the <code>Next</code> statement. Variable. Used to iterate through the elements of the collection.
<code>datatype</code>	Optional if <code>Option Infer</code> is on (the default) or <code>element</code> is already declared; required if <code>Option Infer</code> is off and <code>element</code> isn't already declared. The data type of <code>element</code> .
<code>group</code>	Required. A variable with a type that's a collection type or Object. Refers to the collection over which the <code>statements</code> are to be repeated.
<code>statements</code>	Optional. One or more statements between <code>For Each</code> and <code>Next</code> that run on each item in <code>group</code> .
<code>Continue For</code>	Optional. Transfers control to the start of the <code>For Each</code> loop.
<code>Exit For</code>	Optional. Transfers control out of the <code>For Each</code> loop.
<code>Next</code>	Required. Terminates the definition of the <code>For Each</code> loop.

Simple Example

Use a `For Each ... Next` loop when you want to repeat a set of statements for each element of a collection or array.

TIP

A [For...Next Statement](#) works well when you can associate each iteration of a loop with a control variable and determine that variable's initial and final values. However, when you are dealing with a collection, the concept of initial and final values isn't meaningful, and you don't necessarily know how many elements the collection has. In this kind of case, a `For Each ... Next` loop is often a better choice.

In the following example, the `For Each ... Next` statement iterates through all the elements of a List collection.

```
' Create a list of strings by using a
' collection initializer.
Dim lst As New List(Of String) _
    From {"abc", "def", "ghi"}

' Iterate through the list.
For Each item As String In lst
    Debug.WriteLine(item & " ")
Next
Debug.WriteLine("")
'Output: abc def ghi
```

For more examples, see [Collections and Arrays](#).

Nested Loops

You can nest `For Each` loops by putting one loop within another.

The following example demonstrates nested `For Each ... Next` structures.

```
' Create lists of numbers and letters
' by using array initializers.
Dim numbers() As Integer = {1, 4, 7}
Dim letters() As String = {"a", "b", "c"}

' Iterate through the list by using nested loops.
For Each number As Integer In numbers
    For Each letter As String In letters
        Debug.WriteLine(number.ToString & letter & " ")
    Next
Next
Debug.WriteLine("")
'Output: 1a 1b 1c 4a 4b 4c 7a 7b 7c
```

When you nest loops, each loop must have a unique `element` variable.

You can also nest different kinds of control structures within each other. For more information, see [Nested Control Structures](#).

Exit For and Continue For

The [Exit For](#) statement causes execution to exit the `For ... Next` loop and transfers control to the statement that follows the `Next` statement.

The `Continue For` statement transfers control immediately to the next iteration of the loop. For more information, see [Continue Statement](#).

The following example shows how to use the `Continue For` and `Exit For` statements.

```

Dim numberSeq() As Integer =
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

For Each number As Integer In numberSeq
    ' If number is between 5 and 7, continue
    ' with the next iteration.
    If number >= 5 And number <= 8 Then
        Continue For
    End If

    ' Display the number.
    Debug.Write(number.ToString & " ")

    ' If number is 10, exit the loop.
    If number = 10 Then
        Exit For
    End If
Next
Debug.WriteLine("")
' Output: 1 2 3 4 9 10

```

You can put any number of `Exit For` statements in a `For Each` loop. When used within nested `For Each` loops, `Exit For` causes execution to exit the innermost loop and transfers control to the next higher level of nesting.

`Exit For` is often used after an evaluation of some condition, for example, in an `If ... Then ... Else` structure. You might want to use `Exit For` for the following conditions:

- Continuing to iterate is unnecessary or impossible. This might be caused by an erroneous value or a termination request.
- An exception is caught in a `Try ... Catch ... Finally`. You might use `Exit For` at the end of the `Finally` block.
- There an endless loop, which is a loop that could run a large or even infinite number of times. If you detect such a condition, you can use `Exit For` to escape the loop. For more information, see [Do...Loop Statement](#).

Iterators

You use an *iterator* to perform a custom iteration over a collection. An iterator can be a function or a `Get` accessor. It uses a `Yield` statement to return each element of the collection one at a time.

You call an iterator by using a `For Each...Next` statement. Each iteration of the `For Each` loop calls the iterator. When a `Yield` statement is reached in the iterator, the expression in the `Yield` statement is returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator is called.

The following example uses an iterator function. The iterator function has a `Yield` statement that's inside a `For...Next` loop. In the `ListEvenNumbers` method, each iteration of the `For Each` statement body creates a call to the iterator function, which proceeds to the next `Yield` statement.

```

Public Sub ListEvenNumbers()
    For Each number As Integer In EvenSequence(5, 18)
        Debug.WriteLine(number & " ")
    Next
    Debug.WriteLine("")
    ' Output: 6 8 10 12 14 16 18
End Sub

Private Iterator Function EvenSequence(
    ByVal firstNumber As Integer, ByVal lastNumber As Integer) _
    As System.Collections.Generic.IEnumerable(Of Integer)

    ' Yield even numbers in the range.
    For number = firstNumber To lastNumber
        If number Mod 2 = 0 Then
            Yield number
        End If
    Next
End Function

```

For more information, see [Iterators, Yield Statement, and Iterator](#).

Technical Implementation

When a `For Each ... Next` statement runs, Visual Basic evaluates the collection only one time, before the loop starts. If your statement block changes `element` or `group`, these changes don't affect the iteration of the loop.

When all the elements in the collection have been successively assigned to `element`, the `For Each` loop stops and control passes to the statement following the `Next` statement.

If [Option Infer](#) is on (its default setting), the Visual Basic compiler can infer the data type of `element`. If it is off and `element` hasn't been declared outside the loop, you must declare it in the `For Each` statement. To declare the data type of `element` explicitly, use an `As` clause. Unless the data type of `element` is defined outside the `For Each ... Next` construct, its scope is the body of the loop. Note that you cannot declare `element` both outside and inside the loop.

You can optionally specify `element` in the `Next` statement. This improves the readability of your program, especially if you have nested `For Each` loops. You must specify the same variable as the one that appears in the corresponding `For Each` statement.

You might want to avoid changing the value of `element` inside a loop. Doing this can make it more difficult to read and debug your code. Changing the value of `group` doesn't affect the collection or its elements, which were determined when the loop was first entered.

When you're nesting loops, if a `Next` statement of an outer nesting level is encountered before the `Next` of an inner level, the compiler signals an error. However, the compiler can detect this overlapping error only if you specify `element` in every `Next` statement.

If your code depends on traversing a collection in a particular order, a `For Each ... Next` loop isn't the best choice, unless you know the characteristics of the enumerator object the collection exposes. The order of traversal isn't determined by Visual Basic, but by the [MoveNext](#) method of the enumerator object. Therefore, you might not be able to predict which element of the collection is the first to be returned in `element`, or which is the next to be returned after a given element. You might achieve more reliable results using a different loop structure, such as `For ... Next` or `Do ... Loop`.

The runtime must be able to convert the elements in `group` to `element`. The `[Option Strict]` statement controls whether both widening and narrowing conversions are allowed (`Option Strict` is off, its default

value), or whether only widening conversions are allowed (`Option Strict` is on). For more information, see [Narrowing conversions](#).

The data type of `group` must be a reference type that refers to a collection or an array that's enumerable. Most commonly this means that `group` refers to an object that implements the `IEnumerable` interface of the `System.Collections` namespace or the `IEnumerable<T>` interface of the `System.Collections.Generic` namespace. `System.Collections.IEnumerable` defines the `GetEnumerator` method, which returns an enumerator object for the collection. The enumerator object implements the `System.Collections.IEnumerator` interface of the `System.Collections` namespace and exposes the `Current` property and the `Reset` and `MoveNext` methods. Visual Basic uses these to traverse the collection.

Narrowing Conversions

When `Option Strict` is set to `On`, narrowing conversions ordinarily cause compiler errors. In a `For Each` statement, however, conversions from the elements in `group` to `element` are evaluated and performed at run time, and compiler errors caused by narrowing conversions are suppressed.

In the following example, the assignment of `m` as the initial value for `n` doesn't compile when `Option Strict` is on because the conversion of a `Long` to an `Integer` is a narrowing conversion. In the `For Each` statement, however, no compiler error is reported, even though the assignment to `number` requires the same conversion from `Long` to `Integer`. In the `For Each` statement that contains a large number, a run-time error occurs when `ToInteger` is applied to the large number.

```
Option Strict On

Module Module1
    Sub Main()
        ' The assignment of m to n causes a compiler error when
        ' Option Strict is on.
        Dim m As Long = 987
        'Dim n As Integer = m

        ' The For Each loop requires the same conversion but
        ' causes no errors, even when Option Strict is on.
        For Each number As Integer In New Long() {45, 3, 987}
            Console.Write(number & " ")
        Next
        Console.WriteLine()
        ' Output: 45 3 987

        ' Here a run-time error is raised because 9876543210
        ' is too large for type Integer.
        'For Each number As Integer In New Long() {45, 3, 9876543210}
        '    Console.Write(number & " ")
        'Next

        Console.ReadKey()
    End Sub
End Module
```

IEnumerator Calls

When execution of a `For Each ... Next` loop starts, Visual Basic verifies that `group` refers to a valid collection object. If not, it throws an exception. Otherwise, it calls the `MoveNext` method and the `Current` property of the enumerator object to return the first element. If `MoveNext` indicates that there is no next element, that is, if the collection is empty, the `For Each` loop stops and control passes to the statement following the `Next` statement. Otherwise, Visual Basic sets `element` to the first element and runs the statement block.

Each time Visual Basic encounters the `Next` statement, it returns to the `For Each` statement. Again it calls `MoveNext` and `Current` to return the next element, and again it either runs the block or stops the loop.

depending on the result. This process continues until `MoveNext` indicates that there is no next element or an `Exit For` statement is encountered.

Modifying the Collection. The enumerator object returned by `GetEnumerator` normally doesn't let you change the collection by adding, deleting, replacing, or reordering any elements. If you change the collection after you have initiated a `For Each ... Next` loop, the enumerator object becomes invalid, and the next attempt to access an element causes an `InvalidOperationException` exception.

However, this blocking of modification isn't determined by Visual Basic, but rather by the implementation of the `IEnumerable` interface. It is possible to implement `IEnumerable` in a way that allows for modification during iteration. If you are considering doing such dynamic modification, make sure that you understand the characteristics of the `IEnumerable` implementation on the collection you are using.

Modifying Collection Elements. The `Current` property of the enumerator object is `ReadOnly`, and it returns a local copy of each collection element. This means that you cannot modify the elements themselves in a `For Each ... Next` loop. Any modification you make affects only the local copy from `Current` and isn't reflected back into the underlying collection. However, if an element is a reference type, you can modify the members of the instance to which it points. The following example modifies the `BackColor` member of each `thisControl` element. You cannot, however, modify `thisControl` itself.

```
Sub LightBlueBackground(thisForm As System.Windows.Forms.Form)
    For Each thisControl In thisForm.Controls
        thisControl.BackColor = System.Drawing.Color.LightBlue
    Next thisControl
End Sub
```

The previous example can modify the `BackColor` member of each `thisControl` element, although it cannot modify `thisControl` itself.

Traversing Arrays. Because the `Array` class implements the `IEnumerable` interface, all arrays expose the `GetEnumerator` method. This means that you can iterate through an array with a `For Each ... Next` loop. However, you can only read the array elements. You cannot change them.

Example

The following example lists all the folders in the C:\ directory by using the `DirectoryInfo` class.

```
Dim dInfo As New System.IO.DirectoryInfo("c:\")
For Each dir As System.IO.DirectoryInfo In dInfo.GetDirectories()
    Debug.WriteLine(dir.Name)
Next
```

Example

The following example illustrates a procedure for sorting a collection. The example sorts instances of a `Car` class that are stored in a `List<T>`. The `Car` class implements the `IComparable<T>` interface, which requires that the `CompareTo` method be implemented.

Each call to the `CompareTo` method makes a single comparison that's used for sorting. User-written code in the `CompareTo` method returns a value for each comparison of the current object with another object. The value returned is less than zero if the current object is less than the other object, greater than zero if the current object is greater than the other object, and zero if they are equal. This enables you to define in code the criteria for greater than, less than, and equal.

In the `ListCars` method, the `cars.Sort()` statement sorts the list. This call to the `Sort` method of the

`List<T>` causes the `CompareTo` method to be called automatically for the `Car` objects in the `List`.

```
Public Sub ListCars()

    ' Create some new cars.
    Dim cars As New List(Of Car) From
    {
        New Car With {.Name = "car1", .Color = "blue", .Speed = 20},
        New Car With {.Name = "car2", .Color = "red", .Speed = 50},
        New Car With {.Name = "car3", .Color = "green", .Speed = 10},
        New Car With {.Name = "car4", .Color = "blue", .Speed = 50},
        New Car With {.Name = "car5", .Color = "blue", .Speed = 30},
        New Car With {.Name = "car6", .Color = "red", .Speed = 60},
        New Car With {.Name = "car7", .Color = "green", .Speed = 50}
    }

    ' Sort the cars by color alphabetically, and then by speed
    ' in descending order.
    cars.Sort()

    ' View all of the cars.
    For Each thisCar As Car In cars
        Debug.Write(thisCar.Color.PadRight(5) & " ")
        Debug.Write(thisCar.Speed.ToString & " ")
        Debug.Write(thisCar.Name)
        Debug.WriteLine("")
    Next

    ' Output:
    ' blue 50 car4
    ' blue 30 car5
    ' blue 20 car1
    ' green 50 car7
    ' green 10 car3
    ' red   60 car6
    ' red   50 car2
End Sub

Public Class Car
    Implements IComparable(Of Car)

        Public Property Name As String
        Public Property Speed As Integer
        Public Property Color As String

        Public Function CompareTo(ByVal other As Car) As Integer _
            Implements System.IComparable(Of Car).CompareTo
            ' A call to this method makes a single comparison that is
            ' used for sorting.

            ' Determine the relative order of the objects being compared.
            ' Sort by color alphabetically, and then by speed in
            ' descending order.

            ' Compare the colors.
            Dim compare As Integer
            compare = String.Compare(Me.Color, other.Color, True)

            ' If the colors are the same, compare the speeds.
            If compare = 0 Then
                compare = Me.Speed.CompareTo(other.Speed)

                ' Use descending order for speed.
                compare = -compare
            End If

            Return compare
        End Function
    End Class
End Module
```

End Class

See also

- [Collections](#)
- [For...Next Statement](#)
- [Loop Structures](#)
- [While...End While Statement](#)
- [Do...Loop Statement](#)
- [Widening and Narrowing Conversions](#)
- [Object Initializers: Named and Anonymous Types](#)
- [Collection Initializers](#)
- [Arrays](#)

For...Next Statement (Visual Basic)

10/18/2019 • 9 minutes to read • [Edit Online](#)

Repeats a group of statements a specified number of times.

Syntax

```
For counter [ As datatype ] = start To end [ Step step ]
    [ statements ]
    [ Continue For ]
    [ statements ]
    [ Exit For ]
    [ statements ]
Next [ counter ]
```

Parts

PART	DESCRIPTION
counter	Required in the <code>For</code> statement. Numeric variable. The control variable for the loop. For more information, see Counter Argument later in this topic.
datatype	Optional. Data type of <code>counter</code> . For more information, see Counter Argument later in this topic.
start	Required. Numeric expression. The initial value of <code>counter</code> .
end	Required. Numeric expression. The final value of <code>counter</code> .
step	Optional. Numeric expression. The amount by which <code>counter</code> is incremented each time through the loop.
statements	Optional. One or more statements between <code>For</code> and <code>Next</code> that run the specified number of times.
Continue For	Optional. Transfers control to the next loop iteration.
Exit For	Optional. Transfers control out of the <code>For</code> loop.
Next	Required. Terminates the definition of the <code>For</code> loop.

NOTE

The `To` keyword is used in this statement to specify the range for the counter. You can also use this keyword in the [Select...Case Statement](#) and in array declarations. For more information about array declarations, see [Dim Statement](#).

Simple Examples

You use a `For ... Next` structure when you want to repeat a set of statements a set number of times.

In the following example, the `index` variable starts with a value of 1 and is incremented with each iteration of the loop, ending after the value of `index` reaches 5.

```
For index As Integer = 1 To 5
    Debug.WriteLine(index.ToString & " ")
Next
Debug.WriteLine("")
' Output: 1 2 3 4 5
```

In the following example, the `number` variable starts at 2 and is reduced by 0.25 on each iteration of the loop, ending after the value of `number` reaches 0. The `Step` argument of `-0.25` reduces the value by 0.25 on each iteration of the loop.

```
For number As Double = 2 To 0 Step -0.25
    Debug.WriteLine(number.ToString & " ")
Next
Debug.WriteLine("")
' Output: 2 1.75 1.5 1.25 1 0.75 0.5 0.25 0
```

TIP

A [While...End While Statement](#) or [Do...Loop Statement](#) works well when you don't know in advance how many times to run the statements in the loop. However, when you expect to run the loop a specific number of times, a `For ... Next` loop is a better choice. You determine the number of iterations when you first enter the loop.

Nesting Loops

You can nest `For` loops by putting one loop within another. The following example demonstrates nested `For ... Next` structures that have different step values. The outer loop creates a string for every iteration of the loop. The inner loop decrements a loop counter variable for every iteration of the loop.

```
For indexA = 1 To 3
    ' Create a new StringBuilder, which is used
    ' to efficiently build strings.
    Dim sb As New System.Text.StringBuilder()

    ' Append to the StringBuilder every third number
    ' from 20 to 1 descending.
    For indexB = 20 To 1 Step -3
        sb.Append(indexB.ToString)
        sb.Append(" ")
    Next indexB

    ' Display the line.
    Debug.WriteLine(sb.ToString)
Next indexA
' Output:
' 20 17 14 11 8 5 2
' 20 17 14 11 8 5 2
' 20 17 14 11 8 5 2
```

When nesting loops, each loop must have a unique `counter` variable.

You can also nest different kinds control structures within each other. For more information, see [Nested Control Structures](#).

Exit For and Continue For

The `Exit For` statement immediately exits the `For ... Next` loop and transfers control to the statement that follows the `Next` statement.

The `Continue For` statement transfers control immediately to the next iteration of the loop. For more information, see [Continue Statement](#).

The following example illustrates the use of the `Continue For` and `Exit For` statements.

```
For index As Integer = 1 To 100000
    ' If index is between 5 and 7, continue
    ' with the next iteration.
    If index >= 5 AndAlso index <= 8 Then
        Continue For
    End If

    ' Display the index.
    Debug.WriteLine(index.ToString & " ")

    ' If index is 10, exit the loop.
    If index = 10 Then
        Exit For
    End If
Next
Debug.WriteLine("")
' Output: 1 2 3 4 9 10
```

You can put any number of `Exit For` statements in a `For ... Next` loop. When used within nested `For ... Next` loops, `Exit For` exits the innermost loop and transfers control to the next higher level of nesting.

`Exit For` is often used after you evaluate some condition (for example, in an `If ... Then ... Else` structure). You might want to use `Exit For` for the following conditions:

- Continuing to iterate is unnecessary or impossible. An erroneous value or a termination request might create this condition.
- A `Try ... Catch ... Finally` statement catches an exception. You might use `Exit For` at the end of the `Finally` block.
- You have an endless loop, which is a loop that could run a large or even infinite number of times. If you detect such a condition, you can use `Exit For` to escape the loop. For more information, see [Do...Loop Statement](#).

Technical Implementation

When a `For ... Next` loop starts, Visual Basic evaluates `start`, `end`, and `step`. Visual Basic evaluates these values only at this time and then assigns `start` to `counter`. Before the statement block runs, Visual Basic compares `counter` to `end`. If `counter` is already larger than the `end` value (or smaller if `step` is negative), the `For` loop ends and control passes to the statement that follows the `Next` statement. Otherwise, the statement block runs.

Each time Visual Basic encounters the `Next` statement, it increments `counter` by `step` and returns to the `For` statement. Again it compares `counter` to `end`, and again it either runs the block or exits the loop, depending on the result. This process continues until `counter` passes `end` or an `Exit For` statement is

encountered.

The loop doesn't stop until `counter` has passed `end`. If `counter` is equal to `end`, the loop continues. The comparison that determines whether to run the block is `counter <= end` if `step` is positive and `counter >= end` if `step` is negative.

If you change the value of `counter` while inside a loop, your code might be more difficult to read and debug. Changing the value of `start`, `end`, or `step` doesn't affect the iteration values that were determined when the loop was first entered.

If you nest loops, the compiler signals an error if it encounters the `Next` statement of an outer nesting level before the `Next` statement of an inner level. However, the compiler can detect this overlapping error only if you specify `counter` in every `Next` statement.

Step Argument

The value of `step` can be either positive or negative. This parameter determines loop processing according to the following table:

STEP VALUE	LOOP EXECUTES IF
Positive or zero	<code>counter <= end</code>
Negative	<code>counter >= end</code>

The default value of `step` is 1.

Counter Argument

The following table indicates whether `counter` defines a new local variable that's scoped to the entire `For...Next` loop. This determination depends on whether `datatype` is present and whether `counter` is already defined.

IS DATATYPE PRESENT?	IS COUNTER ALREADY DEFINED?	RESULT (WHETHER COUNTER DEFINES A NEW LOCAL VARIABLE THAT'S SCOPED TO THE ENTIRE FOR...NEXT LOOP)
No	Yes	No, because <code>counter</code> is already defined. If the scope of <code>counter</code> isn't local to the procedure, a compile-time warning occurs.
No	No	Yes. The data type is inferred from the <code>start</code> , <code>end</code> , and <code>step</code> expressions. For information about type inference, see Option Infer Statement and Local Type Inference .
Yes	Yes	Yes, but only if the existing <code>counter</code> variable is defined outside the procedure. That variable remains separate. If the scope of the existing <code>counter</code> variable is local to the procedure, a compile-time error occurs.
Yes	No	Yes.

The data type of `counter` determines the type of the iteration, which must be one of the following types:

- A `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, `Decimal`, `Single`, or `Double`.
- An enumeration that you declare by using an [Enum Statement](#).
- An `Object`.
- A type `T` that has the following operators, where `B` is a type that can be used in a `Boolean` expression.

```
Public Shared Operator >= (op1 As T, op2 As T) As B
```

```
Public Shared Operator <= (op1 As T, op2 As T) As B
```

```
Public Shared Operator - (op1 As T, op2 As T) As T
```

```
Public Shared Operator + (op1 As T, op2 As T) As T
```

You can optionally specify the `counter` variable in the `Next` statement. This syntax improves the readability of your program, especially if you have nested `For` loops. You must specify the variable that appears in the corresponding `For` statement.

The `start`, `end`, and `step` expressions can evaluate to any data type that widens to the type of `counter`. If you use a user-defined type for `counter`, you might have to define the `CType` conversion operator to convert the types of `start`, `end`, or `step` to the type of `counter`.

Example

The following example removes all elements from a generic list. Instead of a [For Each...Next Statement](#), the example shows a `For ... Next` statement that iterates in descending order. The example uses this technique because the `removeAt` method causes elements after the removed element to have a lower index value.

```
Dim lst As New List(Of Integer) From {10, 20, 30, 40}

For index As Integer = lst.Count - 1 To 0 Step -1
    lst.RemoveAt(index)
Next

Debug.WriteLine(lst.Count.ToString)
' Output: 0
```

Example

The following example iterates through an enumeration that's declared by using an [Enum Statement](#).

```

Public Enum Mammals
    Buffalo
    Gazelle
    Mongoose
    Rhinoceros
    Whale
End Enum

Public Sub ListSomeMammals()
    For mammal As Mammals = Mammals.Gazelle To Mammals.Rhinoceros
        Debug.WriteLine(mammal.ToString & " ")
    Next
    Debug.WriteLine("")
    ' Output: Gazelle Mongoose Rhinoceros
End Sub

```

Example

In the following example, the statement parameters use a class that has operator overloads for the `+`, `-`, `>=`, and `<=` operators.

```

Private Class Distance
    Public Property Number() As Double

    Public Sub New(ByVal number As Double)
        Me.Number = number
    End Sub

    ' Define operator overloads to support For...Next statements.
    Public Shared Operator +(ByVal op1 As Distance, ByVal op2 As Distance) As Distance
        Return New Distance(op1.Number + op2.Number)
    End Operator

    Public Shared Operator -(ByVal op1 As Distance, ByVal op2 As Distance) As Distance
        Return New Distance(op1.Number - op2.Number)
    End Operator

    Public Shared Operator >=(ByVal op1 As Distance, ByVal op2 As Distance) As Boolean
        Return (op1.Number >= op2.Number)
    End Operator

    Public Shared Operator <=(ByVal op1 As Distance, ByVal op2 As Distance) As Boolean
        Return (op1.Number <= op2.Number)
    End Operator
End Class

Public Sub ListDistances()
    Dim distFrom As New Distance(10)
    Dim distTo As New Distance(25)
    Dim distStep As New Distance(4)

    For dist As Distance = distFrom To distTo Step distStep
        Debug.WriteLine(dist.Number.ToString & " ")
    Next
    Debug.WriteLine("")
    ' Output: 10 14 18 22
End Sub

```

See also

- [List<T>](#)
- [Loop Structures](#)
- [While...End While Statement](#)
- [Do...Loop Statement](#)
- [Nested Control Structures](#)
- [Exit Statement](#)
- [Collections](#)

Function Statement (Visual Basic)

10/18/2019 • 8 minutes to read • [Edit Online](#)

Declares the name, parameters, and code that define a `Function` procedure.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ proceduremodifiers ] [ Shared ] [ Shadows ] [ Async |  
Iterator ]  
Function name [ (Of typeparamlist) ] [ (parameterlist) ] [ As returntype ] [ Implements  
implementslist | Handles eventlist ]  
    [ statements ]  
    [ Exit Function ]  
    [ statements ]  
End Function
```

Parts

- `attributelist`

Optional. See [Attribute List](#).

- `accessmodifier`

Optional. Can be one of the following:

- [Public](#)
- [Protected](#)
- [Friend](#)
- [Private](#)
- [Protected Friend](#)
- [Private Protected](#)

See [Access levels in Visual Basic](#).

- `proceduremodifiers`

Optional. Can be one of the following:

- [Overloads](#)
- [Overrides](#)
- [Overridable](#)
- [NotOverridable](#)
- [MustOverride](#)
- `MustOverride Overrides`
- `NotOverridable Overrides`

- **Shared**

Optional. See [Shared](#).

- **Shadows**

Optional. See [Shadows](#).

- **Async**

Optional. See [Async](#).

- **Iterator**

Optional. See [Iterator](#).

- **name**

Required. Name of the procedure. See [Declared Element Names](#).

- **typeparamlist**

Optional. List of type parameters for a generic procedure. See [Type List](#).

- **parameterlist**

Optional. List of local variable names representing the parameters of this procedure. See [Parameter List](#).

- **returntype**

Required if `option Strict` is `On`. Data type of the value returned by this procedure.

- **Implements**

Optional. Indicates that this procedure implements one or more `Function` procedures, each one defined in an interface implemented by this procedure's containing class or structure. See [Implements Statement](#).

- **implementslist**

Required if `Implements` is supplied. List of `Function` procedures being implemented.

`implementedprocedure [, implementedprocedure ...]`

Each `implementedprocedure` has the following syntax and parts:

`interface.definedname`

PART	DESCRIPTION
<code>interface</code>	Required. Name of an interface implemented by this procedure's containing class or structure.
<code>definedname</code>	Required. Name by which the procedure is defined in <code>interface</code> .

- **Handles**

Optional. Indicates that this procedure can handle one or more specific events. See [Handles](#).

- **eventlist**

Required if `Handles` is supplied. List of events this procedure handles.

`eventspecifier [, eventspecifier ...]`

Each `eventspecifier` has the following syntax and parts:

`eventvariable.event`

PART	DESCRIPTION
<code>eventvariable</code>	Required. Object variable declared with the data type of the class or structure that raises the event.
<code>event</code>	Required. Name of the event this procedure handles.

- `statements`

Optional. Block of statements to be executed within this procedure.

- `End Function`

Terminates the definition of this procedure.

Remarks

All executable code must be inside a procedure. Each procedure, in turn, is declared within a class, a structure, or a module that is referred to as the containing class, structure, or module.

To return a value to the calling code, use a `Function` procedure; otherwise, use a `Sub` procedure.

Defining a Function

You can define a `Function` procedure only at the module level. Therefore, the declaration context for a function must be a class, a structure, a module, or an interface and can't be a source file, a namespace, a procedure, or a block. For more information, see [Declaration Contexts and Default Access Levels](#).

`Function` procedures default to public access. You can adjust their access levels with the access modifiers.

A `Function` procedure can declare the data type of the value that the procedure returns. You can specify any data type or the name of an enumeration, a structure, a class, or an interface. If you don't specify the `returntype` parameter, the procedure returns `Object`.

If this procedure uses the `Implements` keyword, the containing class or structure must also have an `Implements` statement that immediately follows its `Class` or `Structure` statement. The `Implements` statement must include each interface that's specified in `implementslist`. However, the name by which an interface defines the `Function` (in `definedname`) doesn't need to match the name of this procedure (in `name`).

NOTE

You can use lambda expressions to define function expressions inline. For more information, see [Function Expression](#) and [Lambda Expressions](#).

Returning from a Function

When the `Function` procedure returns to the calling code, execution continues with the statement that follows the statement that called the procedure.

To return a value from a function, you can either assign the value to the function name or include it in a `Return` statement.

The `Return` statement simultaneously assigns the return value and exits the function, as the following example shows.

```
Function MyFunction(ByVal j As Integer) As Double
    Return 3.87 * j
End Function
```

The following example assigns the return value to the function name `myFunction` and then uses the `Exit Function` statement to return.

```
Function MyFunction(ByVal j As Integer) As Double
    MyFunction = 3.87 * j
    Exit Function
End Function
```

The `Exit Function` and `Return` statements cause an immediate exit from a `Function` procedure. Any number of `Exit Function` and `Return` statements can appear anywhere in the procedure, and you can mix `Exit Function` and `Return` statements.

If you use `Exit Function` without assigning a value to `name`, the procedure returns the default value for the data type that's specified in `returntype`. If `returntype` isn't specified, the procedure returns `Nothing`, which is the default value for `Object`.

Calling a Function

You call a `Function` procedure by using the procedure name, followed by the argument list in parentheses, in an expression. You can omit the parentheses only if you aren't supplying any arguments. However, your code is more readable if you always include the parentheses.

You call a `Function` procedure the same way that you call any library function such as `Sqr`, `Cos`, or `ChrW`.

You can also call a function by using the `Call` keyword. In that case, the return value is ignored. Use of the `Call` keyword isn't recommended in most cases. For more information, see [Call Statement](#).

Visual Basic sometimes rearranges arithmetic expressions to increase internal efficiency. For that reason, you shouldn't use a `Function` procedure in an arithmetic expression when the function changes the value of variables in the same expression.

Async Functions

The `Async` feature allows you to invoke asynchronous functions without using explicit callbacks or manually splitting your code across multiple functions or lambda expressions.

If you mark a function with the `Async` modifier, you can use the `Await` operator in the function. When control reaches an `Await` expression in the `Async` function, control returns to the caller, and progress in the function is suspended until the awaited task completes. When the task is complete, execution can resume in the function.

NOTE

An `Async` procedure returns to the caller when either it encounters the first awaited object that's not yet complete, or it gets to the end of the `Async` procedure, whichever occurs first.

An `Async` function can have a return type of `Task<TResult>` or `Task`. An example of an `Async` function that has a return type of `Task<TResult>` is provided below.

An `Async` function cannot declare any `ByRef` parameters.

A `Sub Statement` can also be marked with the `Async` modifier. This is primarily used for event handlers, where a value cannot be returned. An `Async Sub` procedure can't be awaited, and the caller of an `Async Sub` procedure can't catch exceptions that are thrown by the `Sub` procedure.

For more information about `Async` functions, see [Asynchronous Programming with Async and Await](#), [Control Flow in Async Programs](#), and [Async Return Types](#).

Iterator Functions

An *iterator* function performs a custom iteration over a collection, such as a list or array. An iterator function uses the `Yield` statement to return each element one at a time. When a `Yield` statement is reached, the current location in code is remembered. Execution is restarted from that location the next time the iterator function is called.

You call an iterator from client code by using a `For Each...Next` statement.

The return type of an iterator function can be `IEnumerable`, `IEnumerable<T>`, `IEnumerator`, or `IEnumerator<T>`.

For more information, see [Iterators](#).

Example

The following example uses the `Function` statement to declare the name, parameters, and code that form the body of a `Function` procedure. The `ParamArray` modifier enables the function to accept a variable number of arguments.

```
Public Function CalcSum(ByVal ParamArray args() As Double) As Double
    CalcSum = 0
    If args.Length <= 0 Then Exit Function
    For i As Integer = 0 To UBound(args, 1)
        CalcSum += args(i)
    Next i
End Function
```

Example

The following example invokes the function declared in the preceding example.

```

Module Module1

Sub Main()
    ' In the following function call, CalcSum's local variables
    ' are assigned the following values: args(0) = 4, args(1) = 3,
    ' and so on. The displayed sum is 10.
    Dim returnedValue As Double = CalcSum(4, 3, 2, 1)
    Console.WriteLine("Sum: " & returnedValue)
    ' Parameter args accepts zero or more arguments. The sum
    ' displayed by the following statements is 0.
    returnedValue = CalcSum()
    Console.WriteLine("Sum: " & returnedValue)
End Sub

Public Function CalcSum(ByVal ParamArray args() As Double) As Double
    CalcSum = 0
    If args.Length <= 0 Then Exit Function
    For i As Integer = 0 To UBound(args, 1)
        CalcSum += args(i)
    Next i
End Function

End Module

```

Example

In the following example, `DelayAsync` is an `Async Function` that has a return type of `Task<TResult>`. `DelayAsync` has a `Return` statement that returns an integer. Therefore the function declaration of `DelayAsync` needs to have a return type of `Task(Of Integer)`. Because the return type is `Task(Of Integer)`, the evaluation of the `Await` expression in `DoSomethingAsync` produces an integer. This is demonstrated in this statement: `Dim result As Integer = Await delayTask`.

The `startButton_Click` procedure is an example of an `Async Sub` procedure. Because `DoSomethingAsync` is an `Async` function, the task for the call to `DoSomethingAsync` must be awaited, as the following statement demonstrates: `Await DoSomethingAsync()`. The `startButton_Click` `Sub` procedure must be defined with the `Async` modifier because it has an `Await` expression.

```
' Imports System.Diagnostics
' Imports System.Threading.Tasks

' This Click event is marked with the Async modifier.
Private Async Sub startButton_Click(sender As Object, e As RoutedEventArgs) Handles
startButton.Click
    Await DoSomethingAsync()
End Sub

Private Async Function DoSomethingAsync() As Task
    Dim delayTask As Task(Of Integer) = DelayAsync()
    Dim result As Integer = Await delayTask

    ' The previous two statements may be combined into
    ' the following statement.
    ' Dim result As Integer = Await DelayAsync()

    Debug.WriteLine("Result: " & result)
End Function

Private Async Function DelayAsync() As Task(Of Integer)
    Await Task.Delay(100)
    Return 5
End Function

' Output:
' Result: 5
```

See also

- [Sub Statement](#)
- [Function Procedures](#)
- [Parameter List](#)
- [Dim Statement](#)
- [Call Statement](#)
- [Of](#)
- [Parameter Arrays](#)
- [How to: Use a Generic Class](#)
- [Troubleshooting Procedures](#)
- [Lambda Expressions](#)
- [Function Expression](#)

Get Statement

10/18/2019 • 3 minutes to read • [Edit Online](#)

Declares a `Get` property procedure used to retrieve the value of a property.

Syntax

```
[ <attributelist> ] [ accessmodifier ] Get()
    [ statements ]
End Get
```

Parts

TERM	DEFINITION
<code>attributelist</code>	Optional. See Attribute List .
<code>accessmodifier</code>	Optional on at most one of the <code>Get</code> and <code>Set</code> statements in this property. Can be one of the following: <ul style="list-style-type: none">- Protected- Friend- Private- <code>Protected Friend</code> See Access levels in Visual Basic .
<code>statements</code>	Optional. One or more statements that run when the <code>Get</code> property procedure is called.
<code>End Get</code>	Required. Terminates the definition of the <code>Get</code> property procedure.

Remarks

Every property must have a `Get` property procedure unless the property is marked `WriteOnly`. The `Get` procedure is used to return the current value of the property.

Visual Basic automatically calls a property's `Get` procedure when an expression requests the property's value.

The body of the property declaration can contain only the property's `Get` and `Set` procedures between the [Property Statement](#) and the `End Property` statement. It cannot store anything other than those procedures. In particular, it cannot store the property's current value. You must store this value outside the property, because if you store it inside either of the property procedures, the other property procedure cannot access it. The usual approach is to store the value in a [Private](#) variable declared at the same level as the property. You must define a `Get` procedure inside the property to which it applies.

The `Get` procedure defaults to the access level of its containing property unless you use `accessmodifier` in the `Get` statement.

Rules

- **Mixed Access Levels.** If you are defining a read-write property, you can optionally specify a different access level for either the `Get` or the `Set` procedure, but not both. If you do this, the procedure access level must be more restrictive than the property's access level. For example, if the property is declared `Friend`, you can declare the `Get` procedure `Private`, but not `Public`.

If you are defining a `ReadOnly` property, the `Get` procedure represents the entire property. You cannot declare a different access level for `Get`, because that would set two access levels for the property.

- **Return Type.** The [Property Statement](#) can declare the data type of the value it returns. The `Get` procedure automatically returns that data type. You can specify any data type or the name of an enumeration, structure, class, or interface.

If the `Property` statement does not specify `returntype`, the procedure returns `Object`.

Behavior

- **Returning from a Procedure.** When the `Get` procedure returns to the calling code, execution continues within the statement that requested the property value.

`Get` property procedures can return a value using either the [Return Statement](#) or by assigning the return value to the property name. For more information, see "Return Value" in [Function Statement](#).

The `Exit Property` and `Return` statements cause an immediate exit from a property procedure. Any number of `Exit Property` and `Return` statements can appear anywhere in the procedure, and you can mix `Exit Property` and `Return` statements.

- **Return Value.** To return a value from a `Get` procedure, you can either assign the value to the property name or include it in a [Return Statement](#). The `Return` statement simultaneously assigns the `Get` procedure return value and exits the procedure.

If you use `Exit Property` without assigning a value to the property name, the `Get` procedure returns the default value for the property's data type. For more information, see "Return Value" in [Function Statement](#).

The following example illustrates two ways the read-only property `quoteForTheDay` can return the value held in the private variable `quoteValue`.

```
Private quoteValue As String = "No quote assigned yet."
```

```
ReadOnly Property QuoteForTheDay() As String
    Get
        QuoteForTheDay = quoteValue
        Exit Property
    End Get
End Property
```

```
ReadOnly Property QuoteForTheDay() As String
    Get
        Return quoteValue
    End Get
End Property
```

Example

The following example uses the `Get` statement to return the value of a property.

```
Class propClass
    ' Define a private local variable to store the property value.
    Private currentTime As String
    ' Define the read-only property.
    Public ReadOnly Property DateAndTime() As String
        Get
            ' The Get procedure is called automatically when the
            ' value of the property is retrieved.
            currentTime = CStr(Now)
            ' Return the date and time As a string.
            Return currentTime
        End Get
    End Property
End Class
```

See also

- [Set Statement](#)
- [Property Statement](#)
- [Exit Statement](#)
- [Objects and Classes](#)
- [Walkthrough: Defining Classes](#)

GoTo Statement

10/18/2019 • 2 minutes to read • [Edit Online](#)

Branches unconditionally to a specified line in a procedure.

Syntax

```
GoTo line
```

Part

line

Required. Any line label.

Remarks

The `GoTo` statement can branch only to lines in the procedure in which it appears. The line must have a line label that `GoTo` can refer to. For more information, see [How to: Label Statements](#).

NOTE

`GoTo` statements can make code difficult to read and maintain. Whenever possible, use a control structure instead. For more information, see [Control Flow](#).

You cannot use a `GoTo` statement to branch from outside a `For ... Next`, `For Each ... Next`, `SyncLock ... End SyncLock`, `Try ... Catch ... Finally`, `With ... End With`, or `Using ... End Using` construction to a label inside.

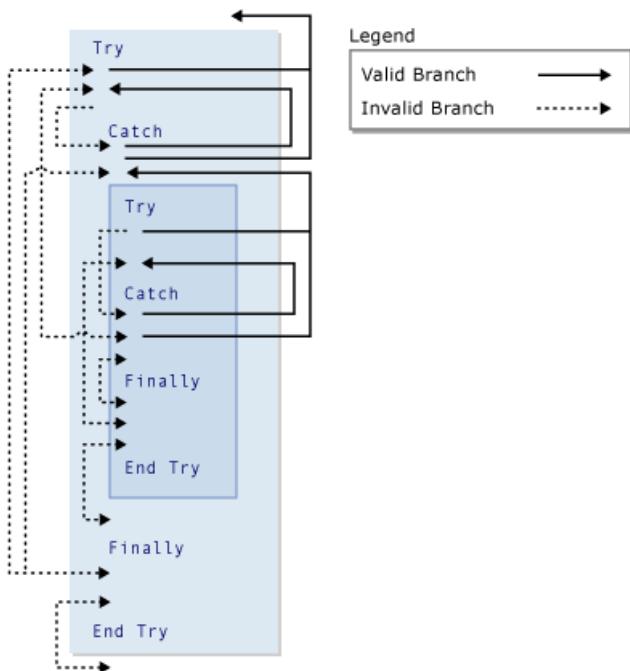
Branching and Try Constructors

Within a `Try ... Catch ... Finally` construction, the following rules apply to branching with the `GoTo` statement.

BLOCK OR REGION	BRANCHING IN FROM OUTSIDE	BRANCHING OUT FROM INSIDE
<code>Try</code> block	Only from a <code>Catch</code> block of the same construction ¹	Only to outside the whole construction
<code>Catch</code> block	Never allowed	Only to outside the whole construction, or to the <code>Try</code> block of the same construction ¹
<code>Finally</code> block	Never allowed	Never allowed

¹ If one `Try ... Catch ... Finally` construction is nested within another, a `Catch` block can branch into the `Try` block at its own nesting level, but not into any other `Try` block. A nested `Try ... Catch ... Finally` construction must be contained completely in a `Try` or `Catch` block of the construction within which it is nested.

The following illustration shows one `Try` construction nested within another. Various branches among the blocks of the two constructions are indicated as valid or invalid.



Example

The following example uses the `GoTo` statement to branch to line labels in a procedure.

```

Sub GoToStatementDemo()
    Dim number As Integer = 1
    Dim sampleString As String
    ' Evaluate number and branch to appropriate label.
    If number = 1 Then GoTo Line1 Else GoTo Line2
Line1:
    sampleString = "Number equals 1"
    GoTo LastLine
Line2:
    ' The following statement never gets executed because number = 1.
    sampleString = "Number equals 2"
LastLine:
    ' Write "Number equals 1" in the Debug window.
    Debug.WriteLine(sampleString)
End Sub

```

See also

- [Do...Loop Statement](#)
- [For...Next Statement](#)
- [For Each...Next Statement](#)
- [If...Then...Else Statement](#)
- [Select...Case Statement](#)
- [Try...Catch...Finally Statement](#)
- [While...End While Statement](#)
- [With...End With Statement](#)

If...Then...Else Statement (Visual Basic)

9/6/2019 • 5 minutes to read • [Edit Online](#)

Conditionally executes a group of statements, depending on the value of an expression.

Syntax

```
' Multiline syntax:  
If condition [ Then ]  
    [ statements ]  
[ ElseIf elseifcondition [ Then ]  
    [ elseifstatements ] ]  
[ Else  
    [ elsestatements ] ]  
End If  
  
' Single-line syntax:  
If condition Then [ statements ] [ Else [ elsestatements ] ]
```

Quick links to example code

This article includes several examples that illustrate uses of the `If ... Then ... Else` statement:

- [Multiline syntax example](#)
- [Nested syntax example](#)
- [Single-line syntax example](#)

Parts

`condition`

Required. Expression. Must evaluate to `True` or `False`, or to a data type that is implicitly convertible to `Boolean`.

If the expression is a `Nullable Boolean` variable that evaluates to `Nothing`, the condition is treated as if the expression is `False`, and the `ElseIf` blocks are evaluated if they exist, or the `Else` block is executed if it exists.

`Then`

Required in the single-line syntax; optional in the multiline syntax.

`statements`

Optional. One or more statements following `If ... Then` that are executed if `condition` evaluates to `True`.

`elseifcondition`

Required if `ElseIf` is present. Expression. Must evaluate to `True` or `False`, or to a data type that is implicitly convertible to `Boolean`.

`elseifstatements`

Optional. One or more statements following `ElseIf ... Then` that are executed if `elseifcondition` evaluates to `True`.

`elsestatements`

Optional. One or more statements that are executed if no previous `condition` or `elseifcondition` expression

evaluates to `True`.

`End If`

Terminates the multiline version of `If ... Then ... Else` block.

Remarks

Multiline syntax

When an `If ... Then ... Else` statement is encountered, `condition` is tested. If `condition` is `True`, the statements following `Then` are executed. If `condition` is `False`, each `ElseIf` statement (if there are any) is evaluated in order. When a `True` `elseifcondition` is found, the statements immediately following the associated `ElseIf` are executed. If no `elseifcondition` evaluates to `True`, or if there are no `ElseIf` statements, the statements following `Else` are executed. After executing the statements following `Then`, `ElseIf`, or `Else`, execution continues with the statement following `End If`.

The `ElseIf` and `Else` clauses are both optional. You can have as many `ElseIf` clauses as you want in an `If ... Then ... Else` statement, but no `ElseIf` clause can appear after an `Else` clause. `If ... Then ... Else` statements can be nested within each other.

In the multiline syntax, the `If` statement must be the only statement on the first line. The `ElseIf`, `Else`, and `End If` statements can be preceded only by a line label. The `If ... Then ... Else` block must end with an `End If` statement.

TIP

The [Select...Case Statement](#) might be more useful when you evaluate a single expression that has several possible values.

Single-Line syntax

You can use the single-line syntax for a single condition with code to execute if it's true. However, the multiple-line syntax provides more structure and flexibility and is easier to read, maintain, and debug.

What follows the `Then` keyword is examined to determine whether a statement is a single-line `If`. If anything other than a comment appears after `Then` on the same line, the statement is treated as a single-line `If` statement. If `Then` is absent, it must be the start of a multiple-line `If ... Then ... Else`.

In the single-line syntax, you can have multiple statements executed as the result of an `If ... Then` decision. All statements must be on the same line and be separated by colons.

Multiline syntax example

The following example illustrates the use of the multiline syntax of the `If ... Then ... Else` statement.

```

Module Multiline
    Public Sub Main()
        'Create a Random object to seed our starting value
        Dim randomizer As New Random()
        'set our variable
        Dim count As Integer = randomizer.Next(0, 5)

        Dim message As String

        'If count is zero, output will be no items
        If count = 0 Then
            message = "There are no items."
        'If count is 1, output will be "There is 1 item.".
        ElseIf count = 1 Then
            message = "There is 1 item."
        'If count is greater than 1, output will be "There are {count} items.", where {count} is replaced by
        'the value of count.
        Else
            message = $"There are {count} items."
        End If

        Console.WriteLine(message)
    End Sub
End Module
'This example displays output like the following:
' There are 4 items.

```

Nested syntax example

The following example contains nested `If ... Then ... Else` statements.

```

Module Nested
    Public Sub Main()
        ' Run the function as part of the WriteLine output.
        Console.WriteLine("Time Check is " & CheckIfTime() & ".")
    End Sub

    Private Function CheckIfTime() As Boolean
        ' Determine the current day of week and hour of day.
        Dim dayW As DayOfWeek = DateTime.Now.DayOfWeek
        Dim hour As Integer = DateTime.Now.Hour

        ' Return True if Wednesday from 2 to 3:59 P.M.,
        ' or if Thursday from noon to 12:59 P.M.
        If dayW = DayOfWeek.Wednesday Then
            If hour = 14 Or hour = 15 Then
                Return True
            Else
                Return False
            End If
        ElseIf dayW = DayOfWeek.Thursday Then
            If hour = 12 Then
                Return True
            Else
                Return False
            End If
        Else
            Return False
        End If
    End Function
End Module
'This example displays output like the following:
'Time Check is False.

```

Single-Line syntax example

The following example illustrates the use of the single-line syntax.

```
Module SingleLine
    Public Sub Main()

        'Create a Random object to seed our starting values
        Dim randomizer As New Random()

        Dim A As Integer = randomizer.Next(10, 20)
        Dim B As Integer = randomizer.Next(0, 20)
        Dim C As Integer = randomizer.Next(0, 5)

        'Let's display the initial values for comparison
        Console.WriteLine($"A value before If: {A}")
        Console.WriteLine($"B value before If: {B}")
        Console.WriteLine($"C value before If: {C}")

        ' If A > 10, execute the three colon-separated statements in the order
        ' that they appear
        If A > 10 Then A = A + 1 : B = B + A : C = C + B

        'If the condition is true, the values will be different
        Console.WriteLine($"A value after If: {A}")
        Console.WriteLine($"B value after If: {B}")
        Console.WriteLine($"C value after If: {C}")

    End Sub
End Module

'This example displays output like the following:
'A value before If: 11
'B value before If: 6
'C value before If: 3
'A value after If: 12
'B value after If: 17
'C value after If: 21
```

See also

- [Choose](#)
- [Switch](#)
- [#If...Then...#Else Directives](#)
- [Select...Case Statement](#)
- [Nested Control Structures](#)
- [Decision Structures](#)
- [Logical and Bitwise Operators in Visual Basic](#)
- [If Operator](#)

Implements Statement

10/18/2019 • 2 minutes to read • [Edit Online](#)

Specifies one or more interfaces, or interface members, that must be implemented in the class or structure definition in which it appears.

Syntax

```
Implements interfacename [, ...]  
' -or -  
Implements interfacename.interfacemember [, ...]
```

Parts

`interfacename`

Required. An interface whose properties, procedures, and events are to be implemented by corresponding members in the class or structure.

`interfacemember`

Required. The member of an interface that is being implemented.

Remarks

An interface is a collection of prototypes representing the members (properties, procedures, and events) the interface encapsulates. Interfaces contain only the declarations for members; classes and structures implement these members. For more information, see [Interfaces](#).

The `Implements` statement must immediately follow the `Class` or `Structure` statement.

When you implement an interface, you must implement all the members declared in the interface. Omitting any member is considered to be a syntax error. To implement an individual member, you specify the `Implements` keyword (which is separate from the `Implements` statement) when you declare the member in the class or structure. For more information, see [Interfaces](#).

Classes can use `Private` implementations of properties and procedures, but these members are accessible only by casting an instance of the implementing class into a variable declared to be of the type of the interface.

Example

The following example shows how to use the `Implements` statement to implement members of an interface. It defines an interface named `ICustomerInfo` with an event, a property, and a procedure. The class `customerInfo` implements all the members defined in the interface.

```

Public Interface ICustomerInfo
    Event UpdateComplete()
    Property CustomerName() As String
    Sub UpdateCustomerStatus()
End Interface

Public Class customerInfo
    Implements ICustomerInfo
    ' Storage for the property value.
    Private customerNameValue As String
    Public Event UpdateComplete() Implements ICustomerInfo.UpdateComplete
    Public Property CustomerName() As String _
        Implements ICustomerInfo.CustomerName
        Get
            Return customerNameValue
        End Get
        Set(ByVal value As String)
            ' The value parameter is passed to the Set procedure
            ' when the contents of this property are modified.
            customerNameValue = value
        End Set
    End Property

    Public Sub UpdateCustomerStatus() _
        Implements ICustomerInfo.UpdateCustomerStatus
        ' Add code here to update the status of this account.
        ' Raise an event to indicate that this procedure is done.
        RaiseEvent UpdateComplete()
    End Sub
End Class

```

Note that the class `customerInfo` uses the `Implements` statement on a separate source code line to indicate that the class implements all the members of the `ICustomerInfo` interface. Then each member in the class uses the `Implements` keyword as part of its member declaration to indicate that it implements that interface member.

Example

The following two procedures show how you could use the interface implemented in the preceding example. To test the implementation, add these procedures to your project and call the `testImplements` procedure.

```

Public Sub TestImplements()
    ' This procedure tests the interface implementation by
    ' creating an instance of the class that implements ICustomerInfo.
    Dim cust As ICustomerInfo = New customerInfo()
    ' Associate an event handler with the event that is raised by
    ' the cust object.
    AddHandler cust.UpdateComplete, AddressOf HandleUpdateComplete
    ' Set the CustomerName Property
    cust.CustomerName = "Fred"
    ' Retrieve and display the CustomerName property.
    MsgBox("Customer name is: " & cust.CustomerName)
    ' Call the UpdateCustomerStatus procedure, which raises the
    ' UpdateComplete event.
    cust.UpdateCustomerStatus()
End Sub

Sub HandleUpdateComplete()
    ' This is the event handler for the UpdateComplete event.
    MsgBox("Update is complete.")
End Sub

```

See also

- [Implements](#)
- [Interface Statement](#)
- [Interfaces](#)

Imports Statement (.NET Namespace and Type)

10/18/2019 • 4 minutes to read • [Edit Online](#)

Enables type names to be referenced without namespace qualification.

Syntax

```
Imports [ aliasname = ] namespace  
' -or-  
Imports [ aliasname = ] namespace.element
```

Parts

TERM	DEFINITION
aliasname	Optional. An <i>import alias</i> or name by which code can refer to namespace instead of the full qualification string. See Declared Element Names .
namespace	Required. The fully qualified name of the namespace being imported. Can be a string of namespaces nested to any level.
element	Optional. The name of a programming element declared in the namespace. Can be any container element.

Remarks

The `Imports` statement enables types that are contained in a given namespace to be referenced directly.

You can supply a single namespace name or a string of nested namespaces. Each nested namespace is separated from the next higher level namespace by a period (`.`), as the following example illustrates.

```
Imports System.Collections.Generic
```

Each source file can contain any number of `Imports` statements. These must follow any option declarations, such as the `Option Strict` statement, and they must precede any programming element declarations, such as `Module` or `Class` statements.

You can use `Imports` only at file level. This means the declaration context for importation must be a source file, and cannot be a namespace, class, structure, module, interface, procedure, or block.

Note that the `Imports` statement does not make elements from other projects and assemblies available to your project. Importing does not take the place of setting a reference. It only removes the need to qualify names that are already available to your project. For more information, see "Importing Containing Elements" in [References to Declared Elements](#).

NOTE

You can define implicit `Imports` statements by using the [References Page, Project Designer \(Visual Basic\)](#). For more information, see [How to: Add or Remove Imported Namespaces \(Visual Basic\)](#).

Import Aliases

An *import alias* defines the alias for a namespace or type. Import aliases are useful when you need to use items with the same name that are declared in one or more namespaces. For more information and an example, see "Qualifying an Element Name" in [References to Declared Elements](#).

You should not declare a member at module level with the same name as `aliasname`. If you do, the Visual Basic compiler uses `aliasname` only for the declared member and no longer recognizes it as an import alias.

Although the syntax used for declaring an import alias is like that used for importing an XML namespace prefix, the results are different. An import alias can be used as an expression in your code, whereas an XML namespace prefix can be used only in XML literals or XML axis properties as the prefix for a qualified element or attribute name.

Element Names

If you supply `element`, it must represent a *container element*, that is, a programming element that can contain other elements. Container elements include classes, structures, modules, interfaces, and enumerations.

The scope of the elements made available by an `Imports` statement depends on whether you specify `element`. If you specify only `namespace`, all uniquely named members of that namespace, and members of container elements within that namespace, are available without qualification. If you specify both `namespace` and `element`, only the members of that element are available without qualification.

Example

The following example returns all the folders in the C:\ directory by using the `DirectoryInfo` class.

The code has no `Imports` statements at the top of the file. Therefore, the `DirectoryInfo`, `StringBuilder`, and `CrLf` references are all fully qualified with the namespaces.

```
Public Function GetFolders() As String
    ' Create a new StringBuilder, which is used
    ' to efficiently build strings.
    Dim sb As New System.Text.StringBuilder

    Dim dInfo As New System.IO.DirectoryInfo("c:\")
    ' Obtain an array of directories, and iterate through
    ' the array.
    For Each dir As System.IO.DirectoryInfo In dInfo.GetDirectories()
        sb.Append(dir.Name)
        sb.Append(Microsoft.VisualBasic.ControlChars.CrLf)
    Next

    Return sb.ToString
End Function
```

Example

The following example includes `Imports` statements for the referenced namespaces. Therefore, the types do not have to be fully qualified with the namespaces.

```
' Place Imports statements at the top of your program.  
Imports System.Text  
Imports System.IO  
Imports Microsoft.VisualBasic.ControlChars
```

```
Public Function GetFolders() As String  
    Dim sb As New StringBuilder  
  
    Dim dInfo As New DirectoryInfo("c:\")  
    For Each dir As DirectoryInfo In dInfo.GetDirectories()  
        sb.Append(dir.Name)  
        sb.Append(CrLf)  
    Next  
  
    Return sb.ToString  
End Function
```

Example

The following example includes `Imports` statements that create aliases for the referenced namespaces. The types are qualified with the aliases.

```
Imports systxt = System.Text  
Imports sysio = System.IO  
Imports ch = Microsoft.VisualBasic.ControlChars
```

```
Public Function GetFolders() As String  
    Dim sb As New systxt.StringBuilder  
  
    Dim dInfo As New sysio.DirectoryInfo("c:\")  
    For Each dir As sysio.DirectoryInfo In dInfo.GetDirectories()  
        sb.Append(dir.Name)  
        sb.Append(ch.CrLf)  
    Next  
  
    Return sb.ToString  
End Function
```

Example

The following example includes `Imports` statements that create aliases for the referenced types. Aliases are used to specify the types.

```
Imports strbld = System.Text.StringBuilder  
Imports dirinf = System.IO.DirectoryInfo
```

```
Public Function GetFolders() As String
    Dim sb As New System.Text.StringBuilder

    Dim dInfo As New DirectoryInfo("c:\")
    For Each dir As DirectoryInfo In dInfo.GetDirectories()
        sb.Append(dir.Name)
        sb.Append(ControlChars.CrLf)
    Next

    Return sb.ToString
End Function
```

See also

- [Namespace Statement](#)
- [Namespaces in Visual Basic](#)
- [References and the Imports Statement](#)
- [Imports Statement \(XML Namespace\)](#)
- [References to Declared Elements](#)

Imports Statement (XML Namespace)

10/18/2019 • 4 minutes to read • [Edit Online](#)

Imports XML namespace prefixes for use in XML literals and XML axis properties.

Syntax

```
Imports <xmlns:xmlNamespacePrefix = "xmlNamespaceName">
```

Parts

`xmlNamespacePrefix`

Optional. The string by which XML elements and attributes can refer to `xmlNamespaceName`. If no `xmlNamespacePrefix` is supplied, the imported XML namespace is the default XML namespace. Must be a valid XML identifier. For more information, see [Names of Declared XML Elements and Attributes](#).

`xmlNamespaceName`

Required. The string identifying the XML namespace being imported.

Remarks

You can use the `Imports` statement to define global XML namespaces that you can use with XML literals and XML axis properties, or as parameters passed to the `GetXmlNamespace` operator. (For information about using the `Imports` statement to import an alias that can be used where type names are used in your code, see [Imports Statement \(.NET Namespace and Type\)](#).) The syntax for declaring an XML namespace by using the `Imports` statement is identical to the syntax used in XML. Therefore, you can copy a namespace declaration from an XML file and use it in an `Imports` statement.

XML namespace prefixes are useful when you want to repeatedly create XML elements that are from the same namespace. The XML namespace prefix declared with the `Imports` statement is global in the sense that it is available to all code in the file. You can use it when you create XML element literals and when you access XML axis properties. For more information, see [XML Element Literal](#) and [XML Axis Properties](#).

If you define a global XML namespace without a namespace prefix (for example,

```
Imports <xmlns="http://SomeNameSpace"> ), that namespace is considered the default XML namespace. The default XML namespace is used for any XML element literals or XML attribute axis properties that do not explicitly specify a namespace. The default namespace is also used if the specified namespace is the empty namespace (that is, xmlns="" ). The default XML namespace does not apply to XML attributes in XML literals or to XML attribute axis properties that do not have a namespace.
```

XML namespaces that are defined in an XML literal, which are called *local XML namespaces*, take precedence over XML namespaces that are defined by the `Imports` statement as global. XML namespaces that are defined by the `Imports` statement take precedence over XML namespaces imported for a Visual Basic project. If an XML literal defines an XML namespace, that local namespace does not apply to embedded expressions.

Global XML namespaces follow the same scoping and definition rules as .NET Framework namespaces. As a result, you can include an `Imports` statement to define a global XML namespace anywhere you can import a .NET Framework namespace. This includes both code files and project-level imported namespaces. For information about project-level imported namespaces, see [References Page, Project Designer \(Visual Basic\)](#).

Each source file can contain any number of `Imports` statements. These must follow option declarations, such as the `Option Strict` statement, and they must precede programming element declarations, such as `Module` or `Class` statements.

Example

The following example imports a default XML namespace and an XML namespace identified with the prefix `ns`. It then creates XML literals that use both namespaces.

```
' Place Imports statements at the top of your program.
Imports <xmlns="http://DefaultNamespace">
Imports <xmlns:ns="http://NewNamespace">

Module Module1

    Sub Main()
        ' Create element by using the default global XML namespace.
        Dim inner = <innerElement/>

        ' Create element by using both the default global XML namespace
        ' and the namespace identified with the "ns" prefix.
        Dim outer = <ns:outer>
            <ns:innerElement></ns:innerElement>
            <siblingElement></siblingElement>
            <%= inner %>
        </ns:outer>

        ' Display element to see its final form.
        Console.WriteLine(outer)
    End Sub

End Module
```

This code displays the following text:

```
<ns:outer xmlns="http://DefaultNamespace"
           xmlns:ns="http://NewNamespace">
    <ns:innerElement></ns:innerElement>
    <siblingElement></siblingElement>
    <innerElement />
</ns:outer>
```

Example

The following example imports the XML namespace prefix `ns`. It then creates an XML literal that uses the namespace prefix and displays the element's final form.

```

' Place Imports statements at the top of your program.
Imports <xmlns:ns="http://SomeNamespace">

Class TestClass1

    Shared Sub TestPrefix()
        ' Create test using a global XML namespace prefix.
        Dim inner2 = <ns:inner2/>

        Dim test =
            <ns:outer>
                <ns:middle xmlns:ns="http://NewNamespace">
                    <ns:inner1/>
                    <%= inner2 %>
                </ns:middle>
            </ns:outer>

        ' Display test to see its final form.
        Console.WriteLine(test)
    End Sub

End Class

```

This code displays the following text:

```

<ns:outer xmlns:ns="http://SomeNamespace">
    <ns:middle xmlns:ns="http://NewNamespace">
        <ns:inner1 />
        <inner2 xmlns="http://SomeNamespace" />
    </ns:middle>
</ns:outer>

```

Notice that the compiler converted the XML namespace prefix from a global prefix to a local prefix definition.

Example

The following example imports the XML namespace prefix `ns`. It then uses the prefix of the namespace to create an XML literal and access the first child node with the qualified name `ns:name`.

```

Imports <xmlns:ns = "http://SomeNamespace">

Class TestClass4

    Shared Sub TestPrefix()
        Dim contact = <ns:contact>
            <ns:name>Patrick Hines</ns:name>
        </ns:contact>
        Console.WriteLine(contact.<ns:name>.Value)
    End Sub

End Class

```

This code displays the following text:

`Patrick Hines`

See also

- [XML Element Literal](#)
- [XML Axis Properties](#)

- [Names of Declared XML Elements and Attributes](#)
- [GetXmlNamespace Operator](#)

Inherits Statement

10/18/2019 • 2 minutes to read • [Edit Online](#)

Causes the current class or interface to inherit the attributes, variables, properties, procedures, and events from another class or set of interfaces.

Syntax

```
Inherits basetypenames
```

Parts

TERM	DEFINITION
<code>basetypenames</code>	Required. The name of the class from which this class derives. -or- The names of the interfaces from which this interface derives. Use commas to separate multiple names.

Remarks

If used, the `Inherits` statement must be the first non-blank, non-comment line in a class or interface definition. It should immediately follow the `Class` or `Interface` statement.

You can use `Inherits` only in a class or interface. This means the declaration context for an inheritance cannot be a source file, namespace, structure, module, procedure, or block.

Rules

- **Class Inheritance.** If a class uses the `Inherits` statement, you can specify only one base class.

A class cannot inherit from a class nested within it.

- **Interface Inheritance.** If an interface uses the `Inherits` statement, you can specify one or more base interfaces. You can inherit from two interfaces even if they each define a member with the same name. If you do so, the implementing code must use name qualification to specify which member it is implementing.

An interface cannot inherit from another interface with a more restrictive access level. For example, a `Public` interface cannot inherit from a `Friend` interface.

An interface cannot inherit from an interface nested within it.

An example of class inheritance in the .NET Framework is the `ArgumentException` class, which inherits from the `SystemException` class. This provides to `ArgumentException` all the predefined properties and procedures required by system exceptions, such as the `Message` property and the `ToString` method.

An example of interface inheritance in the .NET Framework is the `ICollection` interface, which inherits from the

[IEnumerable](#) interface. This causes [ICollection](#) to inherit the definition of the enumerator required to traverse a collection.

Example

The following example uses the `Inherits` statement to show how a class named `thisClass` can inherit all the members of a base class named `anotherClass`.

```
Public Class thisClass
    Inherits anotherClass
    ' Add code to override, overload, or extend members
    ' inherited from the base class.
    ' Add new variable, property, procedure, and event declarations.
End Class
```

Example

The following example shows inheritance of multiple interfaces.

```
Public Interface thisInterface
    Inherits IComparable, IDisposable, IFormattable
    ' Add new property, procedure, and event definitions.
End Interface
```

The interface named `thisInterface` now includes all the definitions in the [IComparable](#), [IDisposable](#), and [IFormattable](#) interfaces. The inherited members provide respectively for type-specific comparison of two objects, releasing allocated resources, and expressing the value of an object as a `String`. A class that implements `thisInterface` must implement every member of every base interface.

See also

- [MustInherit](#)
- [NotInheritable](#)
- [Objects and Classes](#)
- [Inheritance Basics](#)
- [Interfaces](#)

Interface Statement (Visual Basic)

10/18/2019 • 5 minutes to read • [Edit Online](#)

Declares the name of an interface and introduces the definitions of the members that the interface comprises.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ] _  
Interface name [ ( Of typelist ) ]  
    [ Inherits interfacenames ]  
    [ [ modifiers ] Property membername ]  
    [ [ modifiers ] Function membername ]  
    [ [ modifiers ] Sub membername ]  
    [ [ modifiers ] Event membername ]  
    [ [ modifiers ] Interface membername ]  
    [ [ modifiers ] Class membername ]  
    [ [ modifiers ] Structure membername ]  
End Interface
```

Parts

TERM	DEFINITION
<code>attributelist</code>	Optional. See Attribute List .
<code>accessmodifier</code>	Optional. Can be one of the following: <ul style="list-style-type: none">- Public- Protected- Friend- Private- Protected Friend- Private Protected See Access levels in Visual Basic .
<code>Shadows</code>	Optional. See Shadows .
<code>name</code>	Required. Name of this interface. See Declared Element Names .
<code>of</code>	Optional. Specifies that this is a generic interface.
<code>typelist</code>	Required if you use the <code>Of</code> keyword. List of type parameters for this interface. Optionally, each type parameter can be declared variant by using <code>In</code> and <code>out</code> generic modifiers. See Type List .
<code>Inherits</code>	Optional. Indicates that this interface inherits the attributes and members of another interface or interfaces. See Inherits Statement .

TERM	DEFINITION
<code>interfacenames</code>	Required if you use the <code>Inherits</code> statement. The names of the interfaces from which this interface derives.
<code>modifiers</code>	Optional. Appropriate modifiers for the interface member being defined.
<code>Property</code>	Optional. Defines a property that is a member of the interface.
<code>Function</code>	Optional. Defines a <code>Function</code> procedure that is a member of the interface.
<code>Sub</code>	Optional. Defines a <code>Sub</code> procedure that is a member of the interface.
<code>Event</code>	Optional. Defines an event that is a member of the interface.
<code>Interface</code>	Optional. Defines an interface that is a nested within this interface. The nested interface definition must terminate with an <code>End Interface</code> statement.
<code>Class</code>	Optional. Defines a class that is a member of the interface. The member class definition must terminate with an <code>End Class</code> statement.
<code>Structure</code>	Optional. Defines a structure that is a member of the interface. The member structure definition must terminate with an <code>End Structure</code> statement.
<code>membername</code>	Required for each property, procedure, event, interface, class, or structure defined as a member of the interface. The name of the member.
<code>End Interface</code>	Terminates the <code>Interface</code> definition.

Remarks

An *interface* defines a set of members, such as properties and procedures, that classes and structures can implement. The interface defines only the signatures of the members and not their internal workings.

A class or structure implements the interface by supplying code for every member defined by the interface. Finally, when the application creates an instance from that class or structure, an object exists and runs in memory. For more information, see [Objects and Classes](#) and [Interfaces](#).

You can use `Interface` only at namespace or module level. This means the *declaration context* for an interface must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure or block. For more information, see [Declaration Contexts and Default Access Levels](#).

Interfaces default to [Friend](#) access. You can adjust their access levels with the access modifiers. For more information, see [Access levels in Visual Basic](#).

Rules

- **Nesting Interfaces.** You can define one interface within another. The outer interface is called the *containing interface*, and the inner interface is called a *nested interface*.
- **Member Declaration.** When you declare a property or procedure as a member of an interface, you are defining only the *signature* of that property or procedure. This includes the element type (property or procedure), its parameters and parameter types, and its return type. Because of this, the member definition uses only one line of code, and terminating statements such as `End Function` or `End Property` are not valid in an interface.

In contrast, when you define an enumeration or structure, or a nested class or interface, it is necessary to include their data members.

- **Member Modifiers.** You cannot use any access modifiers when defining module members, nor can you specify `Shared` or any procedure modifier except `Overloads`. You can declare any member with `Shadows`, and you can use `Default` when defining a property, as well as `ReadOnly` or `WriteOnly`.
- **Inheritance.** If the interface uses the `Inherits Statement`, you can specify one or more base interfaces. You can inherit from two interfaces even if they each define a member with the same name. If you do so, the implementing code must use name qualification to specify which member it is implementing.

An interface cannot inherit from another interface with a more restrictive access level. For example, a `Public` interface cannot inherit from a `Friend` interface.

An interface cannot inherit from an interface nested within it.

- **Implementation.** When a class uses the `Implements` statement to implement this interface, it must implement every member defined within the interface. Furthermore, each signature in the implementing code must exactly match the corresponding signature defined in this interface. However, the name of the member in the implementing code does not have to match the member name as defined in the interface.

When a class is implementing a procedure, it cannot designate the procedure as `Shared`.

- **Default Property.** An interface can specify at most one property as its *default property*, which can be referenced without using the property name. You specify such a property by declaring it with the `Default` modifier.

Notice that this means that an interface can define a default property only if it inherits none.

Behavior

- **Access Level.** All interface members implicitly have `Public` access. You cannot use any access modifier when defining a member. However, a class implementing the interface can declare an access level for each implemented member.

If you assign a class instance to a variable, the access level of its members can depend on whether the data type of the variable is the underlying interface or the implementing class. The following example illustrates this.

```

Public Interface IDemo
    Sub DoSomething()
End Interface
Public Class implementIDemo
    Implements IDemo
    Private Sub DoSomething() Implements IDemo.DoSomething
    End Sub
End Class
Dim varAsInterface As IDemo = New implementIDemo()
Dim varAsClass As implementIDemo = New implementIDemo()

```

If you access class members through `varAsInterface`, they all have public access. However, if you access members through `varAsClass`, the `Sub` procedure `doSomething` has private access.

- **Scope.** An interface is in scope throughout its namespace, class, structure, or module.

The scope of every interface member is the entire interface.

- **Lifetime.** An interface does not itself have a lifetime, nor do its members. When a class implements an interface and an object is created as an instance of that class, the object has a lifetime within the application in which it is running. For more information, see "Lifetime" in [Class Statement](#).

Example

The following example uses the `Interface` statement to define an interface named `thisInterface`, which must be implemented with a `Property` statement and a `Function` statement.

```

Public Interface thisInterface
    Property ThisProp(ByVal thisStr As String) As Char
    Function ThisFunc(ByVal thisInt As Integer) As Integer
End Interface

```

Note that the `Property` and `Function` statements do not introduce blocks ending with `End Property` and `End Function` within the interface. The interface defines only the signatures of its members. The full `Property` and `Function` blocks appear in a class that implements `thisInterface`.

See also

- [Interfaces](#)
- [Class Statement](#)
- [Module Statement](#)
- [Structure Statement](#)
- [Property Statement](#)
- [Function Statement](#)
- [Sub Statement](#)
- [Generic Types in Visual Basic](#)
- [Variance in Generic Interfaces](#)
- [In](#)
- [Out](#)

Mid Statement

10/18/2019 • 2 minutes to read • [Edit Online](#)

Replaces a specified number of characters in a `String` variable with characters from another string.

Syntax

```
Mid( _
    ByRef Target As String, _
    ByVal Start As Integer, _
    Optional ByVal Length As Integer _
) = StringExpression
```

Parts

`Target`

Required. Name of the `String` variable to modify.

`Start`

Required. `Integer` expression. Character position in `Target` where the replacement of text begins. `Start` uses a one-based index.

`Length`

Optional. `Integer` expression. Number of characters to replace. If omitted, all of `String` is used.

`StringExpression`

Required. `String` expression that replaces part of `Target`.

Exceptions

EXCEPTION TYPE	CONDITION
<code>ArgumentException</code>	<code>Start</code> <= 0 or <code>Length</code> < 0.

Remarks

The number of characters replaced is always less than or equal to the number of characters in `Target`.

Visual Basic has a `Mid` function and a `Mid` statement. These elements both operate on a specified number of characters in a string, but the `Mid` function returns the characters while the `Mid` statement replaces the characters. For more information, see [Mid](#).

NOTE

The `MidB` statement of earlier versions of Visual Basic replaces a substring in bytes, rather than characters. It is used primarily for converting strings in double-byte character set (DBCS) applications. All Visual Basic strings are in Unicode, and `MidB` is no longer supported.

Example

This example uses the `Mid` statement to replace a specified number of characters in a string variable with characters from another string.

```
Dim testString As String
' Initializes string.
testString = "The dog jumps"
' Returns "The fox jumps".
Mid(testString, 5, 3) = "fox"
' Returns "The cow jumps".
Mid(testString, 5) = "cow"
' Returns "The cow jumpe".
Mid(testString, 5) = "cow jumped over"
' Returns "The duc jumpe".
Mid(testString, 5, 3) = "duck"
```

Requirements

Namespace: [Microsoft.VisualBasic](#)

Module: `Strings`

Assembly: Visual Basic Runtime Library (in `Microsoft.VisualBasic.dll`)

See also

- [Mid](#)
- [Strings](#)
- [Introduction to Strings in Visual Basic](#)

Module Statement

10/18/2019 • 3 minutes to read • [Edit Online](#)

Declares the name of a module and introduces the definition of the variables, properties, events, and procedures that the module comprises.

Syntax

```
[ <attributelist> ] [ accessmodifier ] Module name  
[ statements ]  
End Module
```

Parts

`attributelist`

Optional. See [Attribute List](#).

`accessmodifier`

Optional. Can be one of the following:

- [Public](#)
- [Friend](#)

See [Access levels in Visual Basic](#).

`name`

Required. Name of this module. See [Declared Element Names](#).

`statements`

Optional. Statements which define the variables, properties, events, procedures, and nested types of this module.

`End Module`

Terminates the `Module` definition.

Remarks

A `Module` statement defines a reference type available throughout its namespace. A *module* (sometimes called a *standard module*) is similar to a class but with some important distinctions. Every module has exactly one instance and does not need to be created or assigned to a variable. Modules do not support inheritance or implement interfaces. Notice that a module is not a *type* in the sense that a class or structure is — you cannot declare a programming element to have the data type of a module.

You can use `Module` only at namespace level. This means the *declaration context* for a module must be a source file or namespace, and cannot be a class, structure, module, interface, procedure, or block. You cannot nest a module within another module, or within any type. For more information, see [Declaration Contexts and Default Access Levels](#).

A module has the same lifetime as your program. Because its members are all `Shared`, they also have lifetimes equal to that of the program.

Modules default to [Friend](#) access. You can adjust their access levels with the access modifiers. For more information, see [Access levels in Visual Basic](#).

All members of a module are implicitly [Shared](#).

Classes and Modules

These elements have many similarities, but there are some important differences as well.

- **Terminology.** Previous versions of Visual Basic recognize two types of modules: *class modules* (.cls files) and *standard modules* (.bas files). The current version calls these *classes* and *modules*, respectively.
- **Shared Members.** You can control whether a member of a class is a shared or instance member.
- **Object Orientation.** Classes are object-oriented, but modules are not. So only classes can be instantiated as objects. For more information, see [Objects and Classes](#).

Rules

- **Modifiers.** All module members are implicitly [Shared](#). You cannot use the [Shared](#) keyword when declaring a member, and you cannot alter the shared status of any member.
- **Inheritance.** A module cannot inherit from any type other than [Object](#), from which all modules inherit. In particular, one module cannot inherit from another.
You cannot use the [Inherits Statement](#) in a module definition, even to specify [Object](#).
- **Default Property.** You cannot define any default properties in a module. For more information, see [Default](#).

Behavior

- **Access Level.** Within a module, you can declare each member with its own access level. Module members default to [Public](#) access, except variables and constants, which default to [Private](#) access. When a module has more restricted access than one of its members, the specified module access level takes precedence.
- **Scope.** A module is in scope throughout its namespace.

The scope of every module member is the entire module. Notice that all members undergo *type promotion*, which causes their scope to be promoted to the namespace containing the module. For more information, see [Type Promotion](#).

- **Qualification.** You can have multiple modules in a project, and you can declare members with the same name in two or more modules. However, you must qualify any reference to such a member with the appropriate module name if the reference is from outside that module. For more information, see [References to Declared Elements](#).

Example

```
Public Module thisModule
    Sub Main()
        Dim userName As String = InputBox("What is your name?")
        MsgBox("User name is " & userName)
    End Sub
    ' Insert variable, property, procedure, and event declarations.
End Module
```

See also

- [Class Statement](#)
- [Namespace Statement](#)
- [Structure Statement](#)
- [Interface Statement](#)
- [Property Statement](#)
- [Type Promotion](#)

Namespace Statement

4/2/2019 • 4 minutes to read • [Edit Online](#)

Declares the name of a namespace and causes the source code that follows the declaration to be compiled within that namespace.

Syntax

```
Namespace [Global.] { name | name.name }
    [ componenttypes ]
End Namespace
```

Parts

Global

Optional. Allows you to define a namespace out of the root namespace of your project. See [Namespaces in Visual Basic](#).

name

Required. A unique name that identifies the namespace. Must be a valid Visual Basic identifier. For more information, see [Declared Element Names](#).

componenttypes

Optional. Elements that make up the namespace. These include, but are not limited to, enumerations, structures, interfaces, classes, modules, delegates, and other namespaces.

End Namespace

Terminates a `Namespace` block.

Remarks

Namespaces are used as an organizational system. They provide a way to classify and present programming elements that are exposed to other programs and applications. Note that a namespace is not a *type* in the sense that a class or structure is—you cannot declare a programming element to have the data type of a namespace.

All programming elements declared after a `Namespace` statement belong to that namespace. Visual Basic continues to compile elements into the last declared namespace until it encounters either an `End Namespace` statement or another `Namespace` statement.

If a namespace is already defined, even outside your project, you can add programming elements to it. To do this, you use a `Namespace` statement to direct Visual Basic to compile elements into that namespace.

You can use a `Namespace` statement only at the file or namespace level. This means the *declaration context* for a namespace must be a source file or another namespace, and cannot be a class, structure, module, interface, or procedure. For more information, see [Declaration Contexts and Default Access Levels](#).

You can declare one namespace within another. There is no strict limit to the levels of nesting you can declare, but remember that when other code accesses the elements declared in the innermost namespace, it must use a qualification string that contains all the namespace names in the nesting hierarchy.

Access Level

Namespaces are treated as if they have a `Public` access level. A namespace can be accessed from code anywhere in the same project, from other projects that reference the project, and from any assembly built from the project.

Programming elements declared at namespace level, meaning in a namespace but not inside any other element, can have `Public` or `Friend` access. If unspecified, the access level of such an element uses `Friend` by default. Elements you can declare at namespace level include classes, structures, modules, interfaces, enumerations, and delegates. For more information, see [Declaration Contexts and Default Access Levels](#).

Root Namespace

All namespace names in your project are based on a *root namespace*. Visual Studio assigns your project name as the default root namespace for all code in your project. For example, if your project is named `Payroll`, its programming elements belong to namespace `Payroll`. If you declare `Namespace funding`, the full name of that namespace is `Payroll.funding`.

If you want to specify an existing namespace in a `Namespace` statement, such as in the generic list class example, you can set your root namespace to a null value. To do this, click **Project Properties** from the **Project** menu and then clear the **Root namespace** entry so that the box is empty. If you did not do this in the generic list class example, the Visual Basic compiler would take `System.Collections.Generic` as a new namespace within project `Payroll`, with the full name of `Payroll.System.Collections.Generic`.

Alternatively, you can use the `Global` keyword to refer to elements of namespaces defined outside your project. Doing so lets you retain your project name as the root namespace. This reduces the chance of unintentionally merging your programming elements together with those of existing namespaces. For more information, see the "Global Keyword in Fully Qualified Names" section in [Namespaces in Visual Basic](#).

The `Global` keyword can also be used in a Namespace statement. This lets you define a namespace out of the root namespace of your project. For more information, see the "Global Keyword in Namespace Statements" section in [Namespaces in Visual Basic](#).

Troubleshooting. The root namespace can lead to unexpected concatenations of namespace names. If you make reference to namespaces defined outside your project, the Visual Basic compiler can construe them as nested namespaces in the root namespace. In such a case, the compiler does not recognize any types that have been already defined in the external namespaces. To avoid this, either set your root namespace to a null value as described in "Root Namespace," or use the `Global` keyword to access elements of external namespaces.

Attributes and Modifiers

You cannot apply attributes to a namespace. An attribute contributes information to the assembly's metadata, which is not meaningful for source classifiers such as namespaces.

You cannot apply any access or procedure modifiers, or any other modifiers, to a namespace. Because it is not a type, these modifiers are not meaningful.

Example

The following example declares two namespaces, one nested in the other.

```
Namespace n1
    Namespace n2
        Class a
            ' Insert class definition.
        End Class
    End Namespace
End Namespace
```

Example

The following example declares multiple nested namespaces on a single line, and it is equivalent to the previous example.

```
Namespace n1.n2
    Class a
        ' Insert class definition.
    End Class
End Namespace
```

Example

The following example accesses the class defined in the previous examples.

```
Dim instance As New n1.n2.a
```

Example

The following example defines the skeleton of a new generic list class and adds it to the [System.Collections.Generic](#) namespace.

```
Namespace System.Collections.Generic
    Class specialSortedList(Of T)
        Inherits List(Of T)
        ' Insert code to define the special generic list class.
    End Class
End Namespace
```

See also

- [Imports Statement \(.NET Namespace and Type\)](#)
- [Declared Element Names](#)
- [Namespaces in Visual Basic](#)

On Error Statement (Visual Basic)

8/27/2019 • 7 minutes to read • [Edit Online](#)

Enables an error-handling routine and specifies the location of the routine within a procedure; can also be used to disable an error-handling routine. The `On Error` statement is used in unstructured error handling and can be used instead of structured exception handling. [Structured exception handling](#) is built into .NET, is generally more efficient, and so is recommended when handling runtime errors in your application.

Without error handling or exception handling, any run-time error that occurs is fatal: an error message is displayed, and execution stops.

NOTE

The `Error` keyword is also used in the [Error Statement](#), which is supported for backward compatibility.

Syntax

```
On Error { GoTo [ line | 0 | -1 ] | Resume Next }
```

Parts

TERM	DEFINITION
<code>GoTo line</code>	Enables the error-handling routine that starts at the line specified in the required <i>line</i> argument. The <i>line</i> argument is any line label or line number. If a run-time error occurs, control branches to the specified line, making the error handler active. The specified line must be in the same procedure as the <code>On Error</code> statement or a compile-time error will occur.
<code>GoTo 0</code>	Disables enabled error handler in the current procedure and resets it to <code>Nothing</code> .
<code>GoTo -1</code>	Disables enabled exception in the current procedure and resets it to <code>Nothing</code> .
<code>Resume Next</code>	Specifies that when a run-time error occurs, control goes to the statement immediately following the statement where the error occurred, and execution continues from that point. Use this form rather than <code>on Error GoTo</code> when accessing objects.

Remarks

NOTE

We recommend that you use structured exception handling in your code whenever possible, rather than using unstructured exception handling and the `On Error` statement. For more information, see [Try...Catch...Finally Statement](#).

An "enabled" error handler is one that is turned on by an `On Error` statement. An "active" error handler is an enabled handler that is in the process of handling an error.

If an error occurs while an error handler is active (between the occurrence of the error and a `Resume`, `Exit Sub`, `Exit Function`, or `Exit Property` statement), the current procedure's error handler cannot handle the error. Control returns to the calling procedure.

If the calling procedure has an enabled error handler, it is activated to handle the error. If the calling procedure's error handler is also active, control passes back through previous calling procedures until an enabled, but inactive, error handler is found. If no such error handler is found, the error is fatal at the point at which it actually occurred.

Each time the error handler passes control back to a calling procedure, that procedure becomes the current procedure. Once an error is handled by an error handler in any procedure, execution resumes in the current procedure at the point designated by the `Resume` statement.

NOTE

An error-handling routine is not a `Sub` procedure or a `Function` procedure. It is a section of code marked by a line label or a line number.

Number Property

Error-handling routines rely on the value in the `Number` property of the `Err` object to determine the cause of the error. The routine should test or save relevant property values in the `Err` object before any other error can occur or before a procedure that might cause an error is called. The property values in the `Err` object reflect only the most recent error. The error message associated with `Err.Number` is contained in `Err.Description`.

Throw Statement

An error that is raised with the `Err.Raise` method sets the `Exception` property to a newly created instance of the `Exception` class. In order to support the raising of exceptions of derived exception types, a `Throw` statement is supported in the language. This takes a single parameter that is the exception instance to be thrown. The following example shows how these features can be used with the existing exception handling support:

```
On Error GoTo Handler
Throw New DivideByZeroException()
Handler:
  If (TypeOf Err.GetException() Is DivideByZeroException) Then
    ' Code for handling the error is entered here.
  End If
```

Notice that the `On Error GoTo` statement traps all errors, regardless of the exception class.

On Error Resume Next

`On Error Resume Next` causes execution to continue with the statement immediately following the statement that caused the run-time error, or with the statement immediately following the most recent call out of the procedure

containing the `On Error Resume Next` statement. This statement allows execution to continue despite a run-time error. You can place the error-handling routine where the error would occur rather than transferring control to another location within the procedure. An `On Error Resume Next` statement becomes inactive when another procedure is called, so you should execute an `On Error Resume Next` statement in each called routine if you want inline error handling within that routine.

NOTE

The `On Error Resume Next` construct may be preferable to `On Error GoTo` when handling errors generated during access to other objects. Checking `Err` after each interaction with an object removes ambiguity about which object was accessed by the code. You can be sure which object placed the error code in `Err.Number`, as well as which object originally generated the error (the object specified in `Err.Source`).

On Error GoTo 0

`On Error GoTo 0` disables error handling in the current procedure. It doesn't specify line 0 as the start of the error-handling code, even if the procedure contains a line numbered 0. Without an `On Error GoTo 0` statement, an error handler is automatically disabled when a procedure is exited.

On Error GoTo -1

`On Error GoTo -1` disables the exception in the current procedure. It does not specify line -1 as the start of the error-handling code, even if the procedure contains a line numbered -1. Without an `On Error GoTo -1` statement, an exception is automatically disabled when a procedure is exited.

To prevent error-handling code from running when no error has occurred, place an `Exit Sub`, `Exit Function`, or `Exit Property` statement immediately before the error-handling routine, as in the following fragment:

```
Public Sub InitializeMatrix(ByVal Var1 As Object, ByVal Var2 As Object)
    On Error GoTo ErrorHandler
    ' Insert code that might generate an error here
    Exit Sub
ErrorHandler:
    ' Insert code to handle the error here
    Resume Next
End Sub
```

Here, the error-handling code follows the `Exit Sub` statement and precedes the `End Sub` statement to separate it from the procedure flow. You can place error-handling code anywhere in a procedure.

Untrapped Errors

Untrapped errors in objects are returned to the controlling application when the object is running as an executable file. Within the development environment, untrapped errors are returned to the controlling application only if the proper options are set. See your host application's documentation for a description of which options should be set during debugging, how to set them, and whether the host can create classes.

If you create an object that accesses other objects, you should try to handle any unhandled errors they pass back. If you cannot, map the error codes in `Err.Number` to one of your own errors and then pass them back to the caller of your object. You should specify your error by adding your error code to the `vbObjectError` constant. For example, if your error code is 1052, assign it as follows:

```
Err.Number = vbObjectError + 1052
```

Caution

System errors during calls to Windows dynamic-link libraries (DLLs) do not raise exceptions and cannot be trapped with Visual Basic error trapping. When calling DLL functions, you should check each return value for success or failure (according to the API specifications), and in the event of a failure, check the value in the [Err](#) object's [LastDLError](#) property.

Example

This example first uses the [On Error GoTo](#) statement to specify the location of an error-handling routine within a procedure. In the example, an attempt to divide by zero generates error number 6. The error is handled in the error-handling routine, and control is then returned to the statement that caused the error. The [On Error GoTo 0](#) statement turns off error trapping. Then the [On Error Resume Next](#) statement is used to defer error trapping so that the context for the error generated by the next statement can be known for certain. Note that [Err.Clear](#) is used to clear the [Err](#) object's properties after the error is handled.

```
Public Sub OnErrorDemo()
    On Error GoTo ErrorHandler      ' Enable error-handling routine.
    Dim x As Integer = 32
    Dim y As Integer = 0
    Dim z As Integer
    z = x / y      ' Creates a divide by zero error
    On Error GoTo 0      ' Turn off error trapping.
    On Error Resume Next      ' Defer error trapping.
    z = x / y      ' Creates a divide by zero error again
    If Err.Number = 6 Then
        ' Tell user what happened. Then clear the Err object.
        Dim Msg As String
        Msg = "There was an error attempting to divide by zero!"
        MsgBox(Msg, , "Divide by zero error")
        Err.Clear() ' Clear Err object fields.
    End If
    Exit Sub      ' Exit to avoid handler.
ErrorHandler:  ' Error-handling routine.
    Select Case Err.Number      ' Evaluate error number.
        Case 6      ' Divide by zero error
            MsgBox("You attempted to divide by zero!")
            ' Insert code to handle this error
        Case Else
            ' Insert code to handle other situations here...
    End Select
    Resume Next      ' Resume execution at the statement immediately
                    ' following the statement where the error occurred.
End Sub
```

Requirements

Namespace: [Microsoft.VisualBasic](#)

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- [Err](#)
- [Number](#)
- [Description](#)
- [LastDLError](#)
- [End Statement](#)
- [Exit Statement](#)

- Resume Statement
- Error Messages
- Try...Catch...Finally Statement

Operator Statement

10/18/2019 • 5 minutes to read • [Edit Online](#)

Declares the operator symbol, operands, and code that define an operator procedure on a class or structure.

Syntax

```
[ <attrlist> ] Public [ Overloads ] Shared [ Shadows ] [ Widening | Narrowing ]
Operator operatorsymbol ( operand1 [, operand2 ]) [ As [ <attrlist> ] type ]
    [ statements ]
    [ statements ]
    Return returnvalue
    [ statements ]
End Operator
```

Parts

`attrlist`

Optional. See [Attribute List](#).

`Public`

Required. Indicates that this operator procedure has [Public](#) access.

`Overloads`

Optional. See [Overloads](#).

`Shared`

Required. Indicates that this operator procedure is a [Shared](#) procedure.

`Shadows`

Optional. See [Shadows](#).

`Widening`

Required for a conversion operator unless you specify `Narrowing`. Indicates that this operator procedure defines a [Widening](#) conversion. See "Widening and Narrowing Conversions" on this Help page.

`Narrowing`

Required for a conversion operator unless you specify `Widening`. Indicates that this operator procedure defines a [Narrowing](#) conversion. See "Widening and Narrowing Conversions" on this Help page.

`operatorsymbol`

Required. The symbol or identifier of the operator that this operator procedure defines.

`operand1`

Required. The name and type of the single operand of a unary operator (including a conversion operator) or the left operand of a binary operator.

`operand2`

Required for binary operators. The name and type of the right operand of a binary operator.

`operand1` and `operand2` have the following syntax and parts:

`[ByVal] operandname [As operandtype]`

PART	DESCRIPTION
<code>ByVal</code>	Optional, but the passing mechanism must be <code>ByVal</code> .
<code>operandname</code>	Required. Name of the variable representing this operand. See Declared Element Names .
<code>operandtype</code>	Optional unless <code>Option Strict</code> is <code>On</code> . Data type of this operand.

<code>type</code>	Optional unless <code>Option Strict</code> is <code>On</code> . Data type of the value the operator procedure returns.
<code>statements</code>	Optional. Block of statements that the operator procedure runs.
<code>returnvalue</code>	Required. The value that the operator procedure returns to the calling code.
<code>End Operator</code>	Required. Terminates the definition of this operator procedure.

Remarks

You can use `Operator` only in a class or structure. This means the *declaration context* for an operator cannot be a source file, namespace, module, interface, procedure, or block. For more information, see [Declaration Contexts and Default Access Levels](#).

All operators must be `Public Shared`. You cannot specify `ByRef`, `Optional`, or `ParamArray` for either operand.

You cannot use the operator symbol or identifier to hold a return value. You must use the `Return` statement, and it must specify a value. Any number of `Return` statements can appear anywhere in the procedure.

Defining an operator in this way is called *operator overloading*, whether or not you use the `Overloads` keyword. The following table lists the operators you can define.

TYPE	OPERATORS
Unary	<code>+</code> , <code>-</code> , <code>IsFalse</code> , <code>IsTrue</code> , <code>Not</code>
Binary	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>\</code> , <code>&</code> , <code>^</code> , <code>>></code> , <code><<</code> , <code>=</code> , <code><></code> , <code>></code> , <code>>=</code> , <code><</code> , <code><=</code> , <code>And</code> , <code>Like</code> , <code>Mod</code> , <code>Or</code> , <code>Xor</code>
Conversion (unary)	<code>CType</code>

Note that the `=` operator in the binary list is the comparison operator, not the assignment operator.

When you define `CType`, you must specify either `Widening` or `Narrowing`.

Matched Pairs

You must define certain operators as matched pairs. If you define either operator of such a pair, you must define the other as well. The matched pairs are the following:

- `=` and `<>`

- `>` and `<`
- `>=` and `<=`
- `IsTrue` and `IsFalse`

Data Type Restrictions

Every operator you define must involve the class or structure on which you define it. This means that the class or structure must appear as the data type of the following:

- The operand of a unary operator.
- At least one of the operands of a binary operator.
- Either the operand or the return type of a conversion operator.

Certain operators have additional data type restrictions, as follows:

- If you define the `IsTrue` and `IsFalse` operators, they must both return the `Boolean` type.
- If you define the `<<` and `>>` operators, they must both specify the `Integer` type for the `operandtype` of `operand2`.

The return type does not have to correspond to the type of either operand. For example, a comparison operator such as `=` or `<>` can return `Boolean` even if neither operand is `Boolean`.

Logical and Bitwise Operators

The `And`, `Or`, `Not`, and `Xor` operators can perform either logical or bitwise operations in Visual Basic. However, if you define one of these operators on a class or structure, you can define only its bitwise operation.

You cannot define the `AndAlso` operator directly with an `Operator` statement. However, you can use `AndAlso` if you have fulfilled the following conditions:

- You have defined `And` on the same operand types you want to use for `AndAlso`.
- Your definition of `And` returns the same type as the class or structure on which you have defined it.
- You have defined the `IsFalse` operator on the class or structure on which you have defined `And`.

Similarly, you can use `OrElse` if you have defined `Or` on the same operands, with the return type of the class or structure, and you have defined `.IsTrue` on the class or structure.

Widening and Narrowing Conversions

A *widening conversion* always succeeds at run time, while a *narrowing conversion* can fail at run time. For more information, see [Widening and Narrowing Conversions](#).

If you declare a conversion procedure to be `Widening`, your procedure code must not generate any failures. This means the following:

- It must always return a valid value of type `type`.
- It must handle all possible exceptions and other error conditions.
- It must handle any error returns from any procedures it calls.

If there is any possibility that a conversion procedure might not succeed, or that it might cause an unhandled exception, you must declare it to be `Narrowing`.

Example

The following code example uses the `Operator` statement to define the outline of a structure that includes operator procedures for the `And`, `Or`, `IsFalse`, and `IsTrue` operators. `And` and `Or` each take two operands of type `abc` and return type `abc`. `IsFalse` and `IsTrue` each take a single operand of type `abc` and return `Boolean`. These definitions allow the calling code to use `And`, `AndAlso`, `Or`, and `OrElse` with operands of type `abc`.

```
Public Structure abc
    Dim d As Date
    Public Shared Operator And(ByVal x As abc, ByVal y As abc) As abc
        Dim r As New abc
        ' Insert code to calculate And of x and y.
        Return r
    End Operator
    Public Shared Operator Or(ByVal x As abc, ByVal y As abc) As abc
        Dim r As New abc
        ' Insert code to calculate Or of x and y.
        Return r
    End Operator
    Public Shared Operator IsFalse(ByVal z As abc) As Boolean
        Dim b As Boolean
        ' Insert code to calculate IsFalse of z.
        Return b
    End Operator
    Public Shared Operator IsTrue(ByVal z As abc) As Boolean
        Dim b As Boolean
        ' Insert code to calculate IsTrue of z.
        Return b
    End Operator
End Structure
```

See also

- [IsFalse Operator](#)
- [IsTrue Operator](#)
- [Widening](#)
- [Narrowing](#)
- [Widening and Narrowing Conversions](#)
- [Operator Procedures](#)
- [How to: Define an Operator](#)
- [How to: Define a Conversion Operator](#)
- [How to: Call an Operator Procedure](#)
- [How to: Use a Class that Defines Operators](#)

Option <keyword> Statement

4/2/2019 • 2 minutes to read • [Edit Online](#)

Introduces a statement that specifies a compiler option that applies to the entire source file.

Remarks

The compiler options can control whether all variables must be explicitly declared, whether narrowing type conversions must be explicit, or whether strings should be compared as text or as binary quantities.

The `option` keyword can be used in these contexts:

[Option Compare Statement](#)

[Option Explicit Statement](#)

[Option Infer Statement](#)

[Option Strict Statement](#)

See also

- [Keywords](#)

Option Compare Statement

10/18/2019 • 3 minutes to read • [Edit Online](#)

Declares the default comparison method to use when comparing string data.

Syntax

```
Option Compare { Binary | Text }
```

Parts

TERM	DEFINITION
<code>Binary</code>	<p>Optional. Results in string comparisons based on a sort order derived from the internal binary representations of the characters.</p> <p>This type of comparison is useful especially if the strings can contain characters that are not to be interpreted as text. In this case, you do not want to bias comparisons with alphabetical equivalences, such as case insensitivity.</p>
<code>Text</code>	<p>Optional. Results in string comparisons based on a case-insensitive text sort order determined by your system's locale.</p> <p>This type of comparison is useful if your strings contain all text characters, and you want to compare them taking into account alphabetic equivalences such as case insensitivity and closely related letters. For example, you might want to consider <code>A</code> and <code>a</code> to be equal, and <code>Ä</code> and <code>ä</code> to come before <code>B</code> and <code>b</code>.</p>

Remarks

If used, the `Option Compare` statement must appear in a file before any other source code statements.

The `Option Compare` statement specifies the string comparison method (`Binary` or `Text`). The default text comparison method is `Binary`.

A `Binary` comparison compares the numeric Unicode value of each character in each string. A `Text` comparison compares each Unicode character based on its lexical meaning in the current culture.

In Microsoft Windows, sort order is determined by the code page. For more information, see [Code Pages](#).

In the following example, characters in the English/European code page (ANSI 1252) are sorted by using `Option Compare Binary`, which produces a typical binary sort order.

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

When the same characters in the same code page are sorted by using `Option Compare Text`, the following text sort order is produced.

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

When an Option Compare Statement Is Not Present

If the source code does not contain an `Option Compare` statement, the **Option Compare** setting on the [Compile Page, Project Designer \(Visual Basic\)](#) is used. If you use the command-line compiler, the setting specified by the `/optioncompare` compiler option is used.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

To set Option Compare in the IDE

1. In **Solution Explorer**, select a project. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Set the value in the **Option Compare** box.

When you create a project, the **Option Compare** setting on the **Compile** tab is set to the **Option Compare** setting in the **Options** dialog box. To change this setting, on the **Tools** menu, click **Options**. In the **Options** dialog box, expand **Projects and Solutions**, and then click **VB Defaults**. The initial default setting in **VB Defaults** is **Binary**.

To set Option Compare on the command line

- Include the `/optioncompare` compiler option in the **vbc** command.

Example

The following example uses the `Option Compare` statement to set the binary comparison as the default string comparison method. To use this code, uncomment the `Option Compare Binary` statement, and put it at the top of the source file.

```
' Option Compare Binary  
  
Console.WriteLine("A" < "a")  
' Output: True
```

Example

The following example uses the `Option Compare` statement to set the case-insensitive text sort order as the default string comparison method. To use this code, uncomment the `Option Compare Text` statement, and put it at the top of the source file.

```
' Option Compare Text  
  
Console.WriteLine("A" = "a")  
' Output: True
```

See also

- [InStr](#)

- [InStrRev](#)
- [Replace](#)
- [Split](#)
- [StrComp](#)
- [/optioncompare](#)
- [Comparison Operators](#)
- [Comparison Operators in Visual Basic](#)
- [Like Operator](#)
- [String Functions](#)
- [Option Explicit Statement](#)
- [Option Strict Statement](#)

Option Explicit Statement (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Forces explicit declaration of all variables in a file, or allows implicit declarations of variables.

Syntax

```
Option Explicit { On | Off }
```

Parts

`On`

Optional. Enables `Option Explicit` checking. If `on` or `off` is not specified, the default is `on`.

`Off`

Optional. Disables `Option Explicit` checking.

Remarks

When `Option Explicit On` or `Option Explicit` appears in a file, you must explicitly declare all variables by using the `Dim` or `ReDim` statements. If you try to use an undeclared variable name, an error occurs at compile time.

The `Option Explicit Off` statement allows implicit declaration of variables.

If used, the `Option Explicit` statement must appear in a file before any other source code statements.

NOTE

Setting `Option Explicit` to `off` is generally not a good practice. You could misspell a variable name in one or more locations, which would cause unexpected results when the program is run.

When an Option Explicit Statement Is Not Present

If the source code does not contain an `Option Explicit` statement, the **Option Explicit** setting on the [Compile Page, Project Designer \(Visual Basic\)](#) is used. If the command-line compiler is used, the `/optionexplicit` compiler option is used.

To set Option Explicit in the IDE

1. In **Solution Explorer**, select a project. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Set the value in the **Option Explicit** box.

When you create a new project, the **Option Explicit** setting on the **Compile** tab is set to the **Option Explicit** setting in the **VB Defaults** dialog box. To access the **VB Defaults** dialog box, on the **Tools** menu, click **Options**. In the **Options** dialog box, expand **Projects and Solutions**, and then click **VB Defaults**. The initial default setting in **VB Defaults** is `On`.

To set Option Explicit on the command line

- Include the `/optionexplicit` compiler option in the `vbc` command.

Example

The following example uses the `Option Explicit` statement to force explicit declaration of all variables.

Attempting to use an undeclared variable causes an error at compile time.

```
' Force explicit variable declaration.  
Option Explicit On
```

```
Dim thisVar As Integer  
thisVar = 10  
' The following assignment produces a COMPILER ERROR because  
' the variable is not declared and Option Explicit is On.  
thisInt = 10 ' causes ERROR
```

See also

- [Dim Statement](#)
- [ReDim Statement](#)
- [Option Compare Statement](#)
- [Option Strict Statement](#)
- [/optioncompare](#)
- [/optionexplicit](#)
- [/optionstrict](#)
- [Visual Basic Defaults, Projects, Options Dialog Box](#)

Option Infer Statement

10/18/2019 • 4 minutes to read • [Edit Online](#)

Enables the use of local type inference in declaring variables.

Syntax

```
Option Infer { On | Off }
```

Parts

TERM	DEFINITION
On	Optional. Enables local type inference.
Off	Optional. Disables local type inference.

Remarks

To set `Option Infer` in a file, type `Option Infer On` or `Option Infer Off` at the top of the file, before any other source code. If the value set for `Option Infer` in a file conflicts with the value set in the IDE or on the command line, the value in the file has precedence.

When you set `Option Infer` to `On`, you can declare local variables without explicitly stating a data type. The compiler infers the data type of a variable from the type of its initialization expression.

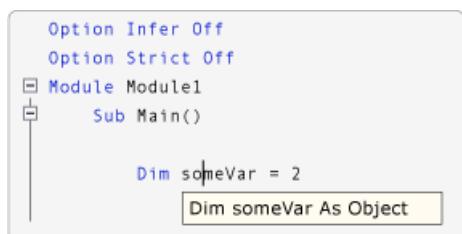
In the following illustration, `Option Infer` is turned on. The variable in the declaration `Dim someVar = 2` is declared as an integer by type inference.

The following screenshot shows IntelliSense when Option Infer is on:



In the following illustration, `Option Infer` is turned off. The variable in the declaration `Dim someVar = 2` is declared as an `Object` by type inference. In this example, the **Option Strict** setting is set to **Off** on the [Compile Page, Project Designer \(Visual Basic\)](#).

The following screenshot shows IntelliSense when Option Infer is off:



NOTE

When a variable is declared as an `Object`, the run-time type can change while the program is running. Visual Basic performs operations called *boxing* and *unboxing* to convert between an `Object` and a value type, which makes execution slower. For information about boxing and unboxing, see the [Visual Basic Language Specification](#).

Type inference applies at the procedure level, and does not apply outside a procedure in a class, structure, module, or interface.

For additional information, see [Local Type Inference](#).

When an Option Infer Statement Is Not Present

If the source code does not contain an `Option Infer` statement, the **Option Infer** setting on the [Compile Page](#), [Project Designer \(Visual Basic\)](#) is used. If the command-line compiler is used, the `/optioninfer` compiler option is used.

To set Option Infer in the IDE

1. In **Solution Explorer**, select a project. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Set the value in the **Option infer** box.

When you create a new project, the **Option Infer** setting on the **Compile** tab is set to the **Option Infer** setting in the **VB Defaults** dialog box. To access the **VB Defaults** dialog box, on the **Tools** menu, click **Options**. In the **Options** dialog box, expand **Projects and Solutions**, and then click **VB Defaults**. The initial default setting in **VB Defaults** is `On`.

To set Option Infer on the command line

Include the `/optioninfer` compiler option in the `vbc` command.

Default Data Types and Values

The following table describes the results of various combinations of specifying the data type and initializer in a `Dim` statement.

DATA TYPE SPECIFIED?	INITIALIZER SPECIFIED?	EXAMPLE	RESULT
No	No	<code>Dim qty</code>	If <code>Option Strict</code> is off (the default), the variable is set to <code>Nothing</code> . If <code>Option Strict</code> is on, a compile-time error occurs.

DATA TYPE SPECIFIED?	INITIALIZER SPECIFIED?	EXAMPLE	RESULT
No	Yes	<code>Dim qty = 5</code>	<p>If <code>Option Infer</code> is on (the default), the variable takes the data type of the initializer. See Local Type Inference.</p> <p>If <code>Option Infer</code> is off and <code>Option Strict</code> is off, the variable takes the data type of <code>Object</code>.</p> <p>If <code>Option Infer</code> is off and <code>Option Strict</code> is on, a compile-time error occurs.</p>
Yes	No	<code>Dim qty As Integer</code>	The variable is initialized to the default value for the data type. For more information, see Dim Statement .
Yes	Yes	<code>Dim qty As Integer = 5</code>	If the data type of the initializer is not convertible to the specified data type, a compile-time error occurs.

Example

The following examples demonstrate how the `Option Infer` statement enables local type inference.

```

' Enable Option Infer before trying these examples.

' Variable num is an Integer.
Dim num = 5

' Variable dbl is a Double.
Dim dbl = 4.113

' Variable str is a String.
Dim str = "abc"

' Variable pList is an array of Process objects.
Dim pList = Process.GetProcesses()

' Variable i is an Integer.
For i = 1 To 10
    Console.WriteLine(i)
Next

' Variable item is a string.
Dim lst As New List(Of String) From {"abc", "def", "ghi"}

For Each item In lst
    Console.WriteLine(item)
Next

' Variable namedCust is an instance of the Customer class.
Dim namedCust = New Customer With {.Name = "Blue Yonder Airlines",
                                    .City = "Snoqualmie"}

' Variable product is an instance of an anonymous type.
Dim product = New With {Key .Name = "paperclips", .Price = 1.29}

' If customers is a collection of Customer objects in the following
' query, the inferred type of cust is Customer, and the inferred type
' of custs is IEnumerable(Of Customer).
Dim custs = From cust In customers
            Where cust.City = "Seattle"
            Select cust.Name, cust.ID

```

Example

The following example demonstrates that the run-time type can differ when a variable is identified as an `Object`.

```

' Disable Option Infer when trying this example.

Dim someVar = 5
Console.WriteLine(someVar.GetType.ToString())

' If Option Infer is instead enabled, the following
' statement causes a run-time error. This is because
' someVar was implicitly defined as an integer.
someVar = "abc"
Console.WriteLine(someVar.GetType.ToString())

' Output:
' System.Int32
' System.String

```

See also

- [Dim Statement](#)
- [Local Type Inference](#)
- [Option Compare Statement](#)
- [Option Explicit Statement](#)
- [Option Strict Statement](#)
- [Visual Basic Defaults, Projects, Options Dialog Box](#)
- [/optioninfer](#)
- [Boxing and Unboxing](#)

Option Strict Statement

10/18/2019 • 9 minutes to read • [Edit Online](#)

Restricts implicit data type conversions to only widening conversions, disallows late binding, and disallows implicit typing that results in an `Object` type.

Syntax

```
Option Strict { On | Off }
```

Parts

TERM	DEFINITION
<code>On</code>	Optional. Enables <code>Option Strict</code> checking.
<code>Off</code>	Optional. Disables <code>Option Strict</code> checking.

Remarks

When `Option Strict On` or `Option Strict` appears in a file, the following conditions cause a compile-time error:

- Implicit narrowing conversions
- Late binding
- Implicit typing that results in an `Object` type

NOTE

In the warning configurations that you can set on the [Compile Page](#), [Project Designer \(Visual Basic\)](#), there are three settings that correspond to the three conditions that cause a compile-time error. For information about how to use these settings, see [To set warning configurations in the IDE](#) later in this topic.

The `Option Strict Off` statement turns off error and warning checking for all three conditions, even if the associated IDE settings specify to turn on these errors or warnings. The `Option Strict On` statement turns on error and warning checking for all three conditions, even if the associated IDE settings specify to turn off these errors or warnings.

If used, the `Option Strict` statement must appear before any other code statements in a file.

When you set `Option Strict` to `On`, Visual Basic checks that data types are specified for all programming elements. Data types can be specified explicitly, or specified by using local type inference. Specifying data types for all your programming elements is recommended, for the following reasons:

- It enables IntelliSense support for your variables and parameters. This enables you to see their properties and other members as you type code.
- It enables the compiler to perform type checking. Type checking helps you find statements that can

fail at run time because of type conversion errors. It also identifies calls to methods on objects that do not support those methods.

- It speeds up the execution of code. One reason for this is that if you do not specify a data type for a programming element, the Visual Basic compiler assigns it the `Object` type. Compiled code might have to convert back and forth between `Object` and other data types, which reduces performance.

Implicit Narrowing Conversion Errors

Implicit narrowing conversion errors occur when there is an implicit data type conversion that is a narrowing conversion.

Visual Basic can convert many data types to other data types. Data loss can occur when the value of one data type is converted to a data type that has less precision or a smaller capacity. A run-time error occurs if such a narrowing conversion fails. `Option Strict` ensures compile-time notification of these narrowing conversions so that you can avoid them. For more information, see [Implicit and Explicit Conversions](#) and [Widening and Narrowing Conversions](#).

Conversions that can cause errors include implicit conversions that occur in expressions. For more information, see the following topics:

- [+ Operator](#)
- [+= Operator](#)
- [\ Operator \(Visual Basic\)](#)
- [/= Operator \(Visual Basic\)](#)
- [Char Data Type](#)

When you concatenate strings by using the [& Operator](#), all conversions to the strings are considered to be widening. So these conversions do not generate an implicit narrowing conversion error, even if `Option Strict` is on.

When you call a method that has an argument that has a data type different from the corresponding parameter, a narrowing conversion causes a compile-time error if `Option Strict` is on. You can avoid the compile-time error by using a widening conversion or an explicit conversion.

Implicit narrowing conversion errors are suppressed at compile-time for conversions from the elements in a `For Each...Next` collection to the loop control variable. This occurs even if `Option Strict` is on. For more information, see the "Narrowing Conversions" section in [For Each...Next Statement](#).

Late Binding Errors

An object is late bound when it is assigned to a property or method of a variable that is declared to be of type `Object`. For more information, see [Early and Late Binding](#).

Implicit Object Type Errors

Implicit object type errors occur when an appropriate type cannot be inferred for a declared variable, so a type of `Object` is inferred. This primarily occurs when you use a `Dim` statement to declare a variable without using an `As` clause, and `Option Infer` is off. For more information, see [Option Infer Statement](#) and the [Visual Basic Language Specification](#).

For method parameters, the `As` clause is optional if `Option Strict` is off. However, if any one parameter uses an `As` clause, they all must use it. If `Option Strict` is on, the `As` clause is required for every

parameter definition.

If you declare a variable without using an `As` clause and set it to `Nothing`, the variable has a type of `Object`. No compile-time error occurs in this case when `Option Strict` is on and `Option Infer` is on. An example of this is `Dim something = Nothing`.

Default Data Types and Values

The following table describes the results of various combinations of specifying the data type and initializer in a [Dim Statement](#).

DATA TYPE SPECIFIED?	INITIALIZER SPECIFIED?	EXAMPLE	RESULT
No	No	<code>Dim qty</code>	If <code>Option Strict</code> is off (the default), the variable is set to <code>Nothing</code> . If <code>Option Strict</code> is on, a compile-time error occurs.
No	Yes	<code>Dim qty = 5</code>	If <code>Option Infer</code> is on (the default), the variable takes the data type of the initializer. See Local Type Inference . If <code>Option Infer</code> is off and <code>Option Strict</code> is off, the variable takes the data type of <code>Object</code> . If <code>Option Infer</code> is off and <code>Option Strict</code> is on, a compile-time error occurs.
Yes	No	<code>Dim qty As Integer</code>	The variable is initialized to the default value for the data type. For more information, see Dim Statement .
Yes	Yes	<code>Dim qty As Integer = 5</code>	If the data type of the initializer is not convertible to the specified data type, a compile-time error occurs.

When an Option Strict Statement Is Not Present

If the source code does not contain an `Option Strict` statement, the **Option strict** setting on the [Compile Page](#), [Project Designer \(Visual Basic\)](#) is used. The **Compile Page** has settings that provide additional control over the conditions that generate an error.

If you are using the command-line compiler, you can use the `/optionstrict` compiler option to specify a setting for `Option Strict`.

To set Option Strict in the IDE

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

1. In **Solution Explorer**, select a project. On the **Project** menu, click **Properties**.
2. On the **Compile** tab, set the value in the **Option Strict** box.

To set warning configurations in the IDE

When you use the [Compile Page](#), [Project Designer \(Visual Basic\)](#) instead of an `Option Strict` statement, you have additional control over the conditions that generate errors. The **Warning configurations** section of the **Compile Page** has settings that correspond to the three conditions that cause a compile-time error when `Option Strict` is on. Following are these settings:

- **Implicit conversion**
- **Late binding; call could fail at run time**
- **Implicit type; object assumed**

When you set **Option Strict** to **On**, all three of these warning configuration settings are set to **Error**. When you set **Option Strict** to **Off**, all three settings are set to **None**.

You can individually change each warning configuration setting to **None**, **Warning**, or **Error**. If all three warning configuration settings are set to **Error**, `On` appears in the `Option Strict` box. If all three are set to **None**, `Off` appears in this box. For any other combination of these settings, **(custom)** appears.

To set the Option Strict default setting for new projects

When you create a project, the **Option Strict** setting on the **Compile** tab is set to the **Option Strict** setting in the **Options** dialog box.

To set `Option Strict` in this dialog box, on the **Tools** menu, click **Options**. In the **Options** dialog box, expand **Projects and Solutions**, and then click **VB Defaults**. The initial default setting in **VB Defaults** is `Off`.

To set Option Strict on the command line

Include the `/optionstrict` compiler option in the **vbc** command.

Example

The following examples demonstrate compile-time errors caused by implicit type conversions that are narrowing conversions. This category of errors corresponds to the **Implicit conversion** condition on the **Compile Page**.

```

' If Option Strict is on, this implicit narrowing
' conversion causes a compile-time error.
' The commented statements below use explicit
' conversions to avoid a compile-time error.
Dim cyclists As Long = 5
Dim bicycles As Integer = cyclists
'Dim bicycles As Integer = CType(cyclists, Integer)
'Dim bicycles As Integer = CInt(cyclists)
'Dim bicycles As Integer = Convert.ToInt32(cyclists)

' If Option Strict is on, this implicit narrowing
' conversion causes a compile-time error.
' The commented statements below use explicit
' conversions to avoid a compile-time error.
Dim charVal As Char = "a"
'Dim charVal As Char = "a"c
'Dim charVal As Char = CType("a", Char)

' If Option Strict is on, a compile-time error occurs.
' If Option Strict is off, the string is implicitly converted
' to a Double, and then is added to the other number.
Dim myAge As Integer = "34" + 6

' If Option Strict is on, a compile-time error occurs.
' If Option Strict is off, the floating-point number
' is implicitly converted to a Long.
Dim num = 123.45 \ 10

```

Example

The following example demonstrates a compile-time error caused by late binding. This category of errors corresponds to the **Late binding; call could fail at run time** condition on the **Compile Page**.

```

' If Option Strict is on, this late binding
' causes a compile-time error. If Option Strict
' is off, the late binding instead causes a
' run-time error.
Dim punchCard As New Object
punchCard.Column = 5

```

Example

The following examples demonstrate errors caused by variables that are declared with an implicit type of **Object**. This category of errors corresponds to the **Implicit type; object assumed** condition on the **Compile Page**.

```

' If Option Strict is on and Option Infer is off,
' this Dim statement without an As clause
' causes a compile-time error.
Dim cardReaders = 5

' If Option Strict is on, a compile-time error occurs.
' If Option Strict is off, the variable is set to Nothing.
Dim dryWall

```

```
' If Option Strict is on, this parameter without an
' As clause causes a compile-time error.
Private Sub DetectIntergalacticRange(ByVal photonAttenuation)

End Sub
```

See also

- [Widening and Narrowing Conversions](#)
- [Implicit and Explicit Conversions](#)
- [Compile Page, Project Designer \(Visual Basic\)](#)
- [Option Explicit Statement](#)
- [Type Conversion Functions](#)
- [How to: Access Members of an Object](#)
- [Embedded Expressions in XML](#)
- [Relaxed Delegate Conversion](#)
- [Late Binding in Office Solutions](#)
- [/optionstrict](#)
- [Visual Basic Defaults, Projects, Options Dialog Box](#)

Property Statement

9/27/2019 • 5 minutes to read • [Edit Online](#)

Declares the name of a property, and the property procedures used to store and retrieve the value of the property.

Syntax

```
[ <attributelist> ] [ Default ] [ accessmodifier ]
[ propertymodifiers ] [ Shared ] [ Shadows ] [ ReadOnly | WriteOnly ] [ Iterator ]
Property name ( [ parameterlist ] ) [ As returntype ] [ Implements implementslist ]
    [ <attributelist> ] [ accessmodifier ] Get
        [ statements ]
    End Get
    [ <attributelist> ] [ accessmodifier ] Set ( ByVal value As returntype [, parameterlist ] )
        [ statements ]
    End Set
End Property
- or -
[ <attributelist> ] [ Default ] [ accessmodifier ]
[ propertymodifiers ] [ Shared ] [ Shadows ] [ ReadOnly | WriteOnly ]
Property name ( [ parameterlist ] ) [ As returntype ] [ Implements implementslist ]
```

Parts

- **attributelist**

Optional. List of attributes that apply to this property or **Get** or **Set** procedure. See [Attribute List](#).

- **Default**

Optional. Specifies that this property is the default property for the class or structure on which it is defined. Default properties must accept parameters and can be set and retrieved without specifying the property name. If you declare the property as **Default**, you cannot use **Private** on the property or on either of its property procedures.

- **accessmodifier**

Optional on the **Property** statement and on at most one of the **Get** and **Set** statements. Can be one of the following:

- [Public](#)
- [Protected](#)
- [Friend](#)
- [Private](#)
- [Protected Friend](#)
- [Private Protected](#)

See [Access levels in Visual Basic](#).

- `propertymodifiers`

Optional. Can be one of the following:

- [Overloads](#)
- [Overrides](#)
- [Overridable](#)
- [NotOverridable](#)
- [MustOverride](#)
- `MustOverride Overrides`
- `NotOverridable Overrides`

- `Shared`

Optional. See [Shared](#).

- `Shadows`

Optional. See [Shadows](#).

- `ReadOnly`

Optional. See [ReadOnly](#).

- `WriteOnly`

Optional. See [WriteOnly](#).

- `Iterator`

Optional. See [Iterator](#).

- `name`

Required. Name of the property. See [Declared Element Names](#).

- `parameterlist`

Optional. List of local variable names representing the parameters of this property, and possible additional parameters of the `Set` procedure. See [Parameter List](#).

- `returntype`

Required if `option Strict` is `on`. Data type of the value returned by this property.

- `Implements`

Optional. Indicates that this property implements one or more properties, each one defined in an interface implemented by this property's containing class or structure. See [Implements Statement](#).

- `implementslist`

Required if `Implements` is supplied. List of properties being implemented.

`implementedproperty [, implementedproperty ...]`

Each `implementedproperty` has the following syntax and parts:

`interface.definedname`

PART	DESCRIPTION
<code>interface</code>	Required. Name of an interface implemented by this property's containing class or structure.
<code>definedname</code>	Required. Name by which the property is defined in <code>interface</code> .

- `Get`

Optional. Required if the property is marked `ReadOnly`. Starts a `Get` property procedure that is used to return the value of the property. The `Get` statement is not used with [auto-implemented properties](#).

- `statements`

Optional. Block of statements to run within the `Get` or `Set` procedure.

- `End Get`

Terminates the `Get` property procedure.

- `Set`

Optional. Required if the property is marked `WriteOnly`. Starts a `Set` property procedure that is used to store the value of the property. The `Set` statement is not used with [auto-implemented properties](#).

- `End Set`

Terminates the `Set` property procedure.

- `End Property`

Terminates the definition of this property.

Remarks

The `Property` statement introduces the declaration of a property. A property can have a `Get` procedure (read only), a `Set` procedure (write only), or both (read-write). You can omit the `Get` and `Set` procedure when using an auto-implemented property. For more information, see [Auto-Implemented Properties](#).

You can use `Property` only at class level. This means the *declaration context* for a property must be a class, structure, module, or interface, and cannot be a source file, namespace, procedure, or block. For more information, see [Declaration Contexts and Default Access Levels](#).

By default, properties use public access. You can adjust a property's access level with an access modifier on the `Property` statement, and you can optionally adjust one of its property procedures to a more restrictive access level.

Visual Basic passes a parameter to the `Set` procedure during property assignments. If you do not supply a parameter for `Set`, the integrated development environment (IDE) uses an implicit parameter named `value`. This parameter holds the value to be assigned to the property. You typically store this value in a private local variable and return it whenever the `Get` procedure is called.

Rules

- **Mixed Access Levels.** If you are defining a read-write property, you can optionally specify a different access level for either the `Get` or the `Set` procedure, but not both. If you do this, the procedure access level must be more restrictive than the property's access level. For example, if the property is declared `Friend`, you can declare the `Set` procedure `Private`, but not `Public`.

If you are defining a `ReadOnly` or `WriteOnly` property, the single property procedure (`Get` or `Set`, respectively) represents all of the property. You cannot declare a different access level for such a procedure, because that would set two access levels for the property.

- **Return Type.** The `Property` statement can declare the data type of the value it returns. You can specify any data type or the name of an enumeration, structure, class, or interface.

If you do not specify `returntype`, the property returns `Object`.

- **Implementation.** If this property uses the `Implements` keyword, the containing class or structure must have an `Implements` statement immediately following its `Class` or `Structure` statement. The `Implements` statement must include each interface specified in `implementslist`. However, the name by which an interface defines the `Property` (in `definedname`) does not have to be the same as the name of this property (in `name`).

Behavior

- **Returning from a Property Procedure.** When the `Get` or `Set` procedure returns to the calling code, execution continues with the statement following the statement that invoked it.

The `Exit Property` and `Return` statements cause an immediate exit from a property procedure. Any number of `Exit Property` and `Return` statements can appear anywhere in the procedure, and you can mix `Exit Property` and `Return` statements.

- **Return Value.** To return a value from a `Get` procedure, you can either assign the value to the property name or include it in a `Return` statement. The following example assigns the return value to the property name `quoteForTheDay` and then uses the `Exit Property` statement to return.

```
Private quoteValue As String = "No quote assigned yet."
```

```
ReadOnly Property QuoteForTheDay() As String
    Get
        QuoteForTheDay = quoteValue
        Exit Property
    End Get
End Property
```

If you use `Exit Property` without assigning a value to `name`, the `Get` procedure returns the default value for the property's data type.

The `Return` statement at the same time assigns the `Get` procedure return value and exits the procedure. The following example shows this.

```
Private quoteValue As String = "No quote assigned yet."
```

```
ReadOnly Property QuoteForTheDay() As String
    Get
        Return quoteValue
    End Get
End Property
```

Example

The following example declares a property in a class.

```
Class Class1
    ' Define a local variable to store the property value.
    Private propertyValue As String
    ' Define the property.
    Public Property Prop1() As String
        Get
            ' The Get property procedure is called when the value
            ' of a property is retrieved.
            Return propertyValue
        End Get
        Set(ByVal value As String)
            ' The Set property procedure is called when the value
            ' of a property is modified. The value to be assigned
            ' is passed in the argument to Set.
            propertyValue = value
        End Set
    End Property
End Class
```

See also

- [Auto-implemented Properties](#)
- [Objects and Classes](#)
- [Get Statement](#)
- [Set Statement](#)
- [Parameter List](#)
- [Default](#)

Q-Z Statements

4/2/2019 • 2 minutes to read • [Edit Online](#)

The following table contains a listing of Visual Basic language statements.

RaiseEvent	ReDim	REM	RemoveHandler
Resume	Return	Select...Case	Set
Stop	Structure	Sub	SyncLock
Then	Throw	Try...Catch...Finally	Using
While...End While	With...End With	Yield	

See also

- [A-E Statements](#)
- [F-P Statements](#)
- [Visual Basic Language Reference](#)

RaiseEvent Statement

10/18/2019 • 3 minutes to read • [Edit Online](#)

Triggers an event declared at module level within a class, form, or document.

Syntax

```
RaiseEvent eventname[( argumentlist )]
```

Parts

`eventname`

Required. Name of the event to trigger.

`argumentlist`

Optional. Comma-delimited list of variables, arrays, or expressions. The `argumentlist` argument must be enclosed by parentheses. If there are no arguments, the parentheses must be omitted.

Remarks

The required `eventname` is the name of an event declared within the module. It follows Visual Basic variable naming conventions.

If the event has not been declared within the module in which it is raised, an error occurs. The following code fragment illustrates an event declaration and a procedure in which the event is raised.

```
' Declare an event at module level.  
Event LogonCompleted(ByVal UserName As String)  
  
Sub Logon(ByVal UserName As String)  
    ' Raise the event.  
    RaiseEvent LogonCompleted(UserName)  
End Sub
```

You cannot use `RaiseEvent` to raise events that are not explicitly declared in the module. For example, all forms inherit a `Click` event from `System.Windows.Forms.Form`, it cannot be raised using `RaiseEvent` in a derived form. If you declare a `Click` event in the form module, it shadows the form's own `Click` event. You can still invoke the form's `Click` event by calling the `OnClick` method.

By default, an event defined in Visual Basic raises its event handlers in the order that the connections are established. Because events can have `ByRef` parameters, a process that connects late may receive parameters that have been changed by an earlier event handler. After the event handlers execute, control is returned to the subroutine that raised the event.

NOTE

Non-shared events should not be raised within the constructor of the class in which they are declared. Although such events do not cause run-time errors, they may fail to be caught by associated event handlers. Use the `Shared` modifier to create a shared event if you need to raise an event from a constructor.

NOTE

You can change the default behavior of events by defining a custom event. For custom events, the `RaiseEvent` statement invokes the event's `RaiseEvent` accessor. For more information on custom events, see [Event Statement](#).

Example

The following example uses events to count down seconds from 10 to 0. The code illustrates several of the event-related methods, properties, and statements, including the `RaiseEvent` statement.

The class that raises an event is the event source, and the methods that process the event are the event handlers. An event source can have multiple handlers for the events it generates. When the class raises the event, that event is raised on every class that has elected to handle events for that instance of the object.

The example also uses a form (`Form1`) with a button (`Button1`) and a text box (`TextBox1`). When you click the button, the first text box displays a countdown from 10 to 0 seconds. When the full time (10 seconds) has elapsed, the first text box displays "Done".

The code for `Form1` specifies the initial and terminal states of the form. It also contains the code executed when events are raised.

To use this example, open a new Windows Application project, add a button named `Button1` and a text box named `TextBox1` to the main form, named `Form1`. Then right-click the form and click **View Code** to open the Code Editor.

Add a `WithEvents` variable to the declarations section of the `Form1` class.

```
Private WithEvents mText As TimerState
```

Example

Add the following code to the code for `Form1`. Replace any duplicate procedures that may exist, such as `Form_Load`, or `Button_Click`.

```

Private Sub Form1_Load() Handles MyBase.Load
    Button1.Text = "Start"
    mText = New TimerState
End Sub

Private Sub Button1_Click() Handles Button1.Click
    mText.StartCountdown(10.0, 0.1)
End Sub

Private Sub mText_ChangeText() Handles mText.Finished
    TextBox1.Text = "Done"
End Sub

Private Sub mTextUpdateTime(ByVal Countdown As Double
) Handles mText.UpdateTime

    TextBox1.Text = Format(Countdown, "##0.0")
    ' Use DoEvents to allow the display to refresh.
    My.Application.DoEvents()
End Sub

Class TimerState
    Public Event UpdateTime(ByVal Countdown As Double)
    Public Event Finished()
    Public Sub StartCountdown(ByVal Duration As Double,
        ByVal Increment As Double)
        Dim Start As Double = DateAndTime.Timer
        Dim ElapsedTime As Double = 0

        Dim SoFar As Double = 0
        Do While ElapsedTime < Duration
            If ElapsedTime > SoFar + Increment Then
                SoFar += Increment
                RaiseEvent UpdateTime(Duration - SoFar)
            End If
            ElapsedTime = DateAndTime.Timer - Start
        Loop
        RaiseEvent Finished()
    End Sub
End Class

```

Press F5 to run the preceding example, and click the button labeled **Start**. The first text box starts to count down the seconds. When the full time (10 seconds) has elapsed, the first text box displays "Done".

NOTE

The `My.Application.DoEvents` method does not process events in exactly the same way as the form does. To allow the form to handle the events directly, you can use multithreading. For more information, see [Managed Threading](#).

See also

- [Events](#)
- [Event Statement](#)
- [AddHandler Statement](#)
- [RemoveHandler Statement](#)
- [Handles](#)

ReDim Statement (Visual Basic)

10/18/2019 • 4 minutes to read • [Edit Online](#)

Reallocates storage space for an array variable.

Syntax

```
ReDim [ Preserve ] name(boundlist) [ , name(boundlist) [, ...] ]
```

Parts

TERM	DEFINITION
Preserve	Optional. Modifier used to preserve the data in the existing array when you change the size of only the last dimension.
name	Required. Name of the array variable. See Declared Element Names .
boundlist	Required. List of bounds of each dimension of the redefined array.

Remarks

You can use the `ReDim` statement to change the size of one or more dimensions of an array that has already been declared. If you have a large array and you no longer need some of its elements, `ReDim` can free up memory by reducing the array size. On the other hand, if your array needs more elements, `ReDim` can add them.

The `ReDim` statement is intended only for arrays. It's not valid on scalars (variables that contain only a single value), collections, or structures. Note that if you declare a variable to be of type `Array`, the `ReDim` statement doesn't have sufficient type information to create the new array.

You can use `ReDim` only at procedure level. Therefore, the declaration context for the variable must be a procedure; it can't be a source file, a namespace, an interface, a class, a structure, a module, or a block. For more information, see [Declaration Contexts and Default Access Levels](#).

Rules

- **Multiple Variables.** You can resize several array variables in the same declaration statement and specify the `name` and `boundlist` parts for each variable. Multiple variables are separated by commas.
- **Array Bounds.** Each entry in `boundlist` can specify the lower and upper bounds of that dimension. The lower bound is always 0 (zero). The upper bound is the highest possible index value for that dimension, not the length of the dimension (which is the upper bound plus one). The index for each dimension can vary from 0 through its upper bound value.

The number of dimensions in `boundlist` must match the original number of dimensions (rank) of the array.

- **Data Types.** The `ReDim` statement cannot change the data type of an array variable or its elements.

- **Initialization.** The `ReDim` statement cannot provide new initialization values for the array elements.
- **Rank.** The `ReDim` statement cannot change the rank (the number of dimensions) of the array.
- **Resizing with Preserve.** If you use `Preserve`, you can resize only the last dimension of the array. For every other dimension, you must specify the bound of the existing array.
For example, if your array has only one dimension, you can resize that dimension and still preserve all the contents of the array, because you are changing the last and only dimension. However, if your array has two or more dimensions, you can change the size of only the last dimension if you use `Preserve`.
- **Properties.** You can use `ReDim` on a property that holds an array of values.

Behavior

- **Array Replacement.** `ReDim` releases the existing array and creates a new array with the same rank. The new array replaces the released array in the array variable.
- **Initialization without Preserve.** If you do not specify `Preserve`, `ReDim` initializes the elements of the new array by using the default value for their data type.
- **Initialization with Preserve.** If you specify `Preserve`, Visual Basic copies the elements from the existing array to the new array.

Example

The following example increases the size of the last dimension of a dynamic array without losing any existing data in the array, and then decreases the size with partial data loss. Finally, it decreases the size back to its original value and reinitializes all the array elements.

```
Dim intArray(10, 10, 10) As Integer
ReDim Preserve intArray(10, 10, 20)
ReDim Preserve intArray(10, 10, 15)
ReDim intArray(10, 10, 10)
```

The `Dim` statement creates a new array with three dimensions. Each dimension is declared with a bound of 10, so the array index for each dimension can range from 0 through 10. In the following discussion, the three dimensions are referred to as layer, row, and column.

The first `ReDim` creates a new array which replaces the existing array in variable `intArray`. `ReDim` copies all the elements from the existing array into the new array. It also adds 10 more columns to the end of every row in every layer and initializes the elements in these new columns to 0 (the default value of `Integer`, which is the element type of the array).

The second `ReDim` creates another new array and copies all the elements that fit. However, five columns are lost from the end of every row in every layer. This is not a problem if you have finished using these columns. Reducing the size of a large array can free up memory that you no longer need.

The third `ReDim` creates another new array and removes another five columns from the end of every row in every layer. This time it does not copy any existing elements. This statement reverts the array to its original size. Because the statement doesn't include the `Preserve` modifier, it sets all array elements to their original default values.

For additional examples, see [Arrays](#).

See also

- [IndexOutOfRangeException](#)
- [Const Statement](#)
- [Dim Statement](#)
- [Erase Statement](#)
- [Nothing](#)
- [Arrays](#)

REM Statement (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Used to include explanatory remarks in the source code of a program.

Syntax

```
REM comment  
' comment
```

Parts

`comment`

Optional. The text of any comment you want to include. A space is required between the `REM` keyword and `comment`.

Remarks

You can put a `REM` statement alone on a line, or you can put it on a line following another statement. The `REM` statement must be the last statement on the line. If it follows another statement, the `REM` must be separated from that statement by a space.

You can use a single quotation mark (`'`) instead of `REM`. This is true whether your comment follows another statement on the same line or sits alone on a line.

NOTE

You cannot continue a `REM` statement by using a line-continuation sequence (`_`). Once a comment begins, the compiler does not examine the characters for special meaning. For a multiple-line comment, use another `REM` statement or a comment symbol (`'`) on each line.

Example

The following example illustrates the `REM` statement, which is used to include explanatory remarks in a program. It also shows the alternative of using the single quotation-mark character (`'`) instead of `REM`.

```
Dim demoStr1, demoStr2 As String  
demoStr1 = "Hello" REM Comment after a statement using REM.  
demoStr2 = "Goodbye" ' Comment after a statement using the ' character.  
REM This entire line is a comment.  
' This entire line is also a comment.
```

See also

- [Comments in Code](#)
- [How to: Break and Combine Statements in Code](#)

RemoveHandler Statement

10/18/2019 • 2 minutes to read • [Edit Online](#)

Removes the association between an event and an event handler.

Syntax

```
RemoveHandler event, AddressOf evenhandler
```

Parts

TERM	DEFINITION
event	The name of the event being handled.
evenhandler	The name of the procedure currently handling the event.

Remarks

The `AddHandler` and `RemoveHandler` statements allow you to start and stop event handling for a specific event at any time during program execution.

NOTE

For custom events, the `RemoveHandler` statement invokes the event's `RemoveHandler` accessor. For more information on custom events, see [Event Statement](#).

Example

```
Sub TestEvents()
    Dim Obj As New Class1
    ' Associate an event handler with an event.
    AddHandler Obj.Ev_Event, AddressOf EventHandler
    ' Call the method to raise the event.
    Obj.CauseSomeEvent()
    ' Stop handling events.
    RemoveHandler Obj.Ev_Event, AddressOf EventHandler
    ' This event will not be handled.
    Obj.CauseSomeEvent()
End Sub

Sub EventHandler()
    ' Handle the event.
    MsgBox("EventHandler caught event.")
End Sub

Public Class Class1
    ' Declare an event.
    Public Event Ev_Event()
    Sub CauseSomeEvent()
        ' Raise an event.
        RaiseEvent Ev_Event()
    End Sub
End Class
```

See also

- [AddHandler Statement](#)
- [Handles](#)
- [Event Statement](#)
- [Events](#)

Resume Statement

10/18/2019 • 2 minutes to read • [Edit Online](#)

Resumes execution after an error-handling routine is finished.

We suggest that you use structured exception handling in your code whenever possible, rather than using unstructured exception handling and the `On Error` and `Resume` statements. For more information, see [Try...Catch...Finally Statement](#).

Syntax

```
Resume [ Next | line ]
```

Parts

`Resume`

Required. If the error occurred in the same procedure as the error handler, execution resumes with the statement that caused the error. If the error occurred in a called procedure, execution resumes at the statement that last called out of the procedure containing the error-handling routine.

`Next`

Optional. If the error occurred in the same procedure as the error handler, execution resumes with the statement immediately following the statement that caused the error. If the error occurred in a called procedure, execution resumes with the statement immediately following the statement that last called out of the procedure containing the error-handling routine (or `On Error Resume Next` statement).

`line`

Optional. Execution resumes at the line specified in the required `line` argument. The `line` argument is a line label or line number and must be in the same procedure as the error handler.

Remarks

NOTE

We recommend that you use structured exception handling in your code whenever possible, rather than using unstructured exception handling and the `On Error` and `Resume` statements. For more information, see [Try...Catch...Finally Statement](#).

If you use a `Resume` statement anywhere other than in an error-handling routine, an error occurs.

The `Resume` statement cannot be used in any procedure that contains a `Try...Catch...Finally` statement.

Example

This example uses the `Resume` statement to end error handling in a procedure and then resume execution with the statement that caused the error. Error number 55 is generated to illustrate use of the `Resume` statement.

```
Sub ResumeStatementDemo()
    On Error GoTo ErrorHandler    ' Enable error-handling routine.
    Dim x As Integer = 32
    Dim y As Integer = 0
    Dim z As Integer
    z = x / y      ' Creates a divide by zero error
    Exit Sub      ' Exit Sub to avoid error handler.
ErrorHandler:      ' Error-handling routine.
    Select Case Err.Number      ' Evaluate error number.
        Case 6      "Divide by zero" error.
            y = 1 ' Sets the value of y to 1 and tries the calculation again.
        Case Else
            ' Handle other situations here....
    End Select
    Resume      ' Resume execution at same line
    ' that caused the error.
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- [Try...Catch...Finally Statement](#)
- [Error Statement](#)
- [On Error Statement](#)

Return Statement (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Returns control to the code that called a `Function`, `Sub`, `Get`, `Set`, or `Operator` procedure.

Syntax

```
Return  
' -or-  
Return expression
```

Part

`expression`

Required in a `Function`, `Get`, or `Operator` procedure. Expression that represents the value to be returned to the calling code.

Remarks

In a `Sub` or `Set` procedure, the `Return` statement is equivalent to an `Exit Sub` or `Exit Property` statement, and `expression` must not be supplied.

In a `Function`, `Get`, or `Operator` procedure, the `Return` statement must include `expression`, and `expression` must evaluate to a data type that is convertible to the return type of the procedure. In a `Function` or `Get` procedure, you also have the alternative of assigning an expression to the procedure name to serve as the return value, and then executing an `Exit Function` or `Exit Property` statement. In an `Operator` procedure, you must use `Return expression`.

You can include as many `Return` statements as appropriate in the same procedure.

NOTE

The code in a `Finally` block runs after a `Return` statement in a `Try` or `Catch` block is encountered, but before that `Return` statement executes. A `Return` statement cannot be included in a `Finally` block.

Example

The following example uses the `Return` statement several times to return to the calling code when the procedure does not have to do anything else.

```
Public Function GetAgePhrase(ByVal age As Integer) As String  
    If age > 60 Then Return "Senior"  
    If age > 40 Then Return "Middle-aged"  
    If age > 20 Then Return "Adult"  
    If age > 12 Then Return "Teen-aged"  
    If age > 4 Then Return "School-aged"  
    If age > 1 Then Return "Toddler"  
    Return "Infant"  
End Function
```

See also

- [Function Statement](#)
- [Sub Statement](#)
- [Get Statement](#)
- [Set Statement](#)
- [Operator Statement](#)
- [Property Statement](#)
- [Exit Statement](#)
- [Try...Catch...Finally Statement](#)

Select...Case Statement (Visual Basic)

10/18/2019 • 4 minutes to read • [Edit Online](#)

Runs one of several groups of statements, depending on the value of an expression.

Syntax

```
Select [ Case ] testexpression
    [ Case expressionlist
        [ statements ] ]
    [ Case Else
        [ elsestatements ] ]
End Select
```

Parts

TERM	DEFINITION
<code>testexpression</code>	Required. Expression. Must evaluate to one of the elementary data types (<code>Boolean</code> , <code>Byte</code> , <code>Char</code> , <code>Date</code> , <code>Double</code> , <code>Decimal</code> , <code>Integer</code> , <code>Long</code> , <code>Object</code> , <code>SByte</code> , <code>Short</code> , <code>Single</code> , <code>String</code> , <code>UInteger</code> , <code>ULong</code> , and <code>UShort</code>).

TERM	DEFINITION
<code>expressionlist</code>	<p>Required in a <code>Case</code> statement. List of expression clauses representing match values for <code>testexpression</code>. Multiple expression clauses are separated by commas. Each clause can take one of the following forms:</p> <ul style="list-style-type: none"> - <code>expression1 To expression2</code> - [<code>Is</code>] <code>comparisonoperator expression</code> - <code>expression</code> <p>Use the <code>To</code> keyword to specify the boundaries of a range of match values for <code>testexpression</code>. The value of <code>expression1</code> must be less than or equal to the value of <code>expression2</code>.</p> <p>Use the <code>Is</code> keyword with a comparison operator (<code>=</code>, <code><></code>, <code><</code>, <code><=</code>, <code>></code>, or <code>>=</code>) to specify a restriction on the match values for <code>testexpression</code>. If the <code>Is</code> keyword is not supplied, it is automatically inserted before <code>comparisonoperator</code>.</p> <p>The form specifying only <code>expression</code> is treated as a special case of the <code>Is</code> form where <code>comparisonoperator</code> is the equal sign (<code>=</code>). This form is evaluated as <code>testexpression = expression</code>.</p> <p>The expressions in <code>expressionlist</code> can be of any data type, provided they are implicitly convertible to the type of <code>testexpression</code> and the appropriate <code>comparisonoperator</code> is valid for the two types it is being used with.</p>
<code>statements</code>	Optional. One or more statements following <code>Case</code> that run if <code>testexpression</code> matches any clause in <code>expressionlist</code> .
<code>elsestatements</code>	Optional. One or more statements following <code>Case Else</code> that run if <code>testexpression</code> does not match any clause in the <code>expressionlist</code> of any of the <code>Case</code> statements.
<code>End Select</code>	Terminates the definition of the <code>Select ... Case</code> construction.

Remarks

If `testexpression` matches any `Case expressionlist` clause, the statements following that `Case` statement run up to the next `Case`, `Case Else`, or `End Select` statement. Control then passes to the statement following `End Select`. If `testexpression` matches an `expressionlist` clause in more than one `Case` clause, only the statements following the first match run.

The `Case Else` statement is used to introduce the `elsestatements` to run if no match is found between the `testexpression` and an `expressionlist` clause in any of the other `Case` statements. Although not required, it is a good idea to have a `Case Else` statement in your `Select Case` construction to handle unforeseen `testexpression` values. If no `Case expressionlist` clause matches `testexpression` and there is no `Case Else` statement, control passes to the statement following `End Select`.

You can use multiple expressions or ranges in each `Case` clause. For example, the following line is valid.

```
Case 1 To 4, 7 To 9, 11, 13, Is > maxNumber
```

NOTE

The `Is` keyword used in the `Case` and `Case Else` statements is not the same as the [Is Operator](#), which is used for object reference comparison.

You can specify ranges and multiple expressions for character strings. In the following example, `Case` matches any string that is exactly equal to "apples", has a value between "nuts" and "soup" in alphabetical order, or contains the exact same value as the current value of `testItem`.

```
Case "apples", "nuts" To "soup", testItem
```

The setting of `Option Compare` can affect string comparisons. Under `Option Compare Text`, the strings "Apples" and "apples" compare as equal, but under `Option Compare Binary`, they do not.

NOTE

A `Case` statement with multiple clauses can exhibit behavior known as *short-circuiting*. Visual Basic evaluates the clauses from left to right, and if one produces a match with `testexpression`, the remaining clauses are not evaluated. Short-circuiting can improve performance, but it can produce unexpected results if you are expecting every expression in `expressionlist` to be evaluated. For more information on short-circuiting, see [Boolean Expressions](#).

If the code within a `Case` or `Case Else` statement block does not need to run any more of the statements in the block, it can exit the block by using the `Exit Select` statement. This transfers control immediately to the statement following `End Select`.

`Select Case` constructions can be nested. Each nested `Select Case` construction must have a matching `End Select` statement and must be completely contained within a single `case` or `Case Else` statement block of the outer `Select Case` construction within which it is nested.

Example

The following example uses a `Select Case` construction to write a line corresponding to the value of the variable `number`. The second `Case` statement contains the value that matches the current value of `number`, so the statement that writes "Between 6 and 8, inclusive" runs.

```
Dim number As Integer = 8
Select Case number
    Case 1 To 5
        Debug.WriteLine("Between 1 and 5, inclusive")
        ' The following is the only Case clause that evaluates to True.
    Case 6, 7, 8
        Debug.WriteLine("Between 6 and 8, inclusive")
    Case 9 To 10
        Debug.WriteLine("Equal to 9 or 10")
    Case Else
        Debug.WriteLine("Not between 1 and 10, inclusive")
End Select
```

See also

- [Choose](#)

- End Statement
- If...Then...Else Statement
- Option Compare Statement
- Exit Statement

Set Statement (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Declares a `Set` property procedure used to assign a value to a property.

Syntax

```
[ <attributelist> ] [ accessmodifier ] Set (ByVal value [ As datatype ])
    [ statements ]
End Set
```

Parts

`attributelist`

Optional. See [Attribute List](#).

`accessmodifier`

Optional on at most one of the `Get` and `Set` statements in this property. Can be one of the following:

- [Protected](#)
- [Friend](#)
- [Private](#)
- `Protected Friend`

See [Access levels in Visual Basic](#).

`value`

Required. Parameter containing the new value for the property.

`datatype`

Required if `Option Strict` is `On`. Data type of the `value` parameter. The data type specified must be the same as the data type of the property where this `Set` statement is declared.

`statements`

Optional. One or more statements that run when the `Set` property procedure is called.

`End Set`

Required. Terminates the definition of the `Set` property procedure.

Remarks

Every property must have a `Set` property procedure unless the property is marked `ReadOnly`. The `Set` procedure is used to set the value of the property.

Visual Basic automatically calls a property's `Set` procedure when an assignment statement provides a value to be stored in the property.

Visual Basic passes a parameter to the `Set` procedure during property assignments. If you do not supply a parameter for `Set`, the integrated development environment (IDE) uses an implicit parameter named `value`. The parameter holds the value to be assigned to the property. You typically store this value in a private local

variable and return it whenever the `Get` procedure is called.

The body of the property declaration can contain only the property's `Get` and `Set` procedures between the [Property Statement](#) and the `End Property` statement. It cannot store anything other than those procedures. In particular, it cannot store the property's current value. You must store this value outside the property, because if you store it inside either of the property procedures, the other property procedure cannot access it. The usual approach is to store the value in a [Private](#) variable declared at the same level as the property. You must define a `Set` procedure inside the property to which it applies.

The `Set` procedure defaults to the access level of its containing property unless you use `accessmodifier` in the `Set` statement.

Rules

- **Mixed Access Levels.** If you are defining a read-write property, you can optionally specify a different access level for either the `Get` or the `Set` procedure, but not both. If you do this, the procedure access level must be more restrictive than the property's access level. For example, if the property is declared `Friend`, you can declare the `Set` procedure `Private`, but not `Public`.

If you are defining a `WriteOnly` property, the `Set` procedure represents the entire property. You cannot declare a different access level for `Set`, because that would set two access levels for the property.

Behavior

- **Returning from a Property Procedure.** When the `Set` procedure returns to the calling code, execution continues following the statement that provided the value to be stored.

`Set` property procedures can return using either the [Return Statement](#) or the [Exit Statement](#).

The `Exit Property` and `Return` statements cause an immediate exit from a property procedure. Any number of `Exit Property` and `Return` statements can appear anywhere in the procedure, and you can mix `Exit Property` and `Return` statements.

Example

The following example uses the `Set` statement to set the value of a property.

```
Class propClass
    Private propVal As Integer
    Property Prop1() As Integer
        Get
            Return propVal
        End Get
        Set(ByVal value As Integer)
            propVal = value
        End Set
    End Property
End Class
```

See also

- [Get Statement](#)
- [Property Statement](#)
- [Sub Statement](#)
- [Property Procedures](#)

Stop Statement (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Suspends execution.

Syntax

```
Stop
```

Remarks

You can place `stop` statements anywhere in procedures to suspend execution. Using the `Stop` statement is similar to setting a breakpoint in the code.

The `stop` statement suspends execution, but unlike `End`, it does not close any files or clear any variables, unless it is encountered in a compiled executable (.exe) file.

NOTE

If the `Stop` statement is encountered in code that is running outside of the integrated development environment (IDE), the debugger is invoked. This is true regardless of whether the code was compiled in debug or retail mode.

Example

This example uses the `Stop` statement to suspend execution for each iteration through the `For...Next` loop.

```
Dim i As Integer
For i = 1 To 10
    Debug.WriteLine(i)
    ' Stop during each iteration and wait for user to resume.
    Stop
Next i
```

See also

- [End Statement](#)

Structure Statement

10/18/2019 • 5 minutes to read • [Edit Online](#)

Declares the name of a structure and introduces the definition of the variables, properties, events, and procedures that the structure comprises.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ] [ Partial ] _  
Structure name [ ( Of typelist ) ]  
    [ Implements interfacenames ]  
    [ datamemberdeclarations ]  
    [ methodmemberdeclarations ]  
End Structure
```

Parts

TERM	DEFINITION
<code>attributelist</code>	Optional. See Attribute List .
<code>accessmodifier</code>	Optional. Can be one of the following: <ul style="list-style-type: none">- Public- Protected- Friend- Private- Protected Friend- Private Protected See Access levels in Visual Basic .
<code>Shadows</code>	Optional. See Shadows .
<code>Partial</code>	Optional. Indicates a partial definition of the structure. See Partial .
<code>name</code>	Required. Name of this structure. See Declared Element Names .
<code>of</code>	Optional. Specifies that this is a generic structure.
<code>typelist</code>	Required if you use the <code>Of</code> keyword. List of type parameters for this structure. See Type List .
<code>Implements</code>	Optional. Indicates that this structure implements the members of one or more interfaces. See Implements Statement .
<code>interfacenames</code>	Required if you use the <code>Implements</code> statement. The names of the interfaces this structure implements.

TERM	DEFINITION
<code>datamemberdeclarations</code>	Required. Zero or more <code>Const</code> , <code>Dim</code> , <code>Enum</code> , or <code>Event</code> statements declaring <i>data members</i> of the structure.
<code>methodmemberdeclarations</code>	Optional. Zero or more declarations of <code>Function</code> , <code>Operator</code> , <code>Property</code> , or <code>Sub</code> procedures, which serve as <i>method members</i> of the structure.
<code>End Structure</code>	Required. Terminates the <code>Structure</code> definition.

Remarks

The `Structure` statement defines a composite value type that you can customize. A *structure* is a generalization of the user-defined type (UDT) of previous versions of Visual Basic. For more information, see [Structures](#).

Structures support many of the same features as classes. For example, structures can have properties and procedures, they can implement interfaces, and they can have parameterized constructors. However, there are significant differences between structures and classes in areas such as inheritance, declarations, and usage. Also, classes are reference types and structures are value types. For more information, see [Structures and Classes](#).

You can use `Structure` only at namespace or module level. This means the *declaration context* for a structure must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure or block. For more information, see [Declaration Contexts and Default Access Levels](#).

Structures default to `Friend` access. You can adjust their access levels with the access modifiers. For more information, see [Access levels in Visual Basic](#).

Rules

- **Nesting.** You can define one structure within another. The outer structure is called the *containing structure*, and the inner structure is called a *nested structure*. However, you cannot access a nested structure's members through the containing structure. Instead, you must declare a variable of the nested structure's data type.

- **Member Declaration.** You must declare every member of a structure. A structure member cannot be `Protected` or `Protected Friend` because nothing can inherit from a structure. The structure itself, however, can be `Protected` or `Protected Friend`.

You can declare zero or more nonshared variables or nonshared, noncustom events in a structure. You cannot have only constants, properties, and procedures, even if some of them are nonshared.

- **Initialization.** You cannot initialize the value of any nonshared data member of a structure as part of its declaration. You must either initialize such a data member by means of a parameterized constructor on the structure, or assign a value to the member after you have created an instance of the structure.

- **Inheritance.** A structure cannot inherit from any type other than `ValueType`, from which all structures inherit. In particular, one structure cannot inherit from another.

You cannot use the [Inherits Statement](#) in a structure definition, even to specify `ValueType`.

- **Implementation.** If the structure uses the [Implements Statement](#), you must implement every member defined by every interface you specify in `interfacenames`.

- **Default Property.** A structure can specify at most one property as its *default property*, using the [Default](#) modifier. For more information, see [Default](#).

Behavior

- **Access Level.** Within a structure, you can declare each member with its own access level. All structure members default to [Public](#) access. Note that if the structure itself has a more restricted access level, this automatically restricts access to its members, even if you adjust their access levels with the access modifiers.

- **Scope.** A structure is in scope throughout its containing namespace, class, structure, or module.

The scope of every structure member is the entire structure.

- **Lifetime.** A structure does not itself have a lifetime. Rather, each instance of that structure has a lifetime independent of all other instances.

The lifetime of an instance begins when it is created by a [New Operator](#) clause. It ends when the lifetime of the variable that holds it ends.

You cannot extend the lifetime of a structure instance. An approximation to static structure functionality is provided by a module. For more information, see [Module Statement](#).

Structure members have lifetimes depending on how and where they are declared. For more information, see "Lifetime" in [Class Statement](#).

- **Qualification.** Code outside a structure must qualify a member's name with the name of that structure.

If code inside a nested structure makes an unqualified reference to a programming element, Visual Basic searches for the element first in the nested structure, then in its containing structure, and so on out to the outermost containing element. For more information, see [References to Declared Elements](#).

- **Memory Consumption.** As with all composite data types, you cannot safely calculate the total memory consumption of a structure by adding together the nominal storage allocations of its members. Furthermore, you cannot safely assume that the order of storage in memory is the same as your order of declaration. If you need to control the storage layout of a structure, you can apply the [StructLayoutAttribute](#) attribute to the [Structure](#) statement.

Example

The following example uses the [Structure](#) statement to define a set of related data for an employee. It shows the use of [Public](#), [Friend](#), and [Private](#) members to reflect the sensitivity of the data items. It also shows procedure, property, and event members.

```

Public Structure employee
    ' Public members, accessible from throughout declaration region.
    Public firstName As String
    Public middleName As String
    Public lastName As String
    ' Friend members, accessible from anywhere within the same assembly.
    Friend employeeNumber As Integer
    Friend workPhone As Long
    ' Private members, accessible only from within the structure itself.
    Private homePhone As Long
    Private level As Integer
    Private salary As Double
    Private bonus As Double
    ' Procedure member, which can access structure's private members.
    Friend Sub CalculateBonus(ByVal rate As Single)
        bonus = salary * CDbl(rate)
    End Sub
    ' Property member to return employee's eligibility.
    Friend ReadOnly Property Eligible() As Boolean
        Get
            Return level >= 25
        End Get
    End Property
    ' Event member, raised when business phone number has changed.
    Public Event ChangedWorkPhone(ByVal newPhone As Long)
End Structure

```

See also

- [Class Statement](#)
- [Interface Statement](#)
- [Module Statement](#)
- [Dim Statement](#)
- [Const Statement](#)
- [Enum Statement](#)
- [Event Statement](#)
- [Operator Statement](#)
- [Property Statement](#)
- [Structures and Classes](#)

Sub Statement (Visual Basic)

10/18/2019 • 6 minutes to read • [Edit Online](#)

Declares the name, parameters, and code that define a `Sub` procedure.

Syntax

```
[ <attributelist> ] [ Partial ] [ accessmodifier ] [ proceduremodifiers ] [ Shared ] [ Shadows ] [ Async ]
Sub name [ (Of typeparamlist) ] [ (parameterlist) ] [ Implements implementslist | Handles eventlist ]
    [ statements ]
    [ Exit Sub ]
    [ statements ]
End Sub
```

Parts

- `attributelist`

Optional. See [Attribute List](#).

- `Partial`

Optional. Indicates definition of a partial method. See [Partial Methods](#).

- `accessmodifier`

Optional. Can be one of the following:

- [Public](#)
- [Protected](#)
- [Friend](#)
- [Private](#)
- [Protected Friend](#)
- [Private Protected](#)

See [Access levels in Visual Basic](#).

- `proceduremodifiers`

Optional. Can be one of the following:

- [Overloads](#)
- [Overrides](#)
- [Overridable](#)
- [NotOverridable](#)
- [MustOverride](#)

- `MustOverride Overrides`
- `NotOverridable Overrides`

- `Shared`

Optional. See [Shared](#).

- `Shadows`

Optional. See [Shadows](#).

- `Async`

Optional. See [Async](#).

- `name`

Required. Name of the procedure. See [Declared Element Names](#). To create a constructor procedure for a class, set the name of a `Sub` procedure to the `New` keyword. For more information, see [Object Lifetime: How Objects Are Created and Destroyed](#).

- `typeparamlist`

Optional. List of type parameters for a generic procedure. See [Type List](#).

- `parameterlist`

Optional. List of local variable names representing the parameters of this procedure. See [Parameter List](#).

- `Implements`

Optional. Indicates that this procedure implements one or more `Sub` procedures, each one defined in an interface implemented by this procedure's containing class or structure. See [Implements Statement](#).

- `implementslist`

Required if `Implements` is supplied. List of `Sub` procedures being implemented.

```
implementedprocedure [ , implementedprocedure ... ]
```

Each `implementedprocedure` has the following syntax and parts:

```
interface.definedname
```

PART	DESCRIPTION
<code>interface</code>	Required. Name of an interface implemented by this procedure's containing class or structure.
<code>definedname</code>	Required. Name by which the procedure is defined in <code>interface</code> .

- `Handles`

Optional. Indicates that this procedure can handle one or more specific events. See [Handles](#).

- `eventlist`

Required if `Handles` is supplied. List of events this procedure handles.

```
eventspecifier [ , eventspecifier ... ]
```

Each `eventspecifier` has the following syntax and parts:

```
eventvariable.event
```

PART	DESCRIPTION
<code>eventvariable</code>	Required. Object variable declared with the data type of the class or structure that raises the event.
<code>event</code>	Required. Name of the event this procedure handles.

- `statements`

Optional. Block of statements to run within this procedure.

- `End Sub`

Terminates the definition of this procedure.

Remarks

All executable code must be inside a procedure. Use a `Sub` procedure when you don't want to return a value to the calling code. Use a `Function` procedure when you want to return a value.

Defining a Sub Procedure

You can define a `Sub` procedure only at the module level. The declaration context for a sub procedure must, therefore, be a class, a structure, a module, or an interface and can't be a source file, a namespace, a procedure, or a block. For more information, see [Declaration Contexts and Default Access Levels](#).

`Sub` procedures default to public access. You can adjust their access levels by using the access modifiers.

If the procedure uses the `Implements` keyword, the containing class or structure must have an `Implements` statement that immediately follows its `Class` or `Structure` statement. The `Implements` statement must include each interface that's specified in `implementslist`. However, the name by which an interface defines the `Sub` (in `definedname`) doesn't have to match the name of this procedure (in `name`).

Returning from a Sub Procedure

When a `Sub` procedure returns to the calling code, execution continues with the statement after the statement that called it.

The following example shows a return from a `Sub` procedure.

```
Sub mySub(ByVal q As String)
    Return
End Sub
```

The `Exit Sub` and `Return` statements cause an immediate exit from a `Sub` procedure. Any number of `Exit Sub` and `Return` statements can appear anywhere in the procedure, and you can mix `Exit Sub` and `Return` statements.

Calling a Sub Procedure

You call a `Sub` procedure by using the procedure name in a statement and then following that name with its argument list in parentheses. You can omit the parentheses only if you don't supply any arguments. However, your code is more readable if you always include the parentheses.

A `Sub` procedure and a `Function` procedure can have parameters and perform a series of statements. However, a `Function` procedure returns a value, and a `Sub` procedure doesn't. Therefore, you can't use a `Sub` procedure in an expression.

You can use the `Call` keyword when you call a `Sub` procedure, but that keyword isn't recommended for most uses. For more information, see [Call Statement](#).

Visual Basic sometimes rearranges arithmetic expressions to increase internal efficiency. For that reason, if your argument list includes expressions that call other procedures, you shouldn't assume that those expressions will be called in a particular order.

Async Sub Procedures

By using the Async feature, you can invoke asynchronous functions without using explicit callbacks or manually splitting your code across multiple functions or lambda expressions.

If you mark a procedure with the `Async` modifier, you can use the `Await` operator in the procedure. When control reaches an `Await` expression in the `Async` procedure, control returns to the caller, and progress in the procedure is suspended until the awaited task completes. When the task is complete, execution can resume in the procedure.

NOTE

An `Async` procedure returns to the caller when either the first awaited object that's not yet complete is encountered or the end of the `Async` procedure is reached, whichever occurs first.

You can also mark a [Function Statement](#) with the `Async` modifier. An `Async` function can have a return type of `Task<TResult>` or `Task`. An example later in this topic shows an `Async` function that has a return type of `Task<TResult>`.

`Async Sub` procedures are primarily used for event handlers, where a value can't be returned. An `Async Sub` procedure can't be awaited, and the caller of an `Async Sub` procedure can't catch exceptions that the `Sub` procedure throws.

An `Async` procedure can't declare any `ByRef` parameters.

For more information about `Async` procedures, see [Asynchronous Programming with Async and Await](#), [Control Flow in Async Programs](#), and [Async Return Types](#).

Example

The following example uses the `Sub` statement to define the name, parameters, and code that form the body of a `Sub` procedure.

```

Sub ComputeArea(ByVal length As Double, ByVal width As Double)
    ' Declare local variable.
    Dim area As Double
    If length = 0 Or width = 0 Then
        ' If either argument = 0 then exit Sub immediately.
        Exit Sub
    End If
    ' Calculate area of rectangle.
    area = length * width
    ' Print area to Immediate window.
    Debug.WriteLine(area)
End Sub

```

Example

In the following example, `DelayAsync` is an `Async Function` that has a return type of `Task<TResult>`. `DelayAsync` has a `Return` statement that returns an integer. Therefore, the function declaration of `DelayAsync` must have a return type of `Task(Of Integer)`. Because the return type is `Task(Of Integer)`, the evaluation of the `Await` expression in `DoSomethingAsync` produces an integer, as the following statement shows: `Dim result As Integer = Await delayTask`.

The `startButton_Click` procedure is an example of an `Async Sub` procedure. Because `DoSomethingAsync` is an `Async` function, the task for the call to `DoSomethingAsync` must be awaited, as the following statement shows: `Await DoSomethingAsync()`. The `startButton_Click` `Sub` procedure must be defined with the `Async` modifier because it has an `Await` expression.

```

' Imports System.Diagnostics
' Imports System.Threading.Tasks

' This Click event is marked with the Async modifier.
Private Async Sub startButton_Click(sender As Object, e As RoutedEventArgs) Handles
    startButton.Click
    Await DoSomethingAsync()
End Sub

Private Async Function DoSomethingAsync() As Task
    Dim delayTask As Task(Of Integer) = DelayAsync()
    Dim result As Integer = Await delayTask

    ' The previous two statements may be combined into
    ' the following statement.
    ' Dim result As Integer = Await DelayAsync()

    Debug.WriteLine("Result: " & result)
End Function

Private Async Function DelayAsync() As Task(Of Integer)
    Await Task.Delay(100)
    Return 5
End Function

' Output:
' Result: 5

```

See also

- [Implements Statement](#)
- [Function Statement](#)
- [Parameter List](#)

- [Dim Statement](#)
- [Call Statement](#)
- [Of](#)
- [Parameter Arrays](#)
- [How to: Use a Generic Class](#)
- [Troubleshooting Procedures](#)
- [Partial Methods](#)

SyncLock Statement

10/18/2019 • 5 minutes to read • [Edit Online](#)

Acquires an exclusive lock for a statement block before executing the block.

Syntax

```
SyncLock lockobject  
  [ block ]  
End SyncLock
```

Parts

`lockobject`

Required. Expression that evaluates to an object reference.

`block`

Optional. Block of statements that are to execute when the lock is acquired.

`End SyncLock`

Terminates a `SyncLock` block.

Remarks

The `SyncLock` statement ensures that multiple threads do not execute the statement block at the same time.

`SyncLock` prevents each thread from entering the block until no other thread is executing it.

The most common use of `SyncLock` is to protect data from being updated by more than one thread simultaneously. If the statements that manipulate the data must go to completion without interruption, put them inside a `SyncLock` block.

A statement block protected by an exclusive lock is sometimes called a *critical section*.

Rules

- Branching. You cannot branch into a `SyncLock` block from outside the block.
- Lock Object Value. The value of `lockobject` cannot be `Nothing`. You must create the lock object before you use it in a `SyncLock` statement.

You cannot change the value of `lockobject` while executing a `SyncLock` block. The mechanism requires that the lock object remain unchanged.

- You can't use the `Await` operator in a `SyncLock` block.

Behavior

- Mechanism. When a thread reaches the `SyncLock` statement, it evaluates the `lockobject` expression and suspends execution until it acquires an exclusive lock on the object returned by the expression. When another thread reaches the `SyncLock` statement, it does not acquire a lock until the first thread executes the `End SyncLock` statement.

- Protected Data. If `lockobject` is a `Shared` variable, the exclusive lock prevents a thread in any instance of the class from executing the `SyncLock` block while any other thread is executing it. This protects data that is shared among all the instances.

If `lockobject` is an instance variable (not `shared`), the lock prevents a thread running in the current instance from executing the `SyncLock` block at the same time as another thread in the same instance. This protects data maintained by the individual instance.
- Acquisition and Release. A `SyncLock` block behaves like a `Try...Finally` construction in which the `Try` block acquires an exclusive lock on `lockobject` and the `Finally` block releases it. Because of this, the `SyncLock` block guarantees release of the lock, no matter how you exit the block. This is true even in the case of an unhandled exception.
- Framework Calls. The `SyncLock` block acquires and releases the exclusive lock by calling the `Enter` and `Exit` methods of the `Monitor` class in the `System.Threading` namespace.

Programming Practices

The `lockobject` expression should always evaluate to an object that belongs exclusively to your class. You should declare a `Private` object variable to protect data belonging to the current instance, or a `Private Shared` object variable to protect data common to all instances.

You should not use the `Me` keyword to provide a lock object for instance data. If code external to your class has a reference to an instance of your class, it could use that reference as a lock object for a `SyncLock` block completely different from yours, protecting different data. In this way, your class and the other class could block each other from executing their unrelated `SyncLock` blocks. Similarly locking on a string can be problematic since any other code in the process using the same string will share the same lock.

You should also not use the `Me.GetType` method to provide a lock object for shared data. This is because `GetType` always returns the same `Type` object for a given class name. External code could call `GetType` on your class and obtain the same lock object you are using. This would result in the two classes blocking each other from their `SyncLock` blocks.

Examples

Description

The following example shows a class that maintains a simple list of messages. It holds the messages in an array and the last used element of that array in a variable. The `addAnotherMessage` procedure increments the last element and stores the new message. Those two operations are protected by the `SyncLock` and `End SyncLock` statements, because once the last element has been incremented, the new message must be stored before any other thread can increment the last element again.

If the `simpleMessageList` class shared one list of messages among all its instances, the variables `messagesList` and `messagesLast` would be declared as `Shared`. In this case, the variable `messagesLock` should also be `Shared`, so that there would be a single lock object used by every instance.

Code

```
Class simpleMessageList
    Public messagesList() As String = New String(50) {}
    Public messagesLast As Integer = -1
    Private messagesLock As New Object
    Public Sub addAnotherMessage(ByVal newMessage As String)
        SyncLock messagesLock
            messagesLast += 1
            If messagesLast < messagesList.Length Then
                messagesList(messagesLast) = newMessage
            End If
        End SyncLock
    End Sub
End Class
```

Description

The following example uses threads and `SyncLock`. As long as the `SyncLock` statement is present, the statement block is a critical section and `balance` never becomes a negative number. You can comment out the `SyncLock` and `End SyncLock` statements to see the effect of leaving out the `SyncLock` keyword.

Code

```

Imports System.Threading

Module Module1

    Class Account
        Dim thisLock As New Object
        Dim balance As Integer

        Dim r As New Random()

        Public Sub New(ByVal initial As Integer)
            balance = initial
        End Sub

        Public Function Withdraw(ByVal amount As Integer) As Integer
            ' This condition will never be true unless the SyncLock statement
            ' is commented out:
            If balance < 0 Then
                Throw New Exception("Negative Balance")
            End If

            ' Comment out the SyncLock and End SyncLock lines to see
            ' the effect of leaving out the SyncLock keyword.
            SyncLock thisLock
                If balance >= amount Then
                    Console.WriteLine("Balance before Withdrawal : " & balance)
                    Console.WriteLine("Amount to Withdraw : -" & amount)
                    balance = balance - amount
                    Console.WriteLine("Balance after Withdrawal : " & balance)
                    Return amount
                Else
                    ' Transaction rejected.
                    Return 0
                End If
            End SyncLock
        End Function

        Public Sub DoTransactions()
            For i As Integer = 0 To 99
                Withdraw(r.Next(1, 100))
            Next
        End Sub
    End Class

    Sub Main()
        Dim threads(10) As Thread
        Dim acc As New Account(1000)

        For i As Integer = 0 To 9
            Dim t As New Thread(New ThreadStart(AddressOf acc.DoTransactions))
            threads(i) = t
        Next

        For i As Integer = 0 To 9
            threads(i).Start()
        Next
    End Sub

End Module

```

Comments

See also

- [System.Threading.Monitor](#)

- [System.Threading.Interlocked](#)
- [Overview of synchronization primitives](#)

Then Statement

4/2/2019 • 2 minutes to read • [Edit Online](#)

Introduces a statement block to be compiled or executed if a tested condition is true.

Remarks

The `Then` keyword can be used in these contexts:

[#If...Then...#Else Directive](#)

[If...Then...Else Statement](#)

See also

- [Keywords](#)

Throw Statement (Visual Basic)

8/7/2019 • 2 minutes to read • [Edit Online](#)

Throws an exception within a procedure.

Syntax

```
Throw [ expression ]
```

Part

`expression`

Provides information about the exception to be thrown. Optional when residing in a `Catch` statement, otherwise required.

Remarks

The `Throw` statement throws an exception that you can handle with structured exception-handling code (`Try ... Catch ... Finally`) or unstructured exception-handling code (`On Error GoTo`). You can use the `Throw` statement to trap errors within your code because Visual Basic moves up the call stack until it finds the appropriate exception-handling code.

A `Throw` statement with no expression can only be used in a `Catch` statement, in which case the statement rethrows the exception currently being handled by the `Catch` statement.

The `Throw` statement resets the call stack for the `expression` exception. If `expression` is not provided, the call stack is left unchanged. You can access the call stack for the exception through the `StackTrace` property.

Example

The following code uses the `Throw` statement to throw an exception:

```
' Throws a new exception.  
Throw New System.Exception("An exception has occurred.")
```

See also

- [Try...Catch...Finally Statement](#)
- [On Error Statement](#)

Try...Catch...Finally Statement (Visual Basic)

5/15/2019 • 13 minutes to read • [Edit Online](#)

Provides a way to handle some or all possible errors that may occur in a given block of code, while still running code.

Syntax

```
Try
    [ tryStatements ]
    [ Exit Try ]
    [ Catch [ exception [ As type ] ] [ When expression ]
        [ catchStatements ]
        [ Exit Try ] ]
    [ Catch ... ]
    [ Finally
        [ finallyStatements ] ]
End Try
```

Parts

TERM	DEFINITION
<code>tryStatements</code>	Optional. Statement(s) where an error can occur. Can be a compound statement.
<code>Catch</code>	Optional. Multiple <code>Catch</code> blocks permitted. If an exception occurs when processing the <code>Try</code> block, each <code>Catch</code> statement is examined in textual order to determine whether it handles the exception, with <code>exception</code> representing the exception that has been thrown.
<code>exception</code>	Optional. Any variable name. The initial value of <code>exception</code> is the value of the thrown error. Used with <code>Catch</code> to specify the error caught. If omitted, the <code>Catch</code> statement catches any exception.
<code>type</code>	Optional. Specifies the type of class filter. If the value of <code>exception</code> is of the type specified by <code>type</code> or of a derived type, the identifier becomes bound to the exception object.
<code>When</code>	Optional. A <code>Catch</code> statement with a <code>When</code> clause catches exceptions only when <code>expression</code> evaluates to <code>True</code> . A <code>When</code> clause is applied only after checking the type of the exception, and <code>expression</code> may refer to the identifier representing the exception.

TERM	DEFINITION
<code>expression</code>	Optional. Must be implicitly convertible to <code>Boolean</code> . Any expression that describes a generic filter. Typically used to filter by error number. Used with <code>When</code> keyword to specify circumstances under which the error is caught.
<code>catchStatements</code>	Optional. Statement(s) to handle errors that occur in the associated <code>Try</code> block. Can be a compound statement.
<code>Exit Try</code>	Optional. Keyword that breaks out of the <code>Try...Catch...Finally</code> structure. Execution resumes with the code immediately following the <code>End Try</code> statement. The <code>Finally</code> statement will still be executed. Not allowed in <code>Finally</code> blocks.
<code>Finally</code>	Optional. A <code>Finally</code> block is always executed when execution leaves any part of the <code>Try...Catch</code> statement.
<code>finallyStatements</code>	Optional. Statement(s) that are executed after all other error processing has occurred.
<code>End Try</code>	Terminates the <code>Try...Catch...Finally</code> structure.

Remarks

If you expect that a particular exception might occur during a particular section of code, put the code in a `Try` block and use a `Catch` block to retain control and handle the exception if it occurs.

A `Try...Catch` statement consists of a `Try` block followed by one or more `Catch` clauses, which specify handlers for various exceptions. When an exception is thrown in a `Try` block, Visual Basic looks for the `Catch` statement that handles the exception. If a matching `Catch` statement is not found, Visual Basic examines the method that called the current method, and so on up the call stack. If no `Catch` block is found, Visual Basic displays an unhandled exception message to the user and stops execution of the program.

You can use more than one `Catch` statement in a `Try...Catch` statement. If you do this, the order of the `Catch` clauses is significant because they are examined in order. Catch the more specific exceptions before the less specific ones.

The following `Catch` statement conditions are the least specific, and will catch all exceptions that derive from the `Exception` class. You should ordinarily use one of these variations as the last `Catch` block in the `Try...Catch...Finally` structure, after catching all the specific exceptions you expect. Control flow can never reach a `Catch` block that follows either of these variations.

- The `type` is `Exception`, for example: `Catch ex As Exception`
- The statement has no `exception` variable, for example: `Catch`

When a `Try...Catch...Finally` statement is nested in another `Try` block, Visual Basic first examines each `Catch` statement in the innermost `Try` block. If no matching `Catch` statement is found, the search proceeds to the `Catch` statements of the outer `Try...Catch...Finally` block.

Local variables from a `Try` block are not available in a `Catch` block because they are separate blocks. If you want to use a variable in more than one block, declare the variable outside the `Try...Catch...Finally` structure.

TIP

The `Try...Catch...Finally` statement is available as an IntelliSense code snippet. In the Code Snippets Manager, expand **Code Patterns - If, For Each, Try Catch, Property, etc**, and then **Error Handling (Exceptions)**. For more information, see [Code Snippets](#).

Finally block

If you have one or more statements that must run before you exit the `Try` structure, use a `Finally` block. Control passes to the `Finally` block just before it passes out of the `Try...Catch` structure. This is true even if an exception occurs anywhere inside the `Try` structure.

A `Finally` block is useful for running any code that must execute even if there is an exception. Control is passed to the `Finally` block regardless of how the `Try...Catch` block exits.

The code in a `Finally` block runs even if your code encounters a `Return` statement in a `Try` or `Catch` block. Control does not pass from a `Try` or `Catch` block to the corresponding `Finally` block in the following cases:

- An [End Statement](#) is encountered in the `Try` or `Catch` block.
- A [StackOverflowException](#) is thrown in the `Try` or `Catch` block.

It is not valid to explicitly transfer execution into a `Finally` block. Transferring execution out of a `Finally` block is not valid, except through an exception.

If a `Try` statement does not contain at least one `Catch` block, it must contain a `Finally` block.

TIP

If you do not have to catch specific exceptions, the `Using` statement behaves like a `Try...Finally` block, and guarantees disposal of the resources, regardless of how you exit the block. This is true even with an unhandled exception. For more information, see [Using Statement](#).

Exception argument

The `Catch` block `exception` argument is an instance of the [Exception](#) class or a class that derives from the `Exception` class. The `Exception` class instance corresponds to the error that occurred in the `Try` block.

The properties of the `Exception` object help to identify the cause and location of an exception. For example, the `StackTrace` property lists the called methods that led to the exception, helping you find where the error occurred in the code. `Message` returns a message that describes the exception. `HelpLink` returns a link to an associated Help file. `InnerException` returns the `Exception` object that caused the current exception, or it returns `Nothing` if there is no original `Exception`.

Considerations when using a Try...Catch statement

Use a `Try...Catch` statement only to signal the occurrence of unusual or unanticipated program events.

Reasons for this include the following:

- Catching exceptions at runtime creates additional overhead, and is likely to be slower than pre-checking to avoid exceptions.
- If a `Catch` block is not handled correctly, the exception might not be reported correctly to users.

- Exception handling makes a program more complex.

You do not always need a `Try...Catch` statement to check for a condition that is likely to occur. The following example checks whether a file exists before trying to open it. This reduces the need for catching an exception thrown by the `OpenText` method.

```
Private Sub TextFileExample(ByVal filePath As String)

    ' Verify that the file exists.
    If System.IO.File.Exists(filePath) = False Then
        Console.WriteLine("File Not Found: " & filePath)
    Else
        ' Open the text file and display its contents.
        Dim sr As System.IO.StreamReader =
            System.IO.File.OpenText(filePath)

        Console.WriteLine(sr.ReadToEnd)

        sr.Close()
    End If
End Sub
```

Ensure that code in `Catch` blocks can properly report exceptions to users, whether through thread-safe logging or appropriate messages. Otherwise, exceptions might remain unknown.

Async methods

If you mark a method with the `Async` modifier, you can use the `Await` operator in the method. A statement with the `Await` operator suspends execution of the method until the awaited task completes. The task represents ongoing work. When the task that's associated with the `Await` operator finishes, execution resumes in the same method. For more information, see [Control Flow in Async Programs](#).

A task returned by an `Async` method may end in a faulted state, indicating that it completed due to an unhandled exception. A task may also end in a canceled state, which results in an `OperationCanceledException` being thrown out of the `Await` expression. To catch either type of exception, place the `Await` expression that's associated with the task in a `Try` block, and catch the exception in the `Catch` block. An example is provided later in this topic.

A task can be in a faulted state because multiple exceptions were responsible for its faulting. For example, the task might be the result of a call to `Task.WhenAll`. When you await such a task, the caught exception is only one of the exceptions, and you can't predict which exception will be caught. An example is provided later in this topic.

An `Await` expression can't be inside a `Catch` block or `Finally` block.

Iterators

An iterator function or `Get` accessor performs a custom iteration over a collection. An iterator uses a `Yield` statement to return each element of the collection one at a time. You call an iterator function by using a [For Each...Next Statement](#).

A `Yield` statement can be inside a `Try` block. A `Try` block that contains a `Yield` statement can have `Catch` blocks, and can have a `Finally` block. See the "Try Blocks in Visual Basic" section of [Iterators](#) for an example.

A `Yield` statement cannot be inside a `Catch` block or a `Finally` block.

If the `For Each` body (outside of the iterator function) throws an exception, a `Catch` block in the iterator

function is not executed, but a `Finally` block in the iterator function is executed. A `Catch` block inside an iterator function catches only exceptions that occur inside the iterator function.

Partial-trust situations

In partial-trust situations, such as an application hosted on a network share, `Try...Catch...Finally` does not catch security exceptions that occur before the method that contains the call is invoked. The following example, when you put it on a server share and run from there, produces the error "System.Security.SecurityException: Request Failed." For more information about security exceptions, see the [SecurityException](#) class.

```
Try
    Process.Start("http://www.microsoft.com")
Catch ex As Exception
    MsgBox("Can't load Web page" & vbCrLf & ex.Message)
End Try
```

In such a partial-trust situation, you have to put the `Process.Start` statement in a separate `Sub`. The initial call to the `Sub` will fail. This enables `Try...Catch` to catch it before the `Sub` that contains `Process.Start` is started and the security exception produced.

Examples

The structure of Try...Catch...Finally

The following example illustrates the structure of the `Try...Catch...Finally` statement.

```
Public Sub TryExample()
    ' Declare variables.
    Dim x As Integer = 5
    Dim y As Integer = 0

    ' Set up structured error handling.
    Try
        ' Cause a "Divide by Zero" exception.
        x = x \ y

        ' This statement does not execute because program
        ' control passes to the Catch block when the
        ' exception occurs.
        MessageBox.Show("end of Try block")
    Catch ex As Exception
        ' Show the exception's message.
        MessageBox.Show(ex.Message)

        ' Show the stack trace, which is a list of methods
        ' that are currently executing.
        MessageBox.Show("Stack Trace: " & vbCrLf & ex.StackTrace)
    Finally
        ' This line executes whether or not the exception occurs.
        MessageBox.Show("in Finally block")
    End Try
End Sub
```

Exception in a method called from a Try block

In the following example, the `CreateException` method throws a `NullReferenceException`. The code that generates the exception is not in a `Try` block. Therefore, the `CreateException` method does not handle the exception. The `RunSample` method does handle the exception because the call to the `CreateException` method is in a `Try` block.

The example includes `Catch` statements for several types of exceptions, ordered from the most specific to the most general.

```
Public Sub RunSample()
    Try
        CreateException()
    Catch ex As System.IO.IOException
        ' Code that reacts to IOException.
    Catch ex As NullReferenceException
        MessageBox.Show("NullReferenceException: " & ex.Message)
        MessageBox.Show("Stack Trace: " & vbCrLf & ex.StackTrace)
    Catch ex As Exception
        ' Code that reacts to any other exception.
    End Try
End Sub

Private Sub CreateException()
    ' This code throws a NullReferenceException.
    Dim obj = Nothing
    Dim prop = obj.Name

    ' This code also throws a NullReferenceException.
    Throw New NullReferenceException("Something happened.")
End Sub
```

The Catch When statement

The following example shows how to use a `Catch When` statement to filter on a conditional expression. If the conditional expression evaluates to `True`, the code in the `catch` block runs.

```
Private Sub WhenExample()
    Dim i As Integer = 5

    Try
        Throw New ArgumentException()
    Catch e As OverflowException When i = 5
        Console.WriteLine("First handler")
    Catch e As ArgumentException When i = 4
        Console.WriteLine("Second handler")
    Catch When i = 5
        Console.WriteLine("Third handler")
    End Try
End Sub
' Output: Third handler
```

Nested Try statements

The following example has a `Try...Catch` statement that is contained in a `Try` block. The inner `Catch` block throws an exception that has its `InnerException` property set to the original exception. The outer `Catch` block reports its own exception and the inner exception.

```

Private Sub InnerExceptionExample()
    Try
        Try
            ' Set a reference to a StringBuilder.
            ' The exception below does not occur if the commented
            ' out statement is used instead.
            Dim sb As System.Text.StringBuilder
            'Dim sb As New System.Text.StringBuilder

            ' Cause a NullReferenceException.
            sb.Append("text")
        Catch ex As Exception
            ' Throw a new exception that has the inner exception
            ' set to the original exception.
            Throw New ApplicationException("Something happened :(", ex)
        End Try
    Catch ex2 As Exception
        ' Show the exception.
        Console.WriteLine("Exception: " & ex2.Message)
        Console.WriteLine(ex2.StackTrace)

        ' Show the inner exception, if one is present.
        If ex2.InnerException IsNot Nothing Then
            Console.WriteLine("Inner Exception: " & ex2.InnerException.Message)
            Console.WriteLine(ex2.StackTrace)
        End If
    End Try
End Sub

```

Exception handling for async methods

The following example illustrates exception handling for async methods. To catch an exception that applies to an async task, the `Await` expression is in a `Try` block of the caller, and the exception is caught in the `Catch` block.

Uncomment the `Throw New Exception` line in the example to demonstrate exception handling. The exception is caught in the `Catch` block, the task's `IsFaulted` property is set to `True`, and the task's `Exception.InnerException` property is set to the exception.

Uncomment the `Throw New OperationCancelledException` line to demonstrate what happens when you cancel an asynchronous process. The exception is caught in the `Catch` block, and the task's `IsCanceled` property is set to `True`. However, under some conditions that don't apply to this example, `IsFaulted` is set to `True` and `IsCanceled` is set to `False`.

```

Public Async Function DoSomethingAsync() As Task
    Dim theTask As Task(Of String) = DelayAsync()

    Try
        Dim result As String = Await theTask
        Debug.WriteLine("Result: " & result)
    Catch ex As Exception
        Debug.WriteLine("Exception Message: " & ex.Message)
    End Try

    Debug.WriteLine("Task IsCanceled: " & theTask.IsCanceled)
    Debug.WriteLine("Task IsFaulted: " & theTask.IsFaulted)
    If theTask.Exception IsNot Nothing Then
        Debug.WriteLine("Task Exception Message: " &
            theTask.Exception.Message)
        Debug.WriteLine("Task Inner Exception Message: " &
            theTask.Exception.InnerException.Message)
    End If
End Function

Private Async Function DelayAsync() As Task(Of String)
    Await Task.Delay(100)

    ' Uncomment each of the following lines to
    ' demonstrate exception handling.

    ' Throw New OperationCanceledException("canceled")
    ' Throw New Exception("Something happened.")
    Return "Done"
End Function

' Output when no exception is thrown in the awaited method:
'   Result: Done
'   Task IsCanceled: False
'   Task IsFaulted:  False

' Output when an Exception is thrown in the awaited method:
'   Exception Message: Something happened.
'   Task IsCanceled: False
'   Task IsFaulted:  True
'   Task Exception Message: One or more errors occurred.
'   Task Inner Exception Message: Something happened.

' Output when an OperationCanceledException or TaskCanceledException
' is thrown in the awaited method:
'   Exception Message: canceled
'   Task IsCanceled: True
'   Task IsFaulted:  False

```

Handling multiple exceptions in async methods

The following example illustrates exception handling where multiple tasks can result in multiple exceptions. The `Try` block has the `Await` expression for the task that `Task.WhenAll` returned. The task is complete when the three tasks to which `Task.WhenAll` is applied are complete.

Each of the three tasks causes an exception. The `catch` block iterates through the exceptions, which are found in the `Exception.InnerExceptions` property of the task that `Task.WhenAll` returned.

```

Public Async Function DoMultipleAsync() As Task
    Dim theTask1 As Task = ExcAsync(info:="First Task")
    Dim theTask2 As Task = ExcAsync(info:="Second Task")
    Dim theTask3 As Task = ExcAsync(info:="Third Task")

    Dim allTasks As Task = Task.WhenAll(theTask1, theTask2, theTask3)

    Try
        Await allTasks
    Catch ex As Exception
        Debug.WriteLine("Exception: " & ex.Message)
        Debug.WriteLine("Task IsFaulted: " & allTasks.IsFaulted)
        For Each inEx In allTasks.Exception.InnerExceptions
            Debug.WriteLine("Task Inner Exception: " + inEx.Message)
        Next
    End Try
End Function

Private Async Function ExcAsync(info As String) As Task
    Await Task.Delay(100)

    Throw New Exception("Error-" & info)
End Function

' Output:
'   Exception: Error-First Task
'   Task IsFaulted: True
'   Task Inner Exception: Error-First Task
'   Task Inner Exception: Error-Second Task
'   Task Inner Exception: Error-Third Task

```

See also

- [Err](#)
- [Exception](#)
- [Exit Statement](#)
- [On Error Statement](#)
- [Best Practices for Using Code Snippets](#)
- [Exception Handling](#)
- [Throw Statement](#)

Using Statement (Visual Basic)

9/28/2019 • 4 minutes to read • [Edit Online](#)

Declares the beginning of a `Using` block and optionally acquires the system resources that the block controls.

Syntax

```
Using { resourcelist | resourceexpression }
    [ statements ]
End Using
```

Parts

TERM	DEFINITION
<code>resourcelist</code>	Required if you do not supply <code>resourceexpression</code> . List of one or more system resources that this <code>Using</code> block controls, separated by commas.
<code>resourceexpression</code>	Required if you do not supply <code>resourcelist</code> . Reference variable or expression referring to a system resource to be controlled by this <code>Using</code> block.
<code>statements</code>	Optional. Block of statements that the <code>using</code> block runs.
<code>End Using</code>	Required. Terminates the definition of the <code>Using</code> block and disposes of all the resources that it controls.

Each resource in the `resourcelist` part has the following syntax and parts:

```
resourcename As New resourcetype [ ( [ arglist ] ) ]
```

-or-

```
resourcename As resourcetype = resourceexpression
```

resourcelist Parts

TERM	DEFINITION
<code>resourcename</code>	Required. Reference variable that refers to a system resource that the <code>Using</code> block controls.
<code>New</code>	Required if the <code>Using</code> statement acquires the resource. If you have already acquired the resource, use the second syntax alternative.
<code>resourcetype</code>	Required. The class of the resource. The class must implement the <code>IDisposable</code> interface.

TERM	DEFINITION
<code>arglist</code>	Optional. List of arguments you are passing to the constructor to create an instance of <code>resourcetype</code> . See Parameter List .
<code>resourceexpression</code>	Required. Variable or expression referring to a system resource satisfying the requirements of <code>resourcetype</code> . If you use the second syntax alternative, you must acquire the resource before passing control to the <code>Using</code> statement.

Remarks

Sometimes your code requires an unmanaged resource, such as a file handle, a COM wrapper, or a SQL connection. A `Using` block guarantees the disposal of one or more such resources when your code is finished with them. This makes them available for other code to use.

Managed resources are disposed of by the .NET Framework garbage collector (GC) without any extra coding on your part. You do not need a `Using` block for managed resources. However, you can still use a `Using` block to force the disposal of a managed resource instead of waiting for the garbage collector.

A `Using` block has three parts: acquisition, usage, and disposal.

- *Acquisition* means creating a variable and initializing it to refer to the system resource. The `Using` statement can acquire one or more resources, or you can acquire exactly one resource before entering the block and supply it to the `Using` statement. If you supply `resourceexpression`, you must acquire the resource before passing control to the `Using` statement.
- *Usage* means accessing the resources and performing actions with them. The statements between `Using` and `End Using` represent the usage of the resources.
- *Disposal* means calling the `Dispose` method on the object in `resourcename`. This allows the object to cleanly terminate its resources. The `End Using` statement disposes of the resources under the `Using` block's control.

Behavior

A `Using` block behaves like a `Try ... Finally` construction in which the `Try` block uses the resources and the `Finally` block disposes of them. Because of this, the `Using` block guarantees disposal of the resources, no matter how you exit the block. This is true even in the case of an unhandled exception, except for a [StackOverflowException](#).

The scope of every resource variable acquired by the `Using` statement is limited to the `Using` block.

If you specify more than one system resource in the `Using` statement, the effect is the same as if you nested `Using` blocks one within another.

If `resourcename` is `Nothing`, no call to `Dispose` is made, and no exception is thrown.

Structured Exception Handling Within a Using Block

If you need to handle an exception that might occur within the `Using` block, you can add a complete `Try ... Finally` construction to it. If you need to handle the case where the `Using` statement is not successful in acquiring a resource, you can test to see if `resourcename` is `Nothing`.

Structured Exception Handling Instead of a Using Block

If you need finer control over the acquisition of the resources, or you need additional code in the `Finally` block, you can rewrite the `Using` block as a `Try ... Finally` construction. The following example shows skeleton `Try` and `using` constructions that are equivalent in the acquisition and disposal of `resource`.

```
Using resource As New resourceType
    ' Insert code to work with resource.
End Using

' For the acquisition and disposal of resource, the following
' Try construction is equivalent to the Using block.
Dim resource As New resourceType
Try
    ' Insert code to work with resource.
Finally
    If resource IsNot Nothing Then
        resource.Dispose()
    End If
End Try
```

NOTE

The code inside the `Using` block should not assign the object in `resourcename` to another variable. When you exit the `Using` block, the resource is disposed, and the other variable cannot access the resource to which it points.

Example

The following example creates a file that is named log.txt and writes two lines of text to the file. The example also reads that same file and displays the lines of text:

Because the `TextWriter` and `TextReader` classes implement the `IDisposable` interface, the code can use `Using` statements to ensure that the file is correctly closed after the write and read operations.

```
Private Sub WriteFile()
    Using writer As System.IO.TextWriter = System.IO.File.CreateText("log.txt")
        writer.WriteLine("This is line one.")
        writer.WriteLine("This is line two.")
    End Using
End Sub

Private Sub ReadFile()
    Using reader As System.IO.TextReader = System.IO.File.OpenText("log.txt")
        Dim line As String

        line = reader.ReadLine()
        Do Until line Is Nothing
            Console.WriteLine(line)
            line = reader.ReadLine()
        Loop
    End Using
End Sub
```

See also

- [IDisposable](#)
- [Try...Catch...Finally Statement](#)

- How to: Dispose of a System Resource

While...End While Statement (Visual Basic)

10/18/2019 • 3 minutes to read • [Edit Online](#)

Runs a series of statements as long as a given condition is `True`.

Syntax

```
While condition
    [ statements ]
    [ Continue While ]
    [ statements ]
    [ Exit While ]
    [ statements ]
End While
```

Parts

TERM	DEFINITION
<code>condition</code>	Required. <code>Boolean</code> expression. If <code>condition</code> is <code>Nothing</code> , Visual Basic treats it as <code>False</code> .
<code>statements</code>	Optional. One or more statements following <code>While</code> , which run every time <code>condition</code> is <code>True</code> .
<code>Continue While</code>	Optional. Transfers control to the next iteration of the <code>While</code> block.
<code>Exit While</code>	Optional. Transfers control out of the <code>While</code> block.
<code>End While</code>	Required. Terminates the definition of the <code>While</code> block.

Remarks

Use a `While...End While` structure when you want to repeat a set of statements an indefinite number of times, as long as a condition remains `True`. If you want more flexibility with where you test the condition or what result you test it for, you might prefer the [Do...Loop Statement](#). If you want to repeat the statements a set number of times, the [For...Next Statement](#) is usually a better choice.

NOTE

The `While` keyword is also used in the [Do...Loop Statement](#), the [Skip While Clause](#) and the [Take While Clause](#).

If `condition` is `True`, all of the `statements` run until the `End While` statement is encountered. Control then returns to the `While` statement, and `condition` is again checked. If `condition` is still `True`, the process is repeated. If it's `False`, control passes to the statement that follows the `End While` statement.

The `While` statement always checks the condition before it starts the loop. Looping continues while the

condition remains `True`. If `condition` is `False` when you first enter the loop, it doesn't run even once.

The `condition` usually results from a comparison of two values, but it can be any expression that evaluates to a [Boolean Data Type](#) value (`True` or `False`). This expression can include a value of another data type, such as a numeric type, that has been converted to `Boolean`.

You can nest `While` loops by placing one loop within another. You can also nest different kinds of control structures within one another. For more information, see [Nested Control Structures](#).

Exit While

The [Exit While](#) statement can provide another way to exit a `While` loop. `Exit While` immediately transfers control to the statement that follows the `End While` statement.

You typically use `Exit While` after some condition is evaluated (for example, in an `If...Then...Else` structure). You might want to exit a loop if you detect a condition that makes it unnecessary or impossible to continue iterating, such as an erroneous value or a termination request. You can use `Exit While` when you test for a condition that could cause an *endless loop*, which is a loop that could run an extremely large or even infinite number of times. You can then use `Exit While` to escape the loop.

You can place any number of `Exit While` statements anywhere in the `While` loop.

When used within nested `While` loops, `Exit While` transfers control out of the innermost loop and into the next higher level of nesting.

The `Continue While` statement immediately transfers control to the next iteration of the loop. For more information, see [Continue Statement](#).

Example

In the following example, the statements in the loop continue to run until the `index` variable is greater than 10.

```
Dim index As Integer = 0
While index <= 10
    Debug.WriteLine(index.ToString & " ")
    index += 1
End While

Debug.WriteLine("")
' Output: 0 1 2 3 4 5 6 7 8 9 10
```

Example

The following example illustrates the use of the `Continue While` and `Exit While` statements.

```

Dim index As Integer = 0
While index < 100000
    index += 1

    ' If index is between 5 and 7, continue
    ' with the next iteration.
    If index >= 5 And index <= 8 Then
        Continue While
    End If

    ' Display the index.
    Debug.WriteLine(index.ToString & " ")

    ' If index is 10, exit the loop.
    If index = 10 Then
        Exit While
    End If
End While

Debug.WriteLine("")
' Output: 1 2 3 4 9 10

```

Example

The following example reads all lines in a text file. The [OpenText](#) method opens the file and returns a [StreamReader](#) that reads the characters. In the `While` condition, the [Peek](#) method of the [StreamReader](#) determines whether the file contains additional characters.

```

Private Sub ShowText(ByVal textFilePath As String)
    If System.IO.File.Exists(textFilePath) = False Then
        Debug.WriteLine("File Not Found: " & textFilePath)
    Else
        Dim sr As System.IO.StreamReader = System.IO.File.OpenText(textFilePath)

        While sr.Peek() >= 0
            Debug.WriteLine(sr.ReadLine())
        End While

        sr.Close()
    End If
End Sub

```

See also

- [Loop Structures](#)
- [Do...Loop Statement](#)
- [For...Next Statement](#)
- [Boolean Data Type](#)
- [Nested Control Structures](#)
- [Exit Statement](#)
- [Continue Statement](#)

With...End With Statement (Visual Basic)

10/18/2019 • 3 minutes to read • [Edit Online](#)

Executes a series of statements that repeatedly refer to a single object or structure so that the statements can use a simplified syntax when accessing members of the object or structure. When using a structure, you can only read the values of members or invoke methods, and you get an error if you try to assign values to members of a structure used in a `With...End With` statement.

Syntax

```
With objectExpression  
    [ statements ]  
End With
```

Parts

TERM	DEFINITION
<code>objectExpression</code>	Required. An expression that evaluates to an object. The expression may be arbitrarily complex and is evaluated only once. The expression can evaluate to any data type, including elementary types.
<code>statements</code>	Optional. One or more statements between <code>With</code> and <code>End With</code> that may refer to members of an object that's produced by the evaluation of <code>objectExpression</code> .
<code>End With</code>	Required. Terminates the definition of the <code>With</code> block.

Remarks

By using `With...End With`, you can perform a series of statements on a specified object without specifying the name of the object multiple times. Within a `With` statement block, you can specify a member of the object starting with a period, as if the `With` statement object preceded it.

For example, to change multiple properties on a single object, place the property assignment statements inside the `With...End With` block, referring to the object only once instead of once for each property assignment.

If your code accesses the same object in multiple statements, you gain the following benefits by using the `With` statement:

- You don't need to evaluate a complex expression multiple times or assign the result to a temporary variable to refer to its members multiple times.
- You make your code more readable by eliminating repetitive qualifying expressions.

The data type of `objectExpression` can be any class or structure type or even a Visual Basic elementary type such as `Integer`. If `objectExpression` results in anything other than an object, you can only read the values of its members or invoke methods, and you get an error if you try to assign values to members of a structure used in a `With...End With` statement. This is the same error you would get if you invoked a method that returned a

structure and immediately accessed and assigned a value to a member of the function's result, such as `GetAPoint().x = 1`. The problem in both cases is that the structure exists only on the call stack, and there is no way a modified structure member in these situations can write to a location such that any other code in the program can observe the change.

The `objectExpression` is evaluated once, upon entry into the block. You can't reassign the `objectExpression` from within the `With` block.

Within a `With` block, you can access the methods and properties of only the specified object without qualifying them. You can use methods and properties of other objects, but you must qualify them with their object names.

You can place one `With...End With` statement within another. Nested `With...End With` statements may be confusing if the objects that are being referred to aren't clear from context. You must provide a fully qualified reference to an object that's in an outer `With` block when the object is referenced from within an inner `With` block.

You can't branch into a `With` statement block from outside the block.

Unless the block contains a loop, the statements run only once. You can nest different kinds of control structures. For more information, see [Nested Control Structures](#).

NOTE

You can use the `With` keyword in object initializers also. For more information and examples, see [Object Initializers: Named and Anonymous Types](#) and [Anonymous Types](#).

If you're using a `With` block only to initialize the properties or fields of an object that you've just instantiated, consider using an object initializer instead.

Example

In the following example, each `With` block executes a series of statements on a single object.

```
Private Sub AddCustomer()
    Dim theCustomer As New Customer

    With theCustomer
        .Name = "Coho Vineyard"
        .URL = "http://www.cohovineyard.com/"
        .City = "Redmond"
    End With

    With theCustomer.Comments
        .Add("First comment.")
        .Add("Second comment.")
    End With
End Sub

Public Class Customer
    Public Property Name As String
    Public Property City As String
    Public Property URL As String

    Public Property Comments As New List(Of String)
End Class
```

Example

The following example nests `With...End With` statements. Within the nested `With` statement, the syntax refers to the inner object.

```
Dim theWindow As New EntryWindow

With theWindow
    With .InfoLabel
        .Content = "This is a message."
        .Foreground = Brushes.DarkSeaGreen
        .Background = Brushes.LightYellow
    End With

    .Title = "The Form Title"
    .Show()
End With
```

See also

- [List<T>](#)
- [Nested Control Structures](#)
- [Object Initializers: Named and Anonymous Types](#)
- [Anonymous Types](#)

Yield Statement (Visual Basic)

10/18/2019 • 4 minutes to read • [Edit Online](#)

Sends the next element of a collection to a `For Each...Next` statement.

Syntax

```
Yield expression
```

Parameters

TERM	DEFINITION
<code>expression</code>	Required. An expression that is implicitly convertible to the type of the iterator function or <code>Get</code> accessor that contains the <code>Yield</code> statement.

Remarks

The `Yield` statement returns one element of a collection at a time. The `Yield` statement is included in an iterator function or `Get` accessor, which perform custom iterations over a collection.

You consume an iterator function by using a [For Each...Next Statement](#) or a LINQ query. Each iteration of the `For Each` loop calls the iterator function. When a `Yield` statement is reached in the iterator function, `expression` is returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator function is called.

An implicit conversion must exist from the type of `expression` in the `Yield` statement to the return type of the iterator.

You can use an `Exit Function` or `Return` statement to end the iteration.

"Yield" is not a reserved word and has special meaning only when it is used in an `Iterator` function or `Get` accessor.

For more information about iterator functions and `Get` accessors, see [Iterators](#).

Iterator Functions and Get Accessors

The declaration of an iterator function or `Get` accessor must meet the following requirements:

- It must include an `Iterator` modifier.
- The return type must be `IEnumerable`, `IEnumerable<T>`, `IEnumerator`, or `IEnumerator<T>`.
- It cannot have any `ByRef` parameters.

An iterator function cannot occur in an event, instance constructor, static constructor, or static destructor.

An iterator function can be an anonymous function. For more information, see [Iterators](#).

Exception Handling

A `Yield` statement can be inside a `Try` block of a [Try...Catch...Finally Statement](#). A `Try` block that has a `Yield` statement can have `Catch` blocks, and can have a `Finally` block.

A `Yield` statement cannot be inside a `Catch` block or a `Finally` block.

If the `For Each` body (outside of the iterator function) throws an exception, a `Catch` block in the iterator function is not executed, but a `Finally` block in the iterator function is executed. A `catch` block inside an iterator function catches only exceptions that occur inside the iterator function.

Technical Implementation

The following code returns an `IEnumerable (Of String)` from an iterator function and then iterates through the elements of the `IEnumerable (Of String)`.

```
Dim elements As IEnumerable(Of String) = MyIteratorFunction()  
...  
For Each element As String In elements  
Next
```

The call to `MyIteratorFunction` doesn't execute the body of the function. Instead the call returns an `IEnumerable(Of String)` into the `elements` variable.

On an iteration of the `For Each` loop, the `MoveNext` method is called for `elements`. This call executes the body of `MyIteratorFunction` until the next `Yield` statement is reached. The `Yield` statement returns an expression that determines not only the value of the `element` variable for consumption by the loop body but also the `Current` property of `elements`, which is an `IEnumerable (Of String)`.

On each subsequent iteration of the `For Each` loop, the execution of the iterator body continues from where it left off, again stopping when it reaches a `Yield` statement. The `For Each` loop completes when the end of the iterator function or a `Return` or `Exit Function` statement is reached.

Example

The following example has a `Yield` statement that is inside a `For...Next` loop. Each iteration of the `For Each` statement body in `Main` creates a call to the `Power` iterator function. Each call to the iterator function proceeds to the next execution of the `Yield` statement, which occurs during the next iteration of the `For...Next` loop.

The return type of the iterator method is `IEnumerable<T>`, an iterator interface type. When the iterator method is called, it returns an enumerable object that contains the powers of a number.

```

Sub Main()
    For Each number In Power(2, 8)
        Console.WriteLine(number & " ")
    Next
    ' Output: 2 4 8 16 32 64 128 256
    Console.ReadKey()
End Sub

Private Iterator Function Power(
    ByVal base As Integer, ByVal highExponent As Integer) _
As System.Collections.Generic.IEnumerable(Of Integer)

    Dim result = 1

    For counter = 1 To highExponent
        result = result * base
        Yield result
    Next
End Function

```

Example

The following example demonstrates a `Get` accessor that is an iterator. The property declaration includes an `Iterator` modifier.

```

Sub Main()
    Dim theGalaxies As New Galaxies
    For Each theGalaxy In theGalaxies.NextGalaxy
        With theGalaxy
            Console.WriteLine(.Name & " " & .MegaLightYears)
        End With
    Next
    Console.ReadKey()
End Sub

Public Class Galaxies
    Public ReadOnly Iterator Property NextGalaxy _
    As System.Collections.Generic.IEnumerable(Of Galaxy)
        Get
            Yield New Galaxy With {.Name = "Tadpole", .MegaLightYears = 400}
            Yield New Galaxy With {.Name = "Pinwheel", .MegaLightYears = 25}
            Yield New Galaxy With {.Name = "Milky Way", .MegaLightYears = 0}
            Yield New Galaxy With {.Name = "Andromeda", .MegaLightYears = 3}
        End Get
    End Property
End Class

Public Class Galaxy
    Public Property Name As String
    Public Property MegaLightYears As Integer
End Class

```

For additional examples, see [Iterators](#).

See also

- [Statements](#)

Clauses (Visual Basic)

5/4/2018 • 2 minutes to read • [Edit Online](#)

The topics in this section document Visual Basic run-time clauses.

In This Section

[Alias](#)

[As](#)

[Handles](#)

[Implements](#)

[In](#)

[Into](#)

[Of](#)

Related Sections

[Visual Basic Language Reference](#)

[Visual Basic](#)

Alias Clause (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Indicates that an external procedure has another name in its DLL.

Remarks

The `Alias` keyword can be used in this context:

Declare Statement

In the following example, the `Alias` keyword is used to provide the name of the function in advapi32.dll, `GetUserNameA`, that `getUserName` is used in place of in this example. Function `getUserName` is called in sub `getUser`, which displays the name of the current user.

```
Declare Function GetUserName Lib "advapi32.dll" Alias "GetUserNameA" (  
    ByVal lpBuffer As String, ByRef nSize As Integer) As Integer  
Sub GetUser()  
    Dim buffer As String = New String(CChar(" "), 25)  
    Dim retVal As Integer = GetUserName(buffer, 25)  
    Dim userName As String = Strings.Left(buffer, InStr(buffer, Chr(0)) - 1)  
    MsgBox(userName)  
End Sub
```

See also

- [Keywords](#)

As Clause (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Introduces an `As` clause, which identifies a data type in a declaration statement or a constraint list on a generic type parameter.

Remarks

The `As` keyword can be used in these contexts:

[Aggregate Clause](#)

[Class Statement](#)

[Const Statement](#)

[Declare Statement](#)

[Delegate Statement](#)

[Dim Statement](#)

[Enum Statement](#)

[Event Statement](#)

[For...Next Statements](#)

[For Each...Next Statements](#)

[From Clause](#)

[Function Statement](#)

[Group Join Clause](#)

[Interface Statement](#)

[Operator Statement](#)

[Property Statement](#)

[Structure Statement](#)

[Sub Statement](#)

[Try...Catch...Finally Statements](#)

See also

- [How to: Create a New Variable](#)
- [Data Types](#)
- [Variable Declaration](#)
- [Type List](#)
- [Generic Types in Visual Basic](#)
- [Keywords](#)

Handles Clause (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Declares that a procedure handles a specified event.

Syntax

```
proceduredeclaration Handles eventlist
```

Parts

`proceduredeclaration`

The `Sub` procedure declaration for the procedure that will handle the event.

`eventlist`

List of the events for `proceduredeclaration` to handle, separated by commas. The events must be raised by either the base class for the current class, or by an object declared using the `WithEvents` keyword.

Remarks

Use the `Handles` keyword at the end of a procedure declaration to cause it to handle events raised by an object variable declared using the `WithEvents` keyword. The `Handles` keyword can also be used in a derived class to handle events from a base class.

The `Handles` keyword and the `AddHandler` statement both allow you to specify that particular procedures handle particular events, but there are differences. Use the `Handles` keyword when defining a procedure to specify that it handles a particular event. The `AddHandler` statement connects procedures to events at run time. For more information, see [AddHandler Statement](#).

For custom events, the application invokes the event's `AddHandler` accessor when it adds the procedure as an event handler. For more information on custom events, see [Event Statement](#).

Example

```

Public Class ContainerClass
    ' Module or class level declaration.
    WithEvents Obj As New Class1

    Public Class Class1
        ' Declare an event.
        Public Event Ev_Event()
        Sub CauseSomeEvent()
            ' Raise an event.
            RaiseEvent Ev_Event()
        End Sub
    End Class

    Sub EventHandler() Handles Obj.Ev_Event
        ' Handle the event.
        MsgBox("EventHandler caught event.")
    End Sub

    ' Call the TestEvents procedure from an instance of the ContainerClass
    ' class to test the Ev_Event event and the event handler.
    Public Sub TestEvents()
        Obj.CauseSomeEvent()
    End Sub
End Class

```

The following example demonstrates how a derived class can use the `Handles` statement to handle an event from a base class.

```

Public Class BaseClass
    ' Declare an event.
    Event Ev1()
End Class
Class DerivedClass
    Inherits BaseClass
    Sub TestEvents() Handles MyBase.Ev1
        ' Add code to handle this event.
    End Sub
End Class

```

Example

The following example contains two button event handlers for a **WPF Application** project.

```

Private Sub Button1_Click(sender As System.Object, e As System.Windows.RoutedEventArgs) Handles
    Button1.Click
    MessageBox.Show(sender.Name & " clicked")
End Sub

Private Sub Button2_Click(sender As System.Object, e As System.Windows.RoutedEventArgs) Handles
    Button2.Click
    MessageBox.Show(sender.Name & " clicked")
End Sub

```

Example

The following example is equivalent to the previous example. The `eventlist` in the `Handles` clause contains the events for both buttons.

```
Private Sub Button_Click(sender As System.Object, e As System.Windows.RoutedEventArgs) Handles  
Button1.Click, Button2.Click  
    MessageBox.Show(sender.Name & " clicked")  
End Sub
```

See also

- [WithEvents](#)
- [AddHandler Statement](#)
- [RemoveHandler Statement](#)
- [Event Statement](#)
- [RaiseEvent Statement](#)
- [Events](#)

Implements Clause (Visual Basic)

9/12/2019 • 2 minutes to read • [Edit Online](#)

Indicates that a class or structure member is providing the implementation for a member defined in an interface.

Remarks

The `Implements` keyword is not the same as the [Implements Statement](#). You use the `Implements` statement to specify that a class or structure implements one or more interfaces, and then for each member you use the `Implements` keyword to specify which interface and which member it implements.

If a class or structure implements an interface, it must include the `Implements` statement immediately after the [Class Statement](#) or [Structure Statement](#), and it must implement all the members defined by the interface.

Reimplementation

In a derived class, you can reimplement an interface member that the base class has already implemented. This is different from overriding the base class member in the following respects:

- The base class member does not need to be [Overridable](#) to be reimplemented.
- You can reimplement the member with a different name.

The `Implements` keyword can be used in the following contexts:

- [Event Statement](#)
- [Function Statement](#)
- [Property Statement](#)
- [Sub Statement](#)

See also

- [Implements Statement](#)
- [Interface Statement](#)
- [Class Statement](#)
- [Structure Statement](#)

In Clause (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Specifies the group that the loop variable is to traverse in a `For Each` loop, or specifies the collection to query in a `From`, `Join`, or `Group Join` clause.

Remarks

The `In` keyword can be used in the following contexts:

[For Each...Next Statement](#)

[From Clause](#)

[Join Clause](#)

[Group Join Clause](#)

See also

- [Keywords](#)

Into Clause (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Identifies aggregate functions or groupings to apply to a collection.

Remarks

The `Each` keyword is used in the following contexts:

[Aggregate Clause](#)

[Group By Clause](#)

[Group Join Clause](#)

See also

- [Keywords](#)

Of Clause (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Introduces an `of` clause, which identifies a *type parameter* on a *generic class*, structure, interface, delegate, or procedure. For information on generic types, see [Generic Types in Visual Basic](#).

Using the Of Keyword

The following code example uses the `of` keyword to define the outline of a class that takes two type parameters. It *constraints* the `keyType` parameter by the [IComparable](#) interface, which means the consuming code must supply a type argument that implements [IComparable](#). This is necessary so that the `add` procedure can call the [IComparable.CompareTo](#) method. For more information on constraints, see [Type List](#).

```
Public Class Dictionary(Of entryType, keyType As IComparable)
    Public Sub add(ByVal e As entryType, ByVal k As keyType)
        Dim dk As keyType
        If k.CompareTo(dk) = 0 Then
            End If
        End Sub
    Public Function find(ByVal k As keyType) As entryType
    End Function
End Class
```

If you complete the preceding class definition, you can construct a variety of `dictionary` classes from it. The types you supply to `entryType` and `keyType` determine what type of entry the class holds and what type of key it associates with each entry. Because of the constraint, you must supply to `keyType` a type that implements [IComparable](#).

The following code example creates an object that holds `String` entries and associates an `Integer` key with each one. `Integer` implements [IComparable](#) and therefore satisfies the constraint on `keyType`.

```
Dim d As New dictionary(Of String, Integer)
```

The `of` keyword can be used in these contexts:

[Class Statement](#)

[Delegate Statement](#)

[Function Statement](#)

[Interface Statement](#)

[Structure Statement](#)

[Sub Statement](#)

See also

- [IComparable](#)
- [Type List](#)
- [Generic Types in Visual Basic](#)

- [In](#)
- [Out](#)

Declaration Contexts and Default Access Levels (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

This topic describes which Visual Basic types can be declared within which other types, and what their access levels default to if not specified.

Declaration Context Levels

The *declaration context* of a programming element is the region of code in which it is declared. This is often another programming element, which is then called the *containing element*.

The levels for declaration contexts are the following:

- *Namespace level* — within a source file or namespace but not within a class, structure, module, or interface
- *Module level* — within a class, structure, module, or interface but not within a procedure or block
- *Procedure level* — within a procedure or block (such as `If` or `For`)

The following table shows the default access levels for various declared programming elements, depending on their declaration contexts.

DECLARED ELEMENT	NAMESPACE LEVEL	MODULE LEVEL	PROCEDURE LEVEL
Variable (Dim Statement)	Not allowed	<code>Private</code> (<code>Public</code> in <code>Structure</code> , not allowed in <code>Interface</code>)	<code>Public</code>
Constant (Const Statement)	Not allowed	<code>Private</code> (<code>Public</code> in <code>Structure</code> , not allowed in <code>Interface</code>)	<code>Public</code>
Enumeration (Enum Statement)	<code>Friend</code>	<code>Public</code>	Not allowed
Class (Class Statement)	<code>Friend</code>	<code>Public</code>	Not allowed
Structure (Structure Statement)	<code>Friend</code>	<code>Public</code>	Not allowed
Module (Module Statement)	<code>Friend</code>	Not allowed	Not allowed
Interface (Interface Statement)	<code>Friend</code>	<code>Public</code>	Not allowed
Procedure (Function Statement , Sub Statement)	Not allowed	<code>Public</code>	Not allowed

DECLARED ELEMENT	NAMESPACE LEVEL	MODULE LEVEL	PROCEDURE LEVEL
External reference (Declare Statement)	Not allowed	<code>Public</code> (not allowed in <code>Interface</code>)	Not allowed
Operator (Operator Statement)	Not allowed	<code>Public</code> (not allowed in <code>Interface</code> or <code>Module</code>)	Not allowed
Property (Property Statement)	Not allowed	<code>Public</code>	Not allowed
Default property (Default)	Not allowed	<code>Public</code> (not allowed in <code>Module</code>)	Not allowed
Event (Event Statement)	Not allowed	<code>Public</code>	Not allowed
Delegate (Delegate Statement)	<code>Friend</code>	<code>Public</code>	Not allowed

For more information, see [Access levels in Visual Basic](#).

See also

- [Friend](#)
- [Private](#)
- [Public](#)

Attribute List (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies the attributes to be applied to a declared programming element. Multiple attributes are separated by commas. Following is the syntax for one attribute.

Syntax

```
[ attributemodifier ] attributename [ ( attributearguments | attributeinitializer ) ]
```

Parts

<code>attributemodifier</code>	Required for attributes applied at the beginning of a source file. Can be Assembly or Module .
<code>attributename</code>	Required. Name of the attribute.
<code>attributearguments</code>	Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.
<code>attributeinitializer</code>	Optional. List of variable or property initializers for this attribute. Multiple initializers are separated by commas.

Remarks

You can apply one or more attributes to nearly any programming element (types, procedures, properties, and so forth). Attributes appear in your assembly's metadata, and they can help you annotate your code or specify how to use a particular programming element. You can apply attributes defined by Visual Basic and the .NET Framework, and you can define your own attributes.

For more information on when to use attributes, see [Attributes overview](#). For information on attribute names, see [Declared Element Names](#).

Rules

- **Placement.** You can apply attributes to most declared programming elements. To apply one or more attributes, you place an *attribute block* at the beginning of the element declaration. Each entry in the attribute list specifies an attribute you wish to apply, and the modifier and arguments you are using for this invocation of the attribute.
- **Angle Brackets.** If you supply an attribute list, you must enclose it in angle brackets ("<" and ">").
- **Part of the Declaration.** The attribute must be part of the element declaration, not a separate statement. You can use the line-continuation sequence ("`_`") to extend the declaration statement onto multiple source-code lines.
- **Modifiers.** An attribute modifier (`Assembly` or `Module`) is required on every attribute applied to a programming element at the beginning of a source file. Attribute modifiers are not allowed on

attributes applied to elements that are not at the beginning of a source file.

- **Arguments.** All positional arguments for an attribute must precede any variable or property initializers.

Example

The following example applies the [DllImportAttribute](#) attribute to a skeleton definition of a [Function](#) procedure.

```
<DllImportAttribute("kernel32.dll", EntryPoint:="MoveFileW",
    SetLastError:=True, CharSet:=CharSet.Unicode,
    ExactSpelling:=True,
    CallingConvention:=CallingConvention.StdCall)>
Public Shared Function MoveFile(ByVal src As String,
    ByVal dst As String) As Boolean
    ' This function copies a file from the path src to the path dst.
    ' Leave this function empty. The DllImport attribute forces calls
    ' to MoveFile to be forwarded to MoveFileW in KERNEL32.DLL.
End Function
```

[DllImportAttribute](#) indicates that the attributed procedure represents an entry point in an unmanaged dynamic-link library (DLL). The attribute supplies the DLL name as a positional argument and the other information as variable initializers.

See also

- [Assembly](#)
- [Module <keyword>](#)
- [Attributes overview](#)
- [How to: Break and Combine Statements in Code](#)

Parameter List (Visual Basic)

10/18/2019 • 3 minutes to read • [Edit Online](#)

Specifies the parameters a procedure expects when it is called. Multiple parameters are separated by commas. The following is the syntax for one parameter.

Syntax

```
[ <attributelist> ] [ Optional ] [{ ByVal | ByRef }] [ ParamArray ]  
parametername[( )] [ As parametertype ] [ = defaultvalue ]
```

Parts

`attributelist`

Optional. List of attributes that apply to this parameter. You must enclose the [Attribute List](#) in angle brackets ("<" and ">").

`Optional`

Optional. Specifies that this parameter is not required when the procedure is called.

`ByVal`

Optional. Specifies that the procedure cannot replace or reassign the variable element underlying the corresponding argument in the calling code.

`ByRef`

Optional. Specifies that the procedure can modify the underlying variable element in the calling code the same way the calling code itself can.

`ParamArray`

Optional. Specifies that the last parameter in the parameter list is an optional array of elements of the specified data type. This lets the calling code pass an arbitrary number of arguments to the procedure.

`parametername`

Required. Name of the local variable representing the parameter.

`parametertype`

Required if `Option Strict` is `On`. Data type of the local variable representing the parameter.

`defaultvalue`

Required for `Optional` parameters. Any constant or constant expression that evaluates to the data type of the parameter. If the type is `Object`, or a class, interface, array, or structure, the default value can only be `Nothing`.

Remarks

Parameters are surrounded by parentheses and separated by commas. A parameter can be declared with any data type. If you do not specify `parametertype`, it defaults to `Object`.

When the calling code calls the procedure, it passes an *argument* to each required parameter. For more information, see [Differences Between Parameters and Arguments](#).

The argument the calling code passes to each parameter is a pointer to an underlying element in the calling

code. If this element is *nonvariable* (a constant, literal, enumeration, or expression), it is impossible for any code to change it. If it is a *variable* element (a declared variable, field, property, array element, or structure element), the calling code can change it. For more information, see [Differences Between Modifiable and Nonmodifiable Arguments](#).

If a variable element is passed `ByRef`, the procedure can change it as well. For more information, see [Differences Between Passing an Argument By Value and By Reference](#).

Rules

- **Parentheses.** If you specify a parameter list, you must enclose it in parentheses. If there are no parameters, you can still use parentheses enclosing an empty list. This improves the readability of your code by clarifying that the element is a procedure.
- **Optional Parameters.** If you use the `optional` modifier on a parameter, all subsequent parameters in the list must also be optional and be declared by using the `optional` modifier.

Every optional parameter declaration must supply the `defaultvalue` clause.

For more information, see [Optional Parameters](#).

- **Parameter Arrays.** You must specify `ByVal` for a `ParamArray` parameter.

You cannot use both `Optional` and `ParamArray` in the same parameter list.

For more information, see [Parameter Arrays](#).

- **Passing Mechanism.** The default mechanism for every argument is `ByVal`, which means the procedure cannot change the underlying variable element. However, if the element is a reference type, the procedure can modify the contents or members of the underlying object, even though it cannot replace or reassign the object itself.
- **Parameter Names.** If the parameter's data type is an array, follow `parametername` immediately by parentheses. For more information on parameter names, see [Declared Element Names](#).

Example

The following example shows a `Function` procedure that defines two parameters.

```
Public Function HowMany(ByVal ch As Char, ByVal st As String) As Integer
End Function
Dim howManyA As Integer = HowMany("a"c, "How many a's in this string?")
```

See also

- [DllImportAttribute](#)
- [Function Statement](#)
- [Sub Statement](#)
- [Declare Statement](#)
- [Structure Statement](#)
- [Option Strict Statement](#)
- [Attributes overview](#)
- [How to: Break and Combine Statements in Code](#)

Type List (Visual Basic)

10/18/2019 • 3 minutes to read • [Edit Online](#)

Specifies the *type parameters* for a *generic* programming element. Multiple parameters are separated by commas. Following is the syntax for one type parameter.

Syntax

```
[genericmodifier] typename [ As constraintlist ]
```

Parts

TERM	DEFINITION
<code>genericmodifier</code>	Optional. Can be used only in generic interfaces and delegates. You can declare a type covariant by using the <code>Out</code> keyword or contravariant by using the <code>In</code> keyword. See Covariance and Contravariance .
<code>typename</code>	Required. Name of the type parameter. This is a placeholder, to be replaced by a defined type supplied by the corresponding type argument.
<code>constraintlist</code>	Optional. List of requirements that constrain the data type that can be supplied for <code>typename</code> . If you have multiple constraints, enclose them in curly braces (<code>{ }</code>) and separate them with commas. You must introduce the constraint list with the <code>As</code> keyword. You use <code>As</code> only once, at the beginning of the list.

Remarks

Every generic programming element must take at least one type parameter. A type parameter is a placeholder for a specific type (a *constructed element*) that client code specifies when it creates an instance of the generic type. You can define a generic class, structure, interface, procedure, or delegate.

For more information on when to define a generic type, see [Generic Types in Visual Basic](#). For more information on type parameter names, see [Declared Element Names](#).

Rules

- **Parentheses.** If you supply a type parameter list, you must enclose it in parentheses, and you must introduce the list with the `Of` keyword. You use `Of` only once, at the beginning of the list.
- **Constraints.** A list of *constraints* on a type parameter can include the following items in any combination:
 - Any number of interfaces. The supplied type must implement every interface in this list.
 - At most one class. The supplied type must inherit from that class.

- The `New` keyword. The supplied type must expose a parameterless constructor that your generic type can access. This is useful if you constrain a type parameter by one or more interfaces. A type that implements interfaces does not necessarily expose a constructor, and depending on the access level of a constructor, the code within the generic type might not be able to access it.
- Either the `Class` keyword or the `Structure` keyword. The `Class` keyword constrains a generic type parameter to require that any type argument passed to it be a reference type, for example a string, array, or delegate, or an object created from a class. The `Structure` keyword constrains a generic type parameter to require that any type argument passed to it be a value type, for example a structure, enumeration, or elementary data type. You cannot include both `Class` and `Structure` in the same `constraintlist`.

The supplied type must satisfy every requirement you include in `constraintlist`.

Constraints on each type parameter are independent of constraints on other type parameters.

Behavior

- **Compile-Time Substitution.** When you create a constructed type from a generic programming element, you supply a defined type for each type parameter. The Visual Basic compiler substitutes that supplied type for every occurrence of `typename` within the generic element.
- **Absence of Constraints.** If you do not specify any constraints on a type parameter, your code is limited to the operations and members supported by the [Object Data Type](#) for that type parameter.

Example

The following example shows a skeleton definition of a generic dictionary class, including a skeleton function to add a new entry to the dictionary.

```
Public Class dictionary(Of entryType, keyType As {IComparable, IFormattable, New})
    Public Sub add(ByVal et As entryType, ByVal kt As keyType)
        Dim dk As keyType
        If kt.CompareTo(dk) = 0 Then
            End If
        End Sub
    End Class
```

Example

Because `dictionary` is generic, the code that uses it can create a variety of objects from it, each having the same functionality but acting on a different data type. The following example shows a line of code that creates a `dictionary` object with `String` entries and `Integer` keys.

```
Dim dictInt As New dictionary(Of String, Integer)
```

Example

The following example shows the equivalent skeleton definition generated by the preceding example.

```
Public Class dictionary
    Public Sub Add(ByVal et As String, ByVal kt As Integer)
        Dim dk As Integer
        If kt.CompareTo(dk) = 0 Then
            End If
    End Sub
End Class
```

See also

- [Of](#)
- [New Operator](#)
- [Access levels in Visual Basic](#)
- [Object Data Type](#)
- [Function Statement](#)
- [Structure Statement](#)
- [Sub Statement](#)
- [How to: Use a Generic Class](#)
- [Covariance and Contravariance](#)
- [In](#)
- [Out](#)

Recommended XML Tags for Documentation Comments (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

The Visual Basic compiler can process documentation comments in your code to an XML file. You can use additional tools to process the XML file into documentation.

XML comments are allowed on code constructs such as types and type members. For partial types, only one part of the type can have XML comments, although there is no restriction on commenting its members.

NOTE

Documentation comments cannot be applied to namespaces. The reason is that one namespace can span several assemblies, and not all assemblies have to be loaded at the same time.

The compiler processes any tag that is valid XML. The following tags provide commonly used functionality in user documentation.

<c>	<code>	<example>
<exception> ¹	<include> ¹	<list>
<para>	<param> ¹	<paramref>
<permission> ¹	<remarks>	<returns>
<see> ¹	<seealso> ¹	<summary>
<typeparam> ¹	<value>	

(¹ The compiler verifies syntax.)

NOTE

If you want angle brackets to appear in the text of a documentation comment, use < and >. For example, the string "<text in angle brackets>" will appear as <text in angle brackets> .

See also

- [Documenting Your Code with XML](#)
- [-doc](#)
- [How to: Create XML Documentation](#)

<c> (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

Indicates that text within a description is code.

Syntax

```
<c>text</c>
```

Parameters

PARAMETER	DESCRIPTION
text	The text you would like to indicate as code.

Remarks

The `<c>` tag gives you a way to indicate that text within a description should be marked as code. Use [<code>](#) to indicate multiple lines as code.

Compile with `-doc` to process documentation comments to a file.

Example

This example uses the `<c>` tag in the summary section to indicate that `Counter` is code.

```
''' <summary>
''' Resets the value the <c>Counter</c> field.
''' </summary>
Public Sub ResetCounter()
    counterValue = 0
End Sub
Private counterValue As Integer = 0
''' <summary>
''' Returns the number of times Counter was called.
''' </summary>
''' <value>Number of times Counter was called.</value>
Public ReadOnly Property Counter() As Integer
    Get
        counterValue += 1
        Return counterValue
    End Get
End Property
```

See also

- [XML Comment Tags](#)

<code> (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

Indicates that the text is multiple lines of code.

Syntax

```
<code>content</code>
```

Parameters

content

The text to mark as code.

Remarks

Use the `<code>` tag to indicate multiple lines as code. Use `<c>` to indicate that text within a description should be marked as code.

Compile with `-doc` to process documentation comments to a file.

Example

This example uses the `<code>` tag to include example code for using the `ID` field.

```
Public Class Employee
    ''' <remarks>
    ''' <example> This sample shows how to set the <c>ID</c> field.
    ''' <code>
    ''' Dim alice As New Employee
    ''' alice.ID = 1234
    ''' </code>
    ''' </example>
    ''' </remarks>
    Public ID As Integer
End Class
```

See also

- [XML Comment Tags](#)

<example> (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

Specifies an example for the member.

Syntax

```
<example>description</example>
```

Parameters

`description`

A description of the code sample.

Remarks

The `<example>` tag lets you specify an example of how to use a method or other library member. This commonly involves using the `<code>` tag.

Compile with `-doc` to process documentation comments to a file.

Example

This example uses the `<example>` tag to include an example for using the `ID` field.

```
Public Class Employee
    ''' <remarks>
    ''' <example> This sample shows how to set the <c>ID</c> field.
    ''' <code>
    ''' Dim alice As New Employee
    ''' alice.ID = 1234
    ''' </code>
    ''' </example>
    ''' </remarks>
    Public ID As Integer
End Class
```

See also

- [XML Comment Tags](#)

<exception> (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

Specifies which exceptions can be thrown.

Syntax

```
<exception cref="member">description</exception>
```

Parameters

`member`

A reference to an exception that is available from the current compilation environment. The compiler checks that the given exception exists and translates `member` to the canonical element name in the output XML. `member` must appear within double quotation marks ("").

`description`

A description.

Remarks

Use the `<exception>` tag to specify which exceptions can be thrown. This tag is applied to a method definition.

Compile with `-doc` to process documentation comments to a file.

Example

This example uses the `<exception>` tag to describe an exception that the `IntDivide` function can throw.

```
''' <exception cref="System.OverflowException">
''' Thrown when <paramref name="denominator"/><c> = 0</c>.
''' </exception>
Public Function IntDivide(
    ByVal numerator As Integer,
    ByVal denominator As Integer
) As Integer
    Return numerator \ denominator
End Function
```

See also

- [XML Comment Tags](#)

<include> (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Refers to another file that describes the types and members in your source code.

Syntax

```
<include file="filename" path="tagpath[@name='id']" />
```

Parameters

`filename`

Required. The name of the file containing the documentation. The file name can be qualified with a path. Enclose `filename` in double quotation marks ("").

`tagpath`

Required. The path of the tags in `filename` that leads to the tag `name`. Enclose the path in double quotation marks ("").

`name`

Required. The name specifier in the tag that precedes the comments. `Name` will have an `id`.

`id`

Required. The ID for the tag that precedes the comments. Enclose the ID in single quotation marks ('').

Remarks

Use the `<include>` tag to refer to comments in another file that describe the types and members in your source code. This is an alternative to placing documentation comments directly in your source code file.

The `<include>` tag uses the W3C XML Path Language (XPath) Version 1.0 Recommendation. For more information about ways to customize your `<include>` use, see <https://www.w3.org/TR/xpath>.

Example

This example uses the `<include>` tag to import member documentation comments from a file called `commentFile.xml`.

```
''' <include file="commentFile.xml"
''' path="Docs/Members[@name='Open']/*" />
Public Sub Open(ByVal filename As String)
    ' Code goes here.
End Sub
''' <include file="commentFile.xml"
''' path="Docs/Members[@name='Close']/*" />
Public Sub Close(ByVal filename As String)
    ' Code goes here.
End Sub
```

The format of the `commentFile.xml` is as follows.

```
<Docs>
<Members name="Open">
<summary>Opens a file.</summary>
<param name="filename">File name to open.</param>
</Members>
<Members name="Close">
<summary>Closes a file.</summary>
<param name="filename">File name to close.</param>
</Members>
</Docs>
```

See also

- [XML Comment Tags](#)

<list> (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

Defines a list or table.

Syntax

```
<list type="type">
  <listheader>
    <term>term</term>
    <description>description</description>
  </listheader>
  <item>
    <term>term</term>
    <description>description</description>
  </item>
</list>
```

Parameters

`type`

The type of the list. Must be a "bullet" for a bulleted list, "number" for a numbered list, or "table" for a two-column table.

`term`

Only used when `type` is "table." A term to define, which is defined in the description tag.

`description`

When `type` is "bullet" or "number," `description` is an item in the list. When `type` is "table," `description` is the definition of `term`.

Remarks

The `<listheader>` block defines the heading of either a table or definition list. When defining a table, you only have to supply an entry for `term` in the heading.

Each item in the list is specified with an `<item>` block. When creating a definition list, you must specify both `term` and `description`. However, for a table, bulleted list, or numbered list, you only have to supply an entry for `description`.

A list or table can have as many `<item>` blocks as needed.

Compile with `-doc` to process documentation comments to a file.

Example

This example uses the `<list>` tag to define a bulleted list in the remarks section.

```
''' <remarks>Before calling the <c>Reset</c> method, be sure to:  
''' <list type="bullet">  
''' <item><description>Close all connections.</description></item>  
''' <item><description>Save the object state.</description></item>  
''' </list>  
''' </remarks>  
Public Sub Reset()  
    ' Code goes here.  
End Sub
```

See also

- [XML Comment Tags](#)

<para> (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

Specifies that the content is formatted as a paragraph.

Syntax

```
<para>content</para>
```

Parameters

content

The text of the paragraph.

Remarks

The `<para>` tag is for use inside a tag, such as `<summary>`, `<remarks>`, or `<returns>`, and lets you add structure to the text.

Compile with `-doc` to process documentation comments to a file.

Example

This example uses the `<para>` tag to split the remarks section for the `UpdateRecord` method into two paragraphs.

```
''' <param name="id">The ID of the record to update.</param>
''' <remarks>Updates the record <paramref name="id"/>.
''' <para>Use <see cref="DoesRecordExist"/> to verify that
''' the record exists before calling this method.</para>
''' </remarks>
Public Sub UpdateRecord(ByVal id As Integer)
    ' Code goes here.
End Sub
''' <param name="id">The ID of the record to check.</param>
''' <returns><c>True</c> if <paramref name="id"/> exists,
''' <c>False</c> otherwise.</returns>
''' <remarks><seealso cref="UpdateRecord"/></remarks>
Public Function DoesRecordExist(ByVal id As Integer) As Boolean
    ' Code goes here.
End Function
```

See also

- [XML Comment Tags](#)

<param> (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

Defines a parameter name and description.

Syntax

```
<param name="name">description</param>
```

Parameters

name

The name of a method parameter. Enclose the name in double quotation marks ("").

description

A description for the parameter.

Remarks

The `<param>` tag should be used in the comment for a method declaration to describe one of the parameters for the method.

The text for the `<param>` tag will appear in the following locations:

- Parameter Info of IntelliSense. For more information, see [Using IntelliSense](#).
- Object Browser. For more information, see [Viewing the Structure of Code](#).

Compile with `-doc` to process documentation comments to a file.

Example

This example uses the `<param>` tag to describe the `id` parameter.

```
''' <param name="id">The ID of the record to update.</param>
''' <remarks>Updates the record <paramref name="id"/>.
''' <para>Use <see cref="DoesRecordExist"/> to verify that
''' the record exists before calling this method.</para>
''' </remarks>
Public Sub UpdateRecord(ByVal id As Integer)
    ' Code goes here.
End Sub
''' <param name="id">The ID of the record to check.</param>
''' <returns><c>True</c> if <paramref name="id"/> exists,
''' <c>False</c> otherwise.</returns>
''' <remarks><seealso cref="UpdateRecord"/></remarks>
Public Function DoesRecordExist(ByVal id As Integer) As Boolean
    ' Code goes here.
End Function
```

See also

- XML Comment Tags

<paramref> (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

Formats a word as a parameter.

Syntax

```
<paramref name="name"/>
```

Parameters

name

The name of the parameter to refer to. Enclose the name in double quotation marks ("").

Remarks

The `<paramref>` tag gives you a way to indicate that a word is a parameter. The XML file can be processed to format this parameter in some distinct way.

Compile with `-doc` to process documentation comments to a file.

Example

This example uses the `<paramref>` tag to refer to the `id` parameter.

```
''' <param name="id">The ID of the record to update.</param>
''' <remarks>Updates the record <paramref name="id"/>.
''' <para>Use <see cref="DoesRecordExist"/> to verify that
''' the record exists before calling this method.</para>
''' </remarks>
Public Sub UpdateRecord(ByVal id As Integer)
    ' Code goes here.
End Sub
''' <param name="id">The ID of the record to check.</param>
''' <returns><c>True</c> if <paramref name="id"/> exists,
''' <c>False</c> otherwise.</returns>
''' <remarks><seealso cref="UpdateRecord"/></remarks>
Public Function DoesRecordExist(ByVal id As Integer) As Boolean
    ' Code goes here.
End Function
```

See also

- [XML Comment Tags](#)

<permission> (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

Specifies a required permission for the member.

Syntax

```
<permission cref="member">description</permission>
```

Parameters

`member`

A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and translates `member` to the canonical element name in the output XML. Enclose `member` in quotation marks ("").

`description`

A description of the access to the member.

Remarks

Use the `<permission>` tag to document the access of a member. Use the [PermissionSet](#) class to specify access to a member.

Compile with `-doc` to process documentation comments to a file.

Example

This example uses the `<permission>` tag to describe that the [FileIOPermission](#) is required by the `ReadFile` method.

```
''' <permission cref="System.Security.Permissions.FileIOPermission">
''' Needs full access to the specified file.
''' </permission>
Public Sub ReadFile(ByVal filename As String)
    ' Code goes here.
End Sub
```

See also

- [XML Comment Tags](#)

<remarks> (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

Specifies a remarks section for the member.

Syntax

```
<remarks>description</remarks>
```

Parameters

`description`

A description of the member.

Remarks

Use the `<remarks>` tag to add information about a type, supplementing the information specified with [`<summary>`](#).

This information appears in the Object Browser. For information about the Object Browser, see [Viewing the Structure of Code](#).

Compile with `-doc` to process documentation comments to a file.

Example

This example uses the `<remarks>` tag to explain what the `UpdateRecord` method does.

```
''' <param name="id">The ID of the record to update.</param>
''' <remarks>Updates the record <paramref name="id"/>.
''' <para>Use <see cref="DoesRecordExist"/> to verify that
''' the record exists before calling this method.</para>
''' </remarks>
Public Sub UpdateRecord(ByVal id As Integer)
    ' Code goes here.
End Sub
''' <param name="id">The ID of the record to check.</param>
''' <returns><c>True</c> if <paramref name="id"/> exists,
''' <c>False</c> otherwise.</returns>
''' <remarks><seealso cref="UpdateRecord"/></remarks>
Public Function DoesRecordExist(ByVal id As Integer) As Boolean
    ' Code goes here.
End Function
```

See also

- [XML Comment Tags](#)

<returns> (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

Specifies the return value of the property or function.

Syntax

```
<returns>description</returns>
```

Parameters

`description`

A description of the return value.

Remarks

Use the `<returns>` tag in the comment for a method declaration to describe the return value.

Compile with `-doc` to process documentation comments to a file.

Example

This example uses the `<returns>` tag to explain what the `DoesRecordExist` function returns.

```
''' <param name="id">The ID of the record to update.</param>
''' <remarks>Updates the record <paramref name="id"/>.
''' <para>Use <see cref="DoesRecordExist"/> to verify that
''' the record exists before calling this method.</para>
''' </remarks>
Public Sub UpdateRecord(ByVal id As Integer)
    ' Code goes here.
End Sub
''' <param name="id">The ID of the record to check.</param>
''' <returns><c>True</c> if <paramref name="id"/> exists,
''' <c>False</c> otherwise.</returns>
''' <remarks><seealso cref="UpdateRecord"/></remarks>
Public Function DoesRecordExist(ByVal id As Integer) As Boolean
    ' Code goes here.
End Function
```

See also

- [XML Comment Tags](#)

<see> (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

Specifies a link to another member.

Syntax

```
<see cref="member"/>
```

Parameters

member

A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and passes member to the element name in the output XML. member must appear within double quotation marks (" ").

Remarks

Use the <see> tag to specify a link from within text. Use <seealso> to indicate text that you might want to appear in a "See Also" section.

Compile with `-doc` to process documentation comments to a file.

Example

This example uses the <see> tag in the UpdateRecord remarks section to refer to the DoesRecordExist method.

```
''' <param name="id">The ID of the record to update.</param>
''' <remarks>Updates the record <paramref name="id"/>.
''' <para>Use <see cref="DoesRecordExist"/> to verify that
''' the record exists before calling this method.</para>
''' </remarks>
Public Sub UpdateRecord(ByVal id As Integer)
    ' Code goes here.
End Sub
''' <param name="id">The ID of the record to check.</param>
''' <returns><c>True</c> if <paramref name="id"/> exists,
''' <c>False</c> otherwise.</returns>
''' <remarks><seealso cref="UpdateRecord"/></remarks>
Public Function DoesRecordExist(ByVal id As Integer) As Boolean
    ' Code goes here.
End Function
```

See also

- [XML Comment Tags](#)

<seealso> (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

Specifies a link that appears in the See Also section.

Syntax

```
<seealso cref="member"/>
```

Parameters

member

A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and passes member to the element name in the output XML. member must appear within double quotation marks (" ").

Remarks

Use the <seealso> tag to specify the text that you want to appear in a See Also section. Use <see> to specify a link from within text.

Compile with `-doc` to process documentation comments to a file.

Example

This example uses the <seealso> tag in the DoesRecordExist remarks section to refer to the UpdateRecord method.

```
''' <param name="id">The ID of the record to update.</param>
''' <remarks>Updates the record <paramref name="id"/>.
''' <para>Use <see cref="DoesRecordExist"/> to verify that
''' the record exists before calling this method.</para>
''' </remarks>
Public Sub UpdateRecord(ByVal id As Integer)
    ' Code goes here.
End Sub
''' <param name="id">The ID of the record to check.</param>
''' <returns><c>True</c> if <paramref name="id"/> exists,
''' <c>False</c> otherwise.</returns>
''' <remarks><seealso cref="UpdateRecord"/></remarks>
Public Function DoesRecordExist(ByVal id As Integer) As Boolean
    ' Code goes here.
End Function
```

See also

- [XML Comment Tags](#)

<summary> (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

Specifies the summary of the member.

Syntax

```
<summary>description</summary>
```

Parameters

`description`

A summary of the object.

Remarks

Use the `<summary>` tag to describe a type or a type member. Use [`<remarks>`](#) to add supplemental information to a type description.

The text for the `<summary>` tag is the only source of information about the type in IntelliSense, and is also displayed in the Object Browser. For information about the Object Browser, see [Viewing the Structure of Code](#).

Compile with `-doc` to process documentation comments to a file.

Example

This example uses the `<summary>` tag to describe the `ResetCounter` method and `Counter` property.

```
''' <summary>
''' Resets the value the <c>Counter</c> field.
''' </summary>
Public Sub ResetCounter()
    counterValue = 0
End Sub
Private counterValue As Integer = 0
''' <summary>
''' Returns the number of times Counter was called.
''' </summary>
''' <value>Number of times Counter was called.</value>
Public ReadOnly Property Counter() As Integer
    Get
        counterValue += 1
        Return counterValue
    End Get
End Property
```

See also

- [XML Comment Tags](#)

<typeparam> (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

Defines a type parameter name and description.

Syntax

```
<typeparam name="name">description</typeparam>
```

Parameters

`name`

The name of the type parameter. Enclose the name in double quotation marks ("").

`description`

A description of the type parameter.

Remarks

Use the `<typeparam>` tag in the comment for a generic type or generic member declaration to describe one of the type parameters.

Compile with `-doc` to process documentation comments to a file.

Example

This example uses the `<typeparam>` tag to describe the `id` parameter.

```
''' <typeparam name="T">
''' The base item type. Must implement IComparable.
''' </typeparam>
Public Class itemManager(Of T As IComparable)
    ' Insert code that defines class members.
End Class
```

See also

- [XML Comment Tags](#)

<value> (Visual Basic)

10/17/2019 • 2 minutes to read • [Edit Online](#)

Specifies the description of a property.

Syntax

```
<value>property-description</value>
```

Parameters

property-description

A description for the property.

Remarks

Use the `<value>` tag to describe a property. Note that when you add a property using the code wizard in the Visual Studio development environment, it will add a `<summary>` tag for the new property. You should then manually add a `<value>` tag to describe the value that the property represents.

Compile with `-doc` to process documentation comments to a file.

Example

This example uses the `<value>` tag to describe what value the `Counter` property holds.

```
''' <summary>
''' Resets the value the <c>Counter</c> field.
''' </summary>
Public Sub ResetCounter()
    counterValue = 0
End Sub
Private counterValue As Integer = 0
''' <summary>
''' Returns the number of times Counter was called.
''' </summary>
''' <value>Number of times Counter was called.</value>
Public ReadOnly Property Counter() As Integer
    Get
        counterValue += 1
        Return counterValue
    End Get
End Property
```

See also

- [XML Comment Tags](#)

XML Axis Properties (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section document the syntax of XML axis properties in Visual Basic. The XML axis properties make it easy to access XML directly in your code.

In This Section

TOPIC	DESCRIPTION
XML Attribute Axis Property	Describes how to access the attributes of an <code>XElement</code> object.
XML Child Axis Property	Describes how to access the children of an <code>XElement</code> object.
XML Descendant Axis Property	Describes how to access the descendants of an <code>XElement</code> object.
Extension Indexer Property	Describes how to access individual elements in a collection of <code>XElement</code> or <code>XAttribute</code> objects.
XML Value Property	Describes how to access the value of the first element of a collection of <code>XElement</code> or <code>XAttribute</code> objects.

See also

- [XML](#)

XML Attribute Axis Property (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Provides access to the value of an attribute for an [XElement](#) object or to the first element in a collection of [XElement](#) objects.

Syntax

```
object.@attribute  
' -or-  
object.<attribute>
```

Parts

`object`

Required. An [XElement](#) object or a collection of [XElement](#) objects.

`.@`

Required. Denotes the start of an attribute axis property.

`<`

Optional. Denotes the beginning of the name of the attribute when `attribute` is not a valid identifier in Visual Basic.

`attribute`

Required. Name of the attribute to access, of the form `[prefix :] name`.

PART	DESCRIPTION
<code>prefix</code>	Optional. XML namespace prefix for the attribute. Must be a global XML namespace defined with an Imports statement.
<code>name</code>	Required. Local attribute name. See Names of Declared XML Elements and Attributes .

`>`

Optional. Denotes the end of the name of the attribute when `attribute` is not a valid identifier in Visual Basic.

Return Value

A string that contains the value of `attribute`. If the attribute name does not exist, `Nothing` is returned.

Remarks

You can use an XML attribute axis property to access the value of an attribute by name from an [XElement](#) object or from the first element in a collection of [XElement](#) objects. You can retrieve an attribute value by name, or add a new attribute to an element by specifying a new name preceded by the @ identifier.

When you refer to an XML attribute using the @ identifier, the attribute value is returned as a string and you do not need to explicitly specify the [Value](#) property.

The naming rules for XML attributes differ from the naming rules for Visual Basic identifiers. To access an XML attribute that has a name that is not a valid Visual Basic identifier, enclose the name in angle brackets (< and >).

XML Namespaces

The name in an attribute axis property can use only XML namespace prefixes declared globally by using the `Imports` statement. It cannot use XML namespace prefixes declared locally within XML element literals. For more information, see [Imports Statement \(XML Namespace\)](#).

Example

The following example shows how to get the values of the XML attributes named `type` from a collection of XML elements that are named `phone`.

```
' Topic: XML Attribute Axis Property
Dim phones As XElement =
    <phones>
        <phone type="home">206-555-0144</phone>
        <phone type="work">425-555-0145</phone>
    </phones>

Dim phoneTypes As XElement =
    <phoneTypes>
        <%= From phone In phones.<phone>
            Select <type><%= phone.@type %></type>
        %>
    </phoneTypes>

Console.WriteLine(phoneTypes)
```

This code displays the following text:

```
<phoneTypes>
    <type>home</type>
    <type>work</type>
</phoneTypes>
```

Example

The following example shows how to create attributes for an XML element both declaratively, as part of the XML, and dynamically by adding an attribute to an instance of an `XElement` object. The `type` attribute is created declaratively and the `owner` attribute is created dynamically.

```
Dim phone2 As XElement = <phone type="home">206-555-0144</phone>
phone2.@owner = "Harris, Phyllis"

Console.WriteLine(phone2)
```

This code displays the following text:

```
<phone type="home" owner="Harris, Phyllis">206-555-0144</phone>
```

Example

The following example uses the angle bracket syntax to get the value of the XML attribute named `number-type`, which is not a valid identifier in Visual Basic.

```
Dim phone As XElement =
    <phone number-type=" work">425-555-0145</phone>

Console.WriteLine("Phone type: " & phone.@<number-type>)
```

This code displays the following text:

```
Phone type: work
```

Example

The following example declares `ns` as an XML namespace prefix. It then uses the prefix of the namespace to create an XML literal and access the first child node with the qualified name "`ns:name`".

```
Imports <xmllns:ns = "http://SomeNamespace">

Class TestClass3

    Shared Sub TestPrefix()
        Dim phone =
            <ns:phone ns:type="home">206-555-0144</ns:phone>

        Console.WriteLine("Phone type: " & phone.@ns:type)
    End Sub

End Class
```

This code displays the following text:

```
Phone type: home
```

See also

- [XElement](#)
- [XML Axis Properties](#)
- [XML Literals](#)
- [Creating XML in Visual Basic](#)
- [Names of Declared XML Elements and Attributes](#)

XML Child Axis Property (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Provides access to the children of one of the following: an [XElement](#) object, an [XDocument](#) object, a collection of [XElement](#) objects, or a collection of [XDocument](#) objects.

Syntax

```
object.<child>
```

Parts

TERM	DEFINITION
<code>object</code>	Required. An XElement object, an XDocument object, a collection of XElement objects, or a collection of XDocument objects.
<code>.<</code>	Required. Denotes the start of a child axis property.
<code>child</code>	Required. Name of the child nodes to access, of the form [<code>prefix:]name</code>]. - <code>Prefix</code> - Optional. XML namespace prefix for the child node. Must be a global XML namespace defined with an Imports statement. - <code>Name</code> - Required. Local child node name. See Names of Declared XML Elements and Attributes .
<code>></code>	Required. Denotes the end of a child axis property.

Return Value

A collection of [XElement](#) objects.

Remarks

You can use an XML child axis property to access child nodes by name from an [XElement](#) or [XDocument](#) object, or from a collection of [XElement](#) or [XDocument](#) objects. Use the XML [Value](#) property to access the value of the first child node in the returned collection. For more information, see [XML Value Property](#).

The Visual Basic compiler converts child axis properties to calls to the [Elements](#) method.

XML Namespaces

The name in a child axis property can use only XML namespace prefixes declared globally with the [Imports](#) statement. It cannot use XML namespace prefixes declared locally within XML element literals. For more information, see [Imports Statement \(XML Namespace\)](#).

Example

The following example shows how to access the child nodes named `phone` from the `contact` object.

```
Dim contact As XElement =
<contact>
    <name>Patrick Hines</name>
    <phone type="home">206-555-0144</phone>
    <phone type="work">425-555-0145</phone>
</contact>

Dim homePhone = From hp In contact.<phone>
                  Where contact.<phone>.@type = "home"
                  Select hp

Console.WriteLine("Home Phone = {0}", homePhone(0).Value)
```

This code displays the following text:

```
Home Phone = 206-555-0144
```

Example

The following example shows how to access the child nodes named `phone` from the collection returned by the `contact` child axis property of the `contacts` object.

```
Dim contacts As XElement =
<contacts>
    <contact>
        <name>Patrick Hines</name>
        <phone type="home">206-555-0144</phone>
    </contact>
    <contact>
        <name>Lance Tucker</name>
        <phone type="work">425-555-0145</phone>
    </contact>
</contacts>

Dim homePhone = From contact In contacts.<contact>
                  Where contact.<phone>.@type = "home"
                  Select contact.<phone>

Console.WriteLine("Home Phone = {0}", homePhone(0).Value)
```

This code displays the following text:

```
Home Phone = 206-555-0144
```

Example

The following example declares `ns` as an XML namespace prefix. It then uses the prefix of the namespace to create an XML literal and access the first child node with the qualified name `ns:name`.

```
Imports <xmlns:ns = "http://SomeNamespace">

Class TestClass4

    Shared Sub TestPrefix()
        Dim contact = <ns:contact>
            <ns:name>Patrick Hines</ns:name>
        </ns:contact>
        Console.WriteLine(contact.<ns:name>.Value)
    End Sub

End Class
```

This code displays the following text:

```
Patrick Hines
```

See also

- [XElement](#)
- [XML Axis Properties](#)
- [XML Literals](#)
- [Creating XML in Visual Basic](#)
- [Names of Declared XML Elements and Attributes](#)

XML Descendant Axis Property (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Provides access to the descendants of the following: an [XElement](#) object, an [XDocument](#) object, a collection of [XElement](#) objects, or a collection of [XDocument](#) objects.

Syntax

```
object...<descendant>
```

Parts

object Required. An [XElement](#) object, an [XDocument](#) object, a collection of [XElement](#) objects, or a collection of [XDocument](#) objects.

...< Required. Denotes the start of a descendant axis property.

descendant Required. Name of the descendant nodes to access, of the form [**prefix:**][name](#).

PART	DESCRIPTION
prefix	Optional. XML namespace prefix for the descendant node. Must be a global XML namespace that is defined by using an Imports statement.
name	Required. Local name of the descendant node. See Names of Declared XML Elements and Attributes .

> Required. Denotes the end of a descendant axis property.

Return Value

A collection of [XElement](#) objects.

Remarks

You can use an XML descendant axis property to access descendant nodes by name from an [XElement](#) or [XDocument](#) object, or from a collection of [XElement](#) or [XDocument](#) objects. Use the XML **Value** property to access the value of the first descendant node in the returned collection. For more information, see [XML Value Property](#).

The Visual Basic compiler converts descendant axis properties into calls to the [Descendants](#) method.

XML Namespaces

The name in a descendant axis property can use only XML namespaces declared globally with the [Imports](#) statement. It cannot use XML namespaces declared locally within XML element literals. For more information, see [Imports Statement \(XML Namespace\)](#).

Example

The following example shows how to access the value of the first descendant node named `name` and the values of all descendant nodes named `phone` from the `contacts` object.

```
Dim contacts As XElement =
    <contacts>
        <contact>
            <name>Patrick Hines</name>
            <phone type="home">206-555-0144</phone>
            <phone type="work">425-555-0145</phone>
        </contact>
    </contacts>

Console.WriteLine("Name: " & contacts...<name>.Value)

Dim homePhone = From phone In contacts...<phone>
                  Select phone.Value

Console.WriteLine("Home Phone = {0}", homePhone(0))
```

This code displays the following text:

Name: Patrick Hines

Home Phone = 206-555-0144

Example

The following example declares `ns` as an XML namespace prefix. It then uses the prefix of the namespace to create an XML literal and access the value of the first child node with the qualified name `ns:name`.

```
Imports <xmldns:ns = "http://SomeNamespace">

Class TestClass2

    Shared Sub TestPrefix()
        Dim contacts =
            <ns:contacts>
                <ns:contact>
                    <ns:name>Patrick Hines</ns:name>
                </ns:contact>
            </ns:contacts>

        Console.WriteLine("Name: " & contacts...<ns:name>.Value)
    End Sub

End Class
```

This code displays the following text:

Name: Patrick Hines

See also

- [XElement](#)
- [XML Axis Properties](#)
- [XML Literals](#)
- [Creating XML in Visual Basic](#)

- Names of Declared XML Elements and Attributes

Extension Indexer Property (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Provides access to individual elements in a collection.

Syntax

```
object(index)
```

Parts

TERM	DEFINITION
object	Required. A queryable collection. That is, a collection that implements IEnumerable<T> or IQueryable<T> .
(Required. Denotes the start of the indexer property.
index	Required. An integer expression that specifies the zero-based position of an element of the collection.
)	Required. Denotes the end of the indexer property.

Return Value

The object from the specified location in the collection, or `Nothing` if the index is out of range.

Remarks

You can use the extension indexer property to access individual elements in a collection. This indexer property is typically used on the output of XML axis properties. The XML child and XML descendant axis properties return collections of [XElement](#) objects or an attribute value.

The Visual Basic compiler converts extension indexer properties to calls to the `ElementAtOrDefault` method. Unlike an array indexer, the `ElementAtOrDefault` method returns `Nothing` if the index is out of range. This behavior is useful when you cannot easily determine the number of elements in a collection.

This indexer property is like an extension property for collections that implement [IEnumerable<T>](#) or [IQueryable<T>](#): it is used only if the collection does not have an indexer or a default property.

To access the value of the first element in a collection of [XElement](#) or [XAttribute](#) objects, you can use the XML `Value` property. For more information, see [XML Value Property](#).

Example

The following example shows how to use the extension indexer to access the second child node in a collection of [XElement](#) objects. The collection is accessed by using the child axis property, which gets all child elements named `phone` in the `contact` object.

```
Dim contact As XElement =
<contact>
  <name>Patrick Hines</name>
  <phone type="home">206-555-0144</phone>
  <phone type="work">425-555-0145</phone>
</contact>

Console.WriteLine("Second phone number: " & contact.<phone>(1).Value)
```

This code displays the following text:

```
Second phone number: 425-555-0145
```

See also

- [XElement](#)
- [XML Axis Properties](#)
- [XML Literals](#)
- [Creating XML in Visual Basic](#)
- [XML Value Property](#)

XML Value Property (Visual Basic)

9/28/2019 • 2 minutes to read • [Edit Online](#)

Provides access to the value of the first element of a collection of [XElement](#) objects.

Syntax

```
object.Value
```

Parts

TERM	DEFINITION
<code>object</code>	Required. Collection of XElement objects.

Return Value

A `String` that contains the value of the first element of the collection, or `Nothing` if the collection is empty.

Remarks

The `Value` property makes it easy to access the value of the first element in a collection of [XElement](#) objects. This property first checks whether the collection contains at least one object. If the collection is empty, this property returns `Nothing`. Otherwise, this property returns the value of the `Value` property of the first element in the collection.

NOTE

When you access the value of an XML attribute using the '@' identifier, the attribute value is returned as a `String` and you do not need to explicitly specify the `Value` property.

To access other elements in a collection, you can use the XML extension indexer property. For more information, see [Extension Indexer Property](#).

Inheritance

Most users will not have to implement `IEnumerable<T>`, and can therefore ignore this section.

The `Value` property is an extension property for types that implement `IEnumerable(Of XElement)`. The binding of this extension property is like the binding of extension methods: if a type implements one of the interfaces and defines a property that has the name "Value", that property has precedence over the extension property. In other words, this `Value` property can be overridden by defining a new property in a class that implements `IEnumerable(Of XElement)`.

Example

The following example shows how to use the `Value` property to access the first node in a collection of [XElement](#)

objects. The example uses the child axis property to get the collection of all child nodes named `phone` that are in the `contact` object.

```
Dim contact As XElement =
<contact>
    <name>Patrick Hines</name>
    <phone type="home">206-555-0144</phone>
    <phone type="work">425-555-0145</phone>
</contact>

Console.WriteLine("Phone number: " & contact.<phone>.Value)
```

This code displays the following text:

```
Phone number: 206-555-0144
```

Example

The following example shows how to get the value of an XML attribute from a collection of [XAttribute](#) objects. The example uses the attribute axis property to display the value of the `type` attribute for all of the `phone` elements.

```
Dim contact As XElement =
<contact>
    <name>Patrick Hines</name>
    <phone type="home">206-555-0144</phone>
    <phone type="work">425-555-0145</phone>
</contact>

Dim types = contact.<phone>.Attributes("type")

For Each attr In types
    Console.WriteLine(attr.Value)
Next
```

This code displays the following text:

```
home
work
```

See also

- [XElement](#)
- [IEnumerable<T>](#)
- [XML Axis Properties](#)
- [XML Literals](#)
- [Creating XML in Visual Basic](#)
- [Extension Methods](#)
- [Extension Indexer Property](#)
- [XML Child Axis Property](#)
- [XML Attribute Axis Property](#)

XML Literals (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

The topics in this section document the syntax of XML literals in Visual Basic. The XML literal syntax enables you to incorporate XML directly in your code.

In This Section

TOPIC	DESCRIPTION
XML Element Literal	Describes the syntax for literals that represent <code>XElement</code> objects.
XML Document Literal	Describes the syntax for literals that represent <code>XDocument</code> objects.
XML CDATA Literal	Describes the syntax for literals that represent <code>XCData</code> objects.
XML Comment Literal	Describes the syntax for literals that represent <code>XComment</code> objects.
XML Processing Instruction Literal	Describes the syntax for literals that represent <code>XProcessingInstruction</code> objects.

See also

- [XML](#)

XML Element Literal (Visual Basic)

4/26/2019 • 4 minutes to read • [Edit Online](#)

A literal that represents an `XElement` object.

Syntax

```
<name [ attributeList ] />  
-or-  
<name [ attributeList ] > [ elementContents ] </[ name ]>
```

Parts

- `<`

Required. Opens the starting element tag.

- `name`

Required. Name of the element. The format is one of the following:

- Literal text for the element name, of the form `[ePrefix:]eName`, where:

PART	DESCRIPTION
<code>ePrefix</code>	Optional. XML namespace prefix for the element. Must be a global XML namespace that is defined with an <code>Imports</code> statement in the file or at the project level, or a local XML namespace that is defined in this element or a parent element.
<code>eName</code>	Required. Name of the element. The format is one of the following: <ul style="list-style-type: none">- Literal text. See Names of Declared XML Elements and Attributes.- Embedded expression of the form <code><%= eNameExp %></code>. The type of <code>eNameExp</code> must be <code>String</code> or a type that is implicitly convertible to <code>XName</code>.

- Embedded expression of the form `<%= nameExp %>`. The type of `nameExp` must be `String` or a type implicitly convertible to `XName`. An embedded expression is not allowed in a closing tag of an element.

- `attributeList`

Optional. List of attributes declared in the literal.

```
attribute [ attribute ... ]
```

Each `attribute` has one of the following syntaxes:

- Attribute assignment, of the form `[aPrefix:]aName=aValue`, where:

PART	DESCRIPTION
aPrefix	Optional. XML namespace prefix for the attribute. Must be a global XML namespace that is defined with an <code>Imports</code> statement, or a local XML namespace that is defined in this element or a parent element.
aName	Required. Name of the attribute. The format is one of the following: - Literal text. See Names of Declared XML Elements and Attributes . - Embedded expression of the form <code><%= aNameExp %></code> . The type of <code>aNameExp</code> must be <code>String</code> or a type that is implicitly convertible to <code>XName</code> .
aValue	Optional. Value of the attribute. The format is one of the following: - Literal text, enclosed in quotation marks. - Embedded expression of the form <code><%= aValueExp %></code> . Any type is allowed.

- Embedded expression of the form `<%= aExp %>`.

- `/>`

Optional. Indicates that the element is an empty element, without content.

- `>`

Required. Ends the beginning or empty element tag.

- `elementContents`

Optional. Content of the element.

`content [content ...]`

Each `content` can be one of the following:

- Literal text. All the white space in `elementContents` becomes significant if there is any literal text.
- Embedded expression of the form `<%= contentExp %>`.
- XML element literal.
- XML comment literal. See [XML Comment Literal](#).
- XML processing instruction literal. See [XML Processing Instruction Literal](#).
- XML CDATA literal. See [XML CDATA Literal](#).

- `</[name]>`

Optional. Represents the closing tag for the element. The optional `name` parameter is not allowed when it is the result of an embedded expression.

Return Value

An [XElement](#) object.

Remarks

You can use the XML element literal syntax to create [XElement](#) objects in your code.

NOTE

An XML literal can span multiple lines without using line continuation characters. This feature enables you to copy content from an XML document and paste it directly into a Visual Basic program.

Embedded expressions of the form `<%= exp %>` enable you to add dynamic information to an XML element literal. For more information, see [Embedded Expressions in XML](#).

The Visual Basic compiler converts the XML element literal into calls to the [XElement](#) constructor and, if it is required, the [XmlAttribute](#) constructor.

XML Namespaces

XML namespace prefixes are useful when you have to create XML literals with elements from the same namespace many times in code. You can use global XML namespace prefixes, which you define by using the [Imports](#) statement, or local prefixes, which you define by using the `xmlns:xmlPrefix="xmlNamespace"` attribute syntax. For more information, see [Imports Statement \(XML Namespace\)](#).

In accordance with the scoping rules for XML namespaces, local prefixes take precedence over global prefixes. However, if an XML literal defines an XML namespace, that namespace is not available to expressions that appear in an embedded expression. The embedded expression can access only the global XML namespace.

The Visual Basic compiler converts each global XML namespace that is used by an XML literal into a one local namespace definition in the generated code. Global XML namespaces that are not used do not appear in the generated code.

Example

The following example shows how to create a simple XML element that has two nested empty elements.

```
Dim test1 As XElement =
<outer>
  <inner1></inner1>
  <inner2/>
</outer>

Console.WriteLine(test1)
```

The example displays the following text. Notice that the literal preserves the structure of the empty elements.

```
<outer>
  <inner1></inner1>
  <inner2 />
</outer>
```

Example

The following example shows how to use embedded expressions to name an element and create attributes.

```

Dim elementType = "book"
Dim authorName = "My Author"
Dim attributeName1 = "year"
Dim attributeValue1 = 1999
Dim attributeName2 = "title"
Dim attributeValue2 = "My Book"

Dim book As XElement =
<<%= elementType %>
    isbn="1234"
    author=<%= authorName %>
    <%= attributeName1 %>=<%= attributeValue1 %>
    <%= New XAttribute(attributeName2, attributeValue2) %>
/>

Console.WriteLine(book)

```

This code displays the following text:

```
<book isbn="1234" author="My Author" year="1999" title="My Book" />
```

Example

The following example declares `ns` as an XML namespace prefix. It then uses the prefix of the namespace to create an XML literal and displays the element's final form.

```

' Place Imports statements at the top of your program.
Imports <xmllns:ns="http://SomeNamespace">

Class TestClass1

    Shared Sub TestPrefix()
        ' Create test using a global XML namespace prefix.
        Dim inner2 = <ns:inner2/>

        Dim test =
<ns:outer>
    <ns:middle xmlns:ns="http://NewNamespace">
        <ns:inner1/>
        <%= inner2 %>
    </ns:middle>
</ns:outer>

        ' Display test to see its final form.
        Console.WriteLine(test)
    End Sub

End Class

```

This code displays the following text:

```
<ns:outer xmlns:ns="http://SomeNamespace">
    <ns:middle xmlns:ns="http://NewNamespace">
        <ns:inner1 />
        <inner2 xmlns="http://SomeNamespace" />
    </ns:middle>
</ns:outer>
```

Notice that the compiler converted the prefix of the global XML namespace into a prefix definition for the XML

namespace. The `<ns:middle>` element redefines the XML namespace prefix for the `<ns:inner1>` element. However, the `<ns:inner2>` element uses the namespace defined by the `Imports` statement.

See also

- [XElement](#)
- [Names of Declared XML Elements and Attributes](#)
- [XML Comment Literal](#)
- [XML CDATA Literal](#)
- [XML Literals](#)
- [Creating XML in Visual Basic](#)
- [Embedded Expressions in XML](#)
- [Imports Statement \(XML Namespace\)](#)

XML Document Literal (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

A literal representing an [XDocument](#) object.

Syntax

```
<?xml version="1.0" [encoding="encoding"] [standalone="standalone"] ?>
[ piCommentList ]
rootElement
[ piCommentList ]
```

Parts

TERM	DEFINITION
<code>encoding</code>	Optional. Literal text declaring which encoding the document uses.
<code>standalone</code>	Optional. Literal text. Must be "yes" or "no".
<code>piCommentList</code>	Optional. List of XML processing instructions and XML comments. Takes the following format: <code>piComment [piComment ...]</code> Each <code>piComment</code> can be one of the following: - XML Processing Instruction Literal . - XML Comment Literal .
<code>rootElement</code>	Required. Root element of the document. The format is one of the following: <ul style="list-style-type: none">• XML Element Literal.• Embedded expression of the form <code><%= elementExp %></code>. The <code>elementExp</code> returns one of the following:<ul style="list-style-type: none">◦ An XElement object.◦ A collection that contains one XElement object and any number of XProcessingInstruction and XComment objects. For more information, see Embedded Expressions in XML .

Return Value

An [XDocument](#) object.

Remarks

An XML document literal is identified by the XML declaration at the start of the literal. Although each XML document literal must have exactly one root XML element, it can have any number of XML processing instructions and XML comments.

An XML document literal cannot appear in an XML element.

NOTE

An XML literal can span multiple lines without using line continuation characters. This enables you to copy content from an XML document and paste it directly into a Visual Basic program.

The Visual Basic compiler converts the XML document literal into calls to the [XDocument](#) and [XDeclaration](#) constructors.

Example

The following example creates an XML document that has an XML declaration, a processing instruction, a comment, and an element that contains another element.

```
Dim libraryRequest As XDocument =
    <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
    <xml-stylesheet type="text/xsl" href="show_book.xsl"?>
    <!-- Tests that the application works. -->
    <books>
        <book/>
    </books>
Console.WriteLine(libraryRequest)
```

See also

- [XElement](#)
- [XProcessingInstruction](#)
- [XComment](#)
- [XDocument](#)
- [XML Processing Instruction Literal](#)
- [XML Comment Literal](#)
- [XML Element Literal](#)
- [XML Literals](#)
- [Creating XML in Visual Basic](#)
- [Embedded Expressions in XML](#)

XML CDATA Literal (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

A literal representing an [XCData](#) object.

Syntax

```
<![CDATA[content]]>
```

Parts

`<![CDATA[`

Required. Denotes the start of the XML CDATA section.

`content`

Required. Text content to appear in the XML CDATA section.

`]]>`

Required. Denotes the end of the section.

Return Value

An [XCData](#) object.

Remarks

XML CDATA sections contain raw text that should be included, but not parsed, with the XML that contains it. A XML CDATA section can contain any text. This includes reserved XML characters. The XML CDATA section ends with the sequence "]]>". This implies the following points:

- You cannot use an embedded expression in an XML CDATA literal because the embedded expression delimiters are valid XML CDATA content.
- XML CDATA sections cannot be nested, because `content` cannot contain the value "]]>".

You can assign an XML CDATA literal to a variable, or include it in an XML element literal.

NOTE

An XML literal can span multiple lines but does not use line continuation characters. This enables you to copy content from an XML document and paste it directly into a Visual Basic program.

The Visual Basic compiler converts the XML CDATA literal to a call to the [XCData](#) constructor.

Example

The following example creates a CDATA section that contains the text "Can contain literal <XML> tags".

```
Dim cdata As XCData = <![CDATA[Can contain literal <XML> tags]]>
```

See also

- [XCDATA](#)
- [XML Element Literal](#)
- [XML Literals](#)
- [Creating XML in Visual Basic](#)

XML Comment Literal (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

A literal representing an [XComment](#) object.

Syntax

```
<!-- content -->
```

Parts

TERM	DEFINITION
<!--	Required. Denotes the start of the XML comment.
content	Required. Text to appear in the XML comment. Cannot contain a series of two hyphens (--) or end with a hyphen adjacent to the closing tag.
-->	Required. Denotes the end of the XML comment.

Return Value

An [XComment](#) object.

Remarks

XML comment literals do not contain document content; they contain information about the document. The XML comment section ends with the sequence "-->". This implies the following points:

- You cannot use an embedded expression in an XML comment literal because the embedded expression delimiters are valid XML comment content.
- XML comment sections cannot be nested, because `content` cannot contain the value "-->".

You can assign an XML comment literal to a variable, or you can include it in an XML element literal.

NOTE

An XML literal can span multiple lines without using line continuation characters. This feature enables you to copy content from an XML document and paste it directly into a Visual Basic program.

The Visual Basic compiler converts the XML comment literal to a call to the [XComment](#) constructor.

Example

The following example creates an XML comment that contains the text "This is a comment".

```
Dim com As XComment = <!-- This is a comment -->
```

See also

- [XComment](#)
- [XML Element Literal](#)
- [XML Literals](#)
- [Creating XML in Visual Basic](#)

XML Processing Instruction Literal (Visual Basic)

8/22/2019 • 2 minutes to read • [Edit Online](#)

A literal representing an [XProcessingInstruction](#) object.

Syntax

```
<?piName [ = piData ] ?>
```

Parts

`<?`

Required. Denotes the start of the XML processing instruction literal.

`piName`

Required. Name indicating which application the processing instruction targets. Cannot begin with "xml" or "XML".

`piData`

Optional. String indicating how the application targeted by `piName` should process the XML document.

`?>`

Required. Denotes the end of the processing instruction.

Return Value

An [XProcessingInstruction](#) object.

Remarks

XML processing instruction literals indicate how applications should process an XML document. When an application loads an XML document, the application can check the XML processing instructions to determine how to process the document. The application interprets the meaning of `piName` and `piData`.

The XML document literal uses syntax that is similar to that of the XML processing instruction. For more information, see [XML Document Literal](#).

NOTE

The `piName` element cannot begin with the strings "xml" or "XML", because the XML 1.0 specification reserves those identifiers.

You can assign an XML processing instruction literal to a variable or include it in an XML document literal.

NOTE

An XML literal can span multiple lines without needing line continuation characters. This enables you to copy content from an XML document and paste it directly into a Visual Basic program.

The Visual Basic compiler converts the XML processing instruction literal to a call to the [XProcessingInstruction](#) constructor.

Example

The following example creates a processing instruction identifying a style-sheet for an XML document.

```
Dim pi As XProcessingInstruction =
    <?xml-stylesheet type="text/xsl" href="show_book.xsl"?>
```

See also

- [XProcessingInstruction](#)
- [XML Document Literal](#)
- [XML Literals](#)
- [Creating XML in Visual Basic](#)

Error Messages (Visual Basic)

4/9/2019 • 2 minutes to read • [Edit Online](#)

When you write, compile, or run a Visual Basic application, the following types of errors can occur:

1. Design-time errors, which occur when you write an application in Visual Studio.
2. Compile-time errors, which occur when you compile an application in Visual Studio or at a command prompt.
3. Run-time errors, which occur when you run an application in Visual Studio or as a stand-alone executable file.

For information about how to troubleshoot a specific error, see [Additional Resources for Visual Basic Programmers](#).

Run Time Errors

If a Visual Basic application tries to perform an action that the system can't execute, a run-time error occurs, and Visual Basic throws an `Exception` object. Visual Basic can generate custom errors of any data type, including `Exception` objects, by using the `Throw` statement. An application can identify the error by displaying the error number and message of a caught exception. If an error isn't caught, the application ends.

The code can trap and examine run-time errors. If you enclose the code that produces the error in a `Try` block, you can catch any thrown error within a matching `Catch` block. For information about how to trap errors at run time and respond to them in your code, see [Try...Catch...Finally Statement](#).

Compile Time Errors

If the Visual Basic compiler encounters a problem in the code, a compile-time error occurs. In the Code Editor, you can easily identify which line of code caused the error because a wavy line appears under that line of code. The error message appears if you either point to the wavy underline or open the **Error List**, which also shows other messages.

If an identifier has a wavy underline and a short underline appears under the rightmost character, you can generate a stub for the class, constructor, method, property, field or enum. For more information, see [Generate From Usage](#).

By resolving warnings from the Visual Basic compiler, you might be able to write code that runs faster and has fewer bugs. These warnings identify code that may cause errors when the application is run. For example, the compiler warns you if you try to invoke a member of an unassigned object variable, return from a function without setting the return value, or execute a `Try` block with errors in the logic to catch exceptions. For more information about warnings, including how to turn them on and off, see [Configuring Warnings in Visual Basic](#).

'#ElseIf' must be preceded by a matching '#If' or '#ElseIf'

4/9/2019 • 2 minutes to read • [Edit Online](#)

#ElseIf is a conditional compilation directive. An #ElseIf clause must be preceded by a matching #If or #ElseIf clause.

Error ID: BC30014

To correct this error

1. Check that a preceding #If or #ElseIf has not been separated from this #ElseIf by an intervening conditional compilation block or an incorrectly placed #End If .
2. If the #ElseIf is preceded by a #Else directive, either remove the #Else or change it to an #ElseIf .
3. If everything else is in order, add an #If directive to the beginning of the conditional compilation block.

See also

- [#If...Then...#Else Directives](#)

'#Region' and '#End Region' statements are not valid within method bodies/multiline lambdas

4/9/2019 • 2 minutes to read • [Edit Online](#)

The `#Region` block must be declared at a class, module, or namespace level. A collapsible region can include one or more procedures, but it cannot begin or end inside of a procedure.

Error ID: BC32025

To correct this error

1. Ensure that the preceding procedure is properly terminated with an `End Function` or `End Sub` statement.
2. Ensure that the `#Region` and `#End Region` directives are in the same code block.

See also

- [#Region Directive](#)

'<attribute>' cannot be applied because the format of the GUID '<number>' is not correct

4/9/2019 • 2 minutes to read • [Edit Online](#)

A `comclassAttribute` attribute block specifies a globally unique identifier (GUID) that does not conform to the proper format for a GUID. `COMClassAttribute` uses GUIDs to uniquely identify the class, the interface, and the creation event.

A GUID consists of 16 bytes, of which the first eight are numeric and the last eight are binary. It is generated by Microsoft utilities such as `uuidgen.exe` and is guaranteed to be unique in space and time.

Error ID: BC32500

To correct this error

1. Determine the correct GUID or GUIDs necessary to identify the COM object.
2. Ensure that the GUID strings presented to the `comclassAttribute` attribute block are copied correctly.

See also

- [Guid](#)
- [Attributes overview](#)

'<classname>' is not CLS-compliant because the interface '<interfacename>' it implements is not CLS-compliant

4/28/2019 • 2 minutes to read • [Edit Online](#)

A class or interface is marked as `<CLSCompliant(True)>` when it derives from or implements a type that is marked as `<CLSCompliant(False)>` or is not marked.

For a class or interface to be compliant with the [Language Independence and Language-Independent Components](#) (CLS), its entire inheritance hierarchy must be compliant. That means every type from which it inherits, directly or indirectly, must be compliant. Similarly, if a class implements one or more interfaces, they must all be compliant throughout their inheritance hierarchies.

When you apply the [`CLSCompliantAttribute`](#) to a programming element, you set the attribute's `isCompliant` parameter to either `True` or `False` to indicate compliance or noncompliance. There is no default for this parameter, and you must supply a value.

If you do not apply the [`CLSCompliantAttribute`](#) to an element, it is considered to be noncompliant.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC40029

To correct this error

- If you require CLS compliance, define this type within a different inheritance hierarchy or implementation scheme.
- If you require that this type remain within its current inheritance hierarchy or implementation scheme, remove the [`CLSCompliantAttribute`](#) from its definition or mark it as `<CLSCompliant(False)>`.

'<elementname>' is obsolete (Visual Basic Warning)

4/28/2019 • 2 minutes to read • [Edit Online](#)

A statement attempts to access a programming element which has been marked with the `ObsoleteAttribute` attribute and the directive to treat it as a warning.

You can mark any programming element as being no longer in use by applying `ObsoleteAttribute` to it. If you do this, you can set the attribute's `IsError` property to either `True` or `False`. If you set it to `True`, the compiler treats an attempt to use the element as an error. If you set it to `False`, or let it default to `False`, the compiler issues a warning if there is an attempt to use the element.

By default, this message is a warning, because the `IsError` property of `ObsoleteAttribute` is `False`. For more information about hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC40008

To correct this error

- Ensure that the source-code reference is spelling the element name correctly.

See also

- [Attributes overview](#)

'<eventname>' is an event, and cannot be called directly

4/9/2019 • 2 minutes to read • [Edit Online](#)

'<eventname>' is an event, and so cannot be called directly. Use a `RaiseEvent` statement to raise an event.

A procedure call specifies an event for the procedure name. An event handler is a procedure, but the event itself is a signaling device, which must be raised and handled.

Error ID: BC32022

To correct this error

1. Use a `RaiseEvent` statement to signal an event and invoke the procedure or procedures that handle it.

See also

- [RaiseEvent Statement](#)

'<expression>' cannot be used as a type constraint

4/28/2019 • 2 minutes to read • [Edit Online](#)

A constraint list includes an expression that does not represent a valid constraint on a type parameter.

A constraint list imposes requirements on the type argument passed to the type parameter. You can specify the following requirements in any combination:

- The type argument must implement one or more interfaces
- The type argument must inherit from at most one class
- The type argument must expose a parameterless constructor that the creating code can access (include the `New` constraint)

If you do not include any specific class or interface in the constraint list, you can impose a more general requirement by specifying one of the following:

- The type argument must be a value type (include the `Structure` constraint)
- The type argument must be a reference type (include the `Class` constraint)

You cannot specify both `Structure` and `Class` for the same type parameter, and you cannot specify either one more than once.

Error ID: BC32061

To correct this error

- Verify that the expression and its elements are spelled correctly.
- If the expression does not qualify for the preceding list of requirements, remove it from the constraint list.
- If the expression refers to an interface or class, verify that the compiler has access to that interface or class. You might need to qualify its name, and you might need to add a reference to your project. For more information, see "References to Projects" in [References to Declared Elements](#).

See also

- [Generic Types in Visual Basic](#)
- [Value Types and Reference Types](#)
- [References to Declared Elements](#)

'<functionname>' is not declared (Smart Device/Visual Basic Compiler Error)

4/28/2019 • 2 minutes to read • [Edit Online](#)

<`functionname`> is not declared. File I/O functionality is normally available in the `Microsoft.VisualBasic` namespace, but the targeted version of the .NET Compact Framework does not support it.

Error ID: BC30766

To correct this error

- Perform file operations with functions defined in the `System.IO` namespace.

See also

- [System.IO](#)
- [File Access with Visual Basic](#)

'<interfacename>.<membername>' is already implemented by the base class '<basedclassname>'. Re-implementation of <type> assumed

4/28/2019 • 2 minutes to read • [Edit Online](#)

A property, procedure, or event in a derived class uses an `Implements` clause specifying an interface member that is already implemented in the base class.

A derived class can reimplement an interface member that is implemented by its base class. This is not the same as overriding the base class implementation. For more information, see [Implements](#).

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC42015

To correct this error

- If you intend to reimplement the interface member, you do not need to take any action. Code in your derived class accesses the reimplemented member unless you use the `MyBase` keyword to access the base class implementation.
- If you do not intend to reimplement the interface member, remove the `Implements` clause from the property, procedure, or event declaration.

See also

- [Interfaces](#)

'<keyword>' is valid only within an instance method

4/28/2019 • 2 minutes to read • [Edit Online](#)

The `Me`, `MyClass`, and `MyBase` keywords refer to specific class instances. You cannot use them inside a shared `Function` or `Sub` procedure.

Error ID: BC30043

To correct this error

- Remove the keyword from the procedure, or remove the `Shared` keyword from the procedure declaration.

See also

- [Object Variable Assignment](#)
- [Me, My, MyBase, and MyClass](#)
- [Inheritance Basics](#)

'<membername>' cannot expose type '<typename>' outside the project through <containertype>'<containertypename>'

10/1/2019 • 2 minutes to read • [Edit Online](#)

A variable, procedure parameter, or function return is exposed outside its container, but it is declared as a type that must not be exposed outside the container.

The following skeleton code shows a situation that generates this error.

```
Private Class privateClass
End Class
Public Class mainClass
    Public exposedVar As New privateClass
End Class
```

A type that is declared `Protected`, `Friend`, `Protected Friend`, or `Private` is intended to have limited access outside its declaration context. Using it as the data type of a variable with less restricted access would defeat this purpose. In the preceding skeleton code, `exposedVar` is `Public` and would expose `privateClass` to code that should not have access to it.

Error ID: BC30909

To correct this error

- Change the access level of the variable, procedure parameter, or function return to be at least as restrictive as the access level of its data type.

See also

- [Access levels in Visual Basic](#)

'<membername>' is ambiguous across the inherited interfaces '<interfacename1>' and '<interfacename2>'

10/1/2019 • 2 minutes to read • [Edit Online](#)

The interface inherits two or more members with the same name from multiple interfaces.

Error ID: BC30685

To correct this error

- Cast the value to the base interface that you want to use; for example:

```
Interface Left
    Sub MySub()
End Interface

Interface Right
    Sub MySub()
End Interface

Interface LeftRight
    Inherits Left, Right
End Interface

Module test
    Sub Main()
        Dim x As LeftRight
        ' x.MySub() 'x is ambiguous.
        CType(x, Left).MySub() ' Cast to base type.
        CType(x, Right).MySub() ' Call the other base type.
    End Sub
End Module
```

See also

- [Interfaces](#)

<message> This error could also be due to mixing a file reference with a project reference to assembly '<assemblyname>'

4/9/2019 • 2 minutes to read • [Edit Online](#)

<message> This error could also be due to mixing a file reference with a project reference to assembly '<assemblyname>'. In this case, try replacing the file reference to '<assemblyfilename>' in project '<projectname1>' with a project reference to '<projectname2>'.

Code in your project accesses a member of another project, but the configuration of your solution does not allow the Visual Basic compiler to resolve the reference.

To access a type defined in another assembly, the Visual Basic compiler must have a reference to that assembly. This must be a single, unambiguous reference that does not cause circular references among projects.

Error ID: BC30971

To correct this error

1. Determine which project produces the best assembly for your project to reference. For this decision, you might use criteria such as ease of file access and frequency of updates.
2. In your project properties, add a reference to the project that contains the assembly that defines the type you are using.

See also

- [Managing references in a project](#)
- [References to Declared Elements](#)
- [Managing Project and Solution Properties](#)
- [Troubleshooting Broken References](#)

'<methodname>' has multiple definitions with identical signatures

4/28/2019 • 2 minutes to read • [Edit Online](#)

A `Function` or `Sub` procedure declaration uses the identical procedure name and argument list as a previous declaration. One possible cause is an attempt to overload the original procedure. Overloaded procedures must have different argument lists.

Error ID: BC30269

To correct this error

- Change the procedure name or the argument list, or remove the duplicate declaration.

See also

- [References to Declared Elements](#)
- [Considerations in Overloading Procedures](#)

'<name>' is ambiguous in the namespace '<namespacename>'

4/28/2019 • 2 minutes to read • [Edit Online](#)

You have provided a name that is ambiguous and therefore conflicts with another name. The Visual Basic compiler does not have any conflict resolution rules; you must disambiguate names yourself.

Error ID: BC30560

To correct this error

- Fully qualify the name.

See also

- [Namespaces in Visual Basic](#)
- [Namespace Statement](#)

'<name1>' is ambiguous, imported from the namespaces or types '<name2>'

4/9/2019 • 2 minutes to read • [Edit Online](#)

You have provided a name that is ambiguous and therefore conflicts with another name. The Visual Basic compiler does not have any conflict resolution rules; you must disambiguate names yourself.

Error ID: BC30561

To correct this error

1. Disambiguate the name by removing namespace imports.
2. Fully qualify the name.

See also

- [Imports Statement \(.NET Namespace and Type\)](#)
- [Namespaces in Visual Basic](#)
- [Namespace Statement](#)

<proceduresignature1> is not CLS-compliant because it overloads <proceduresignature2> which differs from it only by array of array parameter types or by the rank of the array parameter types

10/10/2019 • 2 minutes to read • [Edit Online](#)

A procedure or property is marked as `<CLSCompliant(True)>` when it overrides another procedure or property and the only difference between their parameter lists is the nesting level of a jagged array or the rank of an array.

In the following declarations, the second and third declarations generate this error:

```
Overloads Sub ProcessArray(arrayParam() As Integer)
```

```
Overloads Sub ProcessArray(arrayParam()() As Integer)
```

```
Overloads Sub ProcessArray(arrayParam(,) As Integer)
```

The second declaration changes the original one-dimensional parameter `arrayParam` to an array of arrays. The third declaration changes `arrayParam` to a two-dimensional array (rank 2). While Visual Basic allows overloads to differ only by one of these changes, such overloading is not compliant with the [Language Independence and Language-Independent Components](#) (CLS).

When you apply the [CLSCompliantAttribute](#) to a programming element, you set the attribute's `isCompliant` parameter to either `True` or `False` to indicate compliance or noncompliance. There is no default for this parameter, and you must supply a value.

If you do not apply the [CLSCompliantAttribute](#) to an element, it is considered to be noncompliant.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC40035

To correct this error

- If you require CLS compliance, define your overloads to differ from each other in more ways than only the changes cited on this Help page.
- If you require that the overloads differ only by the changes cited on this Help page, remove the [CLSCompliantAttribute](#) from their definitions or mark them as `<CLSCompliant(False)>`.

See also

- [Procedure Overloading](#)
- [Overloads](#)

<type1>'<typename>' must implement '<membername>' for interface '<interfacename>'

10/1/2019 • 2 minutes to read • [Edit Online](#)

'<typename>' must implement '<membername>' for interface '<interfacename>'. Implementing property must have matching 'ReadOnly'/'WriteOnly' specifiers.

A class or structure claims to implement an interface but does not implement a procedure, property, or event defined by the interface. Every member of the interface must be implemented.

Error ID: BC30154

To correct this error

1. Declare a member with the same name and signature as defined in the interface. Be sure to include at least the `End Function`, `End Sub`, or `End Property` statement.
2. Add an `Implements` clause to the end of the `Function`, `Sub`, `Property`, or `Event` statement. For example:

```
Public Event ItHappened() Implements IBaseInterface.ItHappened
```
3. When implementing a property, make sure that `ReadOnly` or `WriteOnly` is used in the same way as in the interface definition.
4. When implementing a property, declare `Get` and `Set` procedures, as appropriate.

See also

- [Implements Statement](#)
- [Interfaces](#)

<type1>'<typename>' must implement ' '<methodname>' for interface '<interfacename>'

9/28/2019 • 2 minutes to read • [Edit Online](#)

A class or structure claims to implement an interface but does not implement a procedure defined by the interface. Every member of the interface must be implemented.

Error ID: BC30149

To correct this error

1. Declare a procedure with the same name and signature as defined in the interface. Be sure to include at least the `End Function` or `End Sub` statement.
2. Add an `Implements` clause to the end of the `Function` or `Sub` statement. For example:

```
Public Sub DoSomething() Implements IBaseInterface.DoSomething
```

See also

- [Implements Statement](#)
- [Interfaces](#)

'<typename>' cannot inherit from <type>
'<basetypename>' because it expands the access of
the base <type> outside the assembly

4/28/2019 • 2 minutes to read • [Edit Online](#)

A class or interface inherits from a base class or interface but has a less restrictive access level.

For example, a `Public` interface inherits from a `Friend` interface, or a `Protected` class inherits from a `Private` class. This exposes the base class or interface to access beyond the intended level.

Error ID: BC30910

To correct this error

- Change the access level of the derived class or interface to be at least as restrictive as that of the base class or interface.
-or-
- If you require the less restrictive access level, remove the `Inherits` statement. You cannot inherit from a more restricted base class or interface.

See also

- [Class Statement](#)
- [Interface Statement](#)
- [Inherits Statement](#)
- [Access levels in Visual Basic](#)

'<typename>' is a delegate type

4/28/2019 • 2 minutes to read • [Edit Online](#)

'<typename>' is a delegate type. Delegate construction permits only a single `AddressOf` expression as an argument list. Often an `AddressOf` expression can be used instead of a delegate construction.

A `New` clause creating an instance of a delegate class supplies an invalid argument list to the delegate constructor.

You can supply only a single `AddressOf` expression when creating a new delegate instance.

This error can result if you do not pass any arguments to the delegate constructor, if you pass more than one argument, or if you pass a single argument that is not a valid `AddressOf` expression.

Error ID: BC32008

To correct this error

- Use a single `AddressOf` expression in the argument list for the delegate class in the `New` clause.

See also

- [New Operator](#)
- [AddressOf Operator](#)
- [Delegates](#)
- [How to: Invoke a Delegate Method](#)

'<typename>' is a type and cannot be used as an expression

4/28/2019 • 2 minutes to read • [Edit Online](#)

A type name occurs where an expression is required. An expression must consist of some combination of variables, constants, literals, properties, and `Function` procedure calls.

Error ID: BC30108

To correct this error

- Remove the type name and construct the expression using valid elements.

See also

- [Operators and Expressions](#)

A double quote is not a valid comment token for delimited fields where EscapeQuote is set to True

4/28/2019 • 2 minutes to read • [Edit Online](#)

A quotation mark has been supplied as the delimiter for the `TextFieldParser`, but `EscapeQuotes` is set to `True`.

To correct this error

- Set `EscapeQuotes` to `False`.

See also

- [SetDelimiters](#)
- [Delimiters](#)
- [TextFieldParser](#)
- [How to: Read From Comma-Delimited Text Files](#)

A property or method call cannot include a reference to a private object, either as an argument or as a return value

4/28/2019 • 2 minutes to read • [Edit Online](#)

Among the possible causes of this error are:

- A client invoked a property or method of an out-of-process component and attempted to pass a reference to a private object as one of the arguments.
- An out-of-process component invoked a call-back method on its client and attempted to pass a reference to a private object.
- An out-of-process component attempted to pass a reference to a private object as an argument of an event it was raising.
- A client attempted to assign a private object reference to a `ByRef` argument of an event it was handling.

To correct this error

1. Remove the reference.

See also

- [Private](#)

A reference was created to embedded interop assembly '<assembly1>' because of an indirect reference to that assembly from assembly '<assembly2>'

10/17/2019 • 2 minutes to read • [Edit Online](#)

A reference was created to embedded interop assembly '<assembly1>' because of an indirect reference to that assembly from assembly '<assembly2>'. Consider changing the 'Embed Interop Types' property on either assembly.

You have added a reference to an assembly (assembly1) that has the `Embed Interop Types` property set to `True`. This instructs the compiler to embed interop type information from that assembly. However, the compiler cannot embed interop type information from that assembly because another assembly that you have referenced (assembly2) also references that assembly (assembly1) and has the `Embed Interop Types` property set to `False`.

NOTE

Setting the `Embed Interop Types` property on an assembly reference to `True` is equivalent to referencing the assembly by using the `/link` option for the command-line compiler.

Error ID: BC40059

To address this warning

- To embed interop type information for both assemblies, set the `Embed Interop Types` property on all references to assembly1 to `True`.
- To remove the warning, you can set the `Embed Interop Types` property of assembly1 to `False`. In this case, interop type information is provided by a primary interop assembly (PIA).

See also

- [-link \(Visual Basic\)](#)
- [Interoperating with Unmanaged Code](#)

A startup form has not been specified

4/9/2019 • 2 minutes to read • [Edit Online](#)

The application uses the [WindowsFormsApplicationBase](#) class but does not specify the startup form.

This can occur if the **Enable application framework** check box is selected in the project designer but the **Startup form** is not specified. For more information, see [Application Page, Project Designer \(Visual Basic\)](#).

To correct this error

1. Specify a startup object for the application.

For more information, see [Application Page, Project Designer \(Visual Basic\)](#).

2. Override the [OnCreateMainForm](#) method to set the [MainForm](#) property to the startup form.

See also

- [WindowsFormsApplicationBase](#)
- [OnCreateMainForm](#)
- [MainForm](#)
- [Overview of the Visual Basic Application Model](#)

Access of shared member through an instance; qualifying expression will not be evaluated

10/16/2019 • 2 minutes to read • [Edit Online](#)

An instance variable of a class or structure is used to access a `Shared` variable, property, procedure, or event defined in that class or structure. This warning can also occur if an instance variable is used to access an implicitly shared member of a class or structure, such as a constant or enumeration, or a nested class or structure.

The purpose of sharing a member is to create only a single copy of that member and make that single copy available to every instance of the class or structure in which it is declared. It is consistent with this purpose to access a `Shared` member through the name of its class or structure, rather than through a variable that holds an individual instance of that class or structure.

Accessing a `Shared` member through an instance variable can make your code more difficult to understand by obscuring the fact that the member is `Shared`. Furthermore, if such access is part of an expression that performs other actions, such as a `Function` procedure that returns an instance of the shared member, Visual Basic bypasses the expression and any other actions it would otherwise perform.

For more information and an example, see [Shared](#).

By default, this message is a warning. For more information about hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC42025

To correct this error

Use the name of the class or structure that defines the `Shared` member to access it, as shown in the following example:

```
Public Class TestClass
    Public Shared Sub SayHello()
        MsgBox("Hello")
    End Sub
End Class

Module Program
    Public Sub Main()
        ' Access a shared method through an instance variable.
        ' This generates a warning.
        Dim tc As New TestClass()
        tc.SayHello()

        ' Access a shared method by using the class name.
        ' This does not generate a warning.
        TestClass.SayHello()
    End Sub
End Module
```

NOTE

Be alert for the effects of scope when two programming elements have the same name. In the previous example, if you declare an instance by using `Dim testClass as testClass = Nothing`, the compiler treats a call to `testClass.sayHello()` as an access of the method through the class name, and no warning occurs.

See also

- [Shared](#)
- [Scope in Visual Basic](#)

'AddressOf' operand must be the name of a method (without parentheses)

10/16/2019 • 2 minutes to read • [Edit Online](#)

The `AddressOf` operator creates a procedure delegate instance that references a specific procedure. The syntax is as follows.

`AddressOf` `procedurename`

You inserted parentheses around the argument following `AddressOf`, where none are needed.

Error ID: BC30577

To correct this error

1. Remove the parentheses around the argument following `AddressOf`.
2. Make sure the argument is a method name.

See also

- [AddressOf Operator](#)
- [Delegates](#)

An unexpected error has occurred because an operating system resource required for single instance startup cannot be acquired

4/28/2019 • 2 minutes to read • [Edit Online](#)

The application could not acquire a necessary operating system resource. Some of the possible causes for this problem are:

- The application does not have permissions to create named operating-system objects.
- The common language runtime does not have permissions to create memory-mapped files.
- The application needs to access an operating-system object, but another process is using it.

To correct this error

1. Check that the application has sufficient permissions to create named operating-system objects.
2. Check that the common language runtime has sufficient permissions to create memory-mapped files.
3. Restart the computer to clear any process that may be using the resource needed to connect to the original instance application.
4. Note the circumstances under which the error occurred, and call Microsoft Product Support Services

See also

- [Application Page, Project Designer \(Visual Basic\)](#)
- [Debugger Basics](#)
- [Talk to Us](#)

Anonymous type member name can be inferred only from a simple or qualified name with no arguments

10/16/2019 • 2 minutes to read • [Edit Online](#)

You cannot infer an anonymous type member name from a complex expression.

```
Dim numbers() As Integer = {1, 2, 3, 4, 5}  
' Not valid.  
' Dim instanceName1 = New With {numbers(3)}
```

For more information about sources from which anonymous types can and cannot infer member names and types, see [How to: Infer Property Names and Types in Anonymous Type Declarations](#).

Error ID: BC36556

To correct this error

- Assign the expression to a member name, as shown in the following code:

```
Dim instanceName2 = New With {.number = numbers(3)}
```

See also

- [Anonymous Types](#)
- [How to: Infer Property Names and Types in Anonymous Type Declarations](#)

Argument not optional (Visual Basic)

4/9/2019 • 2 minutes to read • [Edit Online](#)

The number and types of arguments must match those expected. Either there is an incorrect number of arguments, or an omitted argument is not optional. An argument can only be omitted from a call to a user-defined procedure if it was declared `Optional` in the procedure definition.

To correct this error

1. Supply all necessary arguments.
2. Make sure omitted arguments are optional. If they are not, either supply the argument in the call, or declare the parameter `Optional` in the definition.

See also

- [Error Types](#)

Array bounds cannot appear in type specifiers

10/16/2019 • 2 minutes to read • [Edit Online](#)

Array sizes cannot be declared as part of a data type specifier.

Error ID: BC30638

To correct this error

- Specify the size of the array immediately following the variable name instead of placing the array size after the type, as shown in the following example.

```
Dim Array(8) As Integer
```

- Define an array and initialize it with the desired number of elements, as shown in the following example.

```
Dim Array2() As Integer = New Integer(8) {}
```

See also

- [Arrays](#)

Array declared as for loop control variable cannot be declared with an initial size

10/16/2019 • 2 minutes to read • [Edit Online](#)

A `For Each` loop uses an array as its *element* iteration variable but initializes that array.

The following statements show how this error can be generated.

```
Dim arrayList As New List(Of Integer())
For Each listElement() As Integer In arrayList
    For Each listElement(1) As Integer In arrayList
```

The first `For Each` statement is the correct way to access elements of `arrayList`. The second `For Each` statement generates this error.

Error ID: BC32039

To correct this error

- Remove the initialization from the declaration of the *element* iteration variable.

See also

- [For...Next Statement](#)
- [Arrays](#)
- [Collections](#)

Array subscript expression missing

10/16/2019 • 2 minutes to read • [Edit Online](#)

An array initialization leaves out one or more of the subscripts that define the array bounds. For example, the statement might contain the expression `myArray (5,5,,10)`, which leaves out the third subscript.

Error ID: BC30306

To correct this error

- Supply the missing subscript.

See also

- [Arrays](#)

Arrays declared as structure members cannot be declared with an initial size

10/16/2019 • 2 minutes to read • [Edit Online](#)

An array in a structure is declared with an initial size. You cannot initialize any structure element, and declaring an array size is one form of initialization.

Error ID: BC31043

Example

The following example generates BC31043:

```
Structure DemoStruct
    Public demoArray(9) As Integer
End Structure
```

To correct this error

1. Define the array in your structure as dynamic (no initial size).
2. If you require a certain size of array, you can redimension a dynamic array with a [ReDim Statement](#) when your code is running. The following example illustrates this:

```
Structure DemoStruct
    Public demoArray() As Integer
End Structure
Sub UseStruct()
    Dim struct As DemoStruct
    ReDim struct.demoArray(9)
    Struct.demoArray(2) = 777
End Sub
```

See also

- [Arrays](#)
- [How to: Declare a Structure](#)

'As Any' is not supported in 'Declare' statements

10/16/2019 • 2 minutes to read • [Edit Online](#)

The `Any` data type was used with `Declare` statements in Visual Basic 6.0 and earlier versions to permit the use of arguments that could contain any type of data. Visual Basic supports overloading, however, and so makes the `Any` data type obsolete.

Error ID: BC30828

To correct this error

1. Declare parameters of the specific type you want to use; for example.

```
Declare Function GetUserName Lib "advapi32.dll" Alias "GetUserNameA" (
    ByVal lpBuffer As String,
    ByRef nSize As Integer) As Integer
```

2. Use the `MarshalAsAttribute` attribute to specify `As Any` when `Void*` is expected by the procedure being called.

```
Declare Sub SetData Lib "..\LIB\UnmgdLib.dll" (
    ByVal x As Short,
    <System.Runtime.InteropServices.MarshalAsAttribute(
        System.Runtime.InteropServices.UnmanagedType.AsAny)>
    ByVal o As Object)
```

See also

- [MarshalAsAttribute](#)
- [Walkthrough: Calling Windows APIs](#)
- [Declare Statement](#)
- [Creating Prototypes in Managed Code](#)

Attribute '<attributename>' cannot be applied multiple times

4/9/2019 • 2 minutes to read • [Edit Online](#)

The attribute can only be applied once. The `AttributeUsage` attribute determines whether an attribute can be applied more than once.

Error ID: BC30663

To correct this error

1. Make sure the attribute is only applied once.
2. If you are using custom attributes you developed, consider changing their `AttributeUsage` attribute to allow multiple attribute usage, as with the following example.

```
<AttributeUsage(AllowMultiple := True)>
```

See also

- [AttributeUsageAttribute](#)
- [Creating Custom Attributes](#)
- [AttributeUsage](#)

Automation error

4/9/2019 • 2 minutes to read • [Edit Online](#)

An error occurred while executing a method or getting or setting a property of an object variable. The error was reported by the application that created the object.

To correct this error

1. Check the properties of the `Err` object to determine the source and nature of the error.
2. Use the `On Error Resume Next` statement immediately before the accessing statement, and then check for errors immediately after the accessing statement.

See also

- [Error Types](#)
- [Talk to Us](#)

Bad checksum value, non hex digits or odd number of hex digits

4/9/2019 • 2 minutes to read • [Edit Online](#)

A checksum value contains invalid hexadecimal digits or has an odd number of digits.

When ASP.NET generates a Visual Basic source file (extension .vb), it calculates a checksum and places it in a hidden source file identified by `#externalchecksum`. It is possible for a user generating a .vb file to do this also, but this process is best left to internal use.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC42033

To correct this error

1. If ASP.NET is generating the Visual Basic source file, restart the project build.
2. If this warning persists after restarting, reinstall ASP.NET and try the build again.
3. If the warning still persists, or if you are not using ASP.NET, gather information about the circumstances and notify Microsoft Product Support Services.

See also

- [ASP.NET Overview](#)
- [Talk to Us](#)

Bad DLL calling convention

4/9/2019 • 2 minutes to read • [Edit Online](#)

Arguments passed to a dynamic-link library (DLL) must exactly match those expected by the routine. Calling conventions deal with number, type, and order of arguments. Your program may be calling a routine in a DLL that is being passed the wrong type or number of arguments.

To correct this error

1. Make sure all argument types agree with those specified in the declaration of the routine that you are calling.
2. Make sure you are passing the same number of arguments indicated in the declaration of the routine that you are calling.
3. If the DLL routine expects arguments by value, make sure `ByVal` is specified for those arguments in the declaration for the routine.

See also

- [Error Types](#)
- [Call Statement](#)
- [Declare Statement](#)

Bad file mode

4/28/2019 • 2 minutes to read • [Edit Online](#)

Statements used in manipulating file contents must be appropriate to the mode in which the file was opened.

Possible causes include:

- A `FilePutObject` or `FileGetObject` statement specifies a sequential file.
- A `Print` statement specifies a file opened for an access mode other than `Output` or `Append`.
- An `Input` statement specifies a file opened for an access mode other than `Input`.
- An attempt to write to a read-only file.

To correct this error

- Make sure `FilePutObject` and `FileGetObject` are only referring to files open for `Random` or `Binary` access.
- Make sure `Print` specifies a file opened for either `Output` or `Append` access mode. If not, use a different statement to place data in the file, or reopen the file in an appropriate mode.
- Make sure `Input` specifies a file opened for `Input`. If not, use a different statement to place data in the file or reopen the file in an appropriate mode.
- If you are writing to a read-only file, change the read/write status of the file or do not try to write to it.
- Use the functionality available in the `My.Computer.FileSystem` object.

See also

- [FileSystem](#)
- [Troubleshooting: Reading from and Writing to Text Files](#)

Bad file name or number

4/28/2019 • 2 minutes to read • [Edit Online](#)

An error occurred while trying to access the specified file. Among the possible causes for this error are:

- A statement refers to a file with a file name or number that was not specified in the `FileOpen` statement or that was specified in a `FileOpen` statement but was subsequently closed.
- A statement refers to a file with a number that is out of the range of file numbers.
- A statement refers to a file name or number that is not valid.

To correct this error

1. Make sure the file name is specified in a `FileOpen` statement. Note that if you invoked the `FileClose` statement without arguments, you may have inadvertently closed all open files.
2. If your code is generating file numbers algorithmically, make sure the numbers are valid.
3. Check the file names to make sure they conform to operating system conventions.

See also

- [FileOpen](#)
- [Visual Basic Naming Conventions](#)

Bad record length

4/28/2019 • 2 minutes to read • [Edit Online](#)

Among the possible causes of this error are:

- The length of a record variable specified in a `FileGet`, `FileGetObject`, `FilePut` or `FilePutObject` statement differs from the length specified in the corresponding `FileOpen` statement.
- The variable in a `FilePut` or `FilePutObject` statement is or includes a variable-length string.
- The variable in a `FilePut` or `FilePutObject` is or includes a `Variant` type.

To correct this error

1. Make sure the sum of the sizes of fixed-length variables in the user-defined type defining the record variable's type is the same as the value stated in the `FileOpen` statement's `Len` clause.
2. If the variable in a `FilePut` or `FilePutObject` statement is or includes a variable-length string, make sure the variable-length string is at least 2 characters shorter than the record length specified in the `Len` clause of the `FileOpen` statement.
3. If the variable in a `FilePut` or `FilePutObject` is or includes a `Variant` make sure the variable-length string is at least 4 bytes shorter than the record length specified in the `Len` clause of the `FileOpen` statement.

See also

- [FileGet](#)
- [FileGetObject](#)
- [FilePut](#)
- [FilePutObject](#)

Because this call is not awaited, the current method continues to run before the call is completed

10/17/2019 • 6 minutes to read • [Edit Online](#)

Because this call is not awaited, execution of the current method continues before the call is completed. Consider applying the 'Await' operator to the result of the call.

The current method calls an async method that returns a [Task](#) or a [Task<TResult>](#) and doesn't apply the [Await](#) operator to the result. The call to the async method starts an asynchronous task. However, because no [Await](#) operator is applied, the program continues without waiting for the task to complete. In most cases, that behavior isn't expected. Usually other aspects of the calling method depend on the results of the call or, minimally, the called method is expected to complete before you return from the method that contains the call.

An equally important issue is what happens with exceptions that are raised in the called async method. An exception that's raised in a method that returns a [Task](#) or [Task<TResult>](#) is stored in the returned task. If you don't await the task or explicitly check for exceptions, the exception is lost. If you await the task, its exception is rethrown.

As a best practice, you should always await the call.

By default, this message is a warning. For more information about hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC42358

To address this warning

- You should consider suppressing the warning only if you're sure that you don't want to wait for the asynchronous call to complete and that the called method won't raise any exceptions. In that case, you can suppress the warning by assigning the task result of the call to a variable.

The following example shows how to cause the warning, how to suppress it, and how to await the call.

```

Async Function CallingMethodAsync() As Task

    ResultsTextBox.Text &= vbCrLf & " Entering calling method."

    ' Variable delay is used to slow down the called method so that you
    ' can distinguish between awaiting and not awaiting in the program's output.
    ' You can adjust the value to produce the output that this topic shows
    ' after the code.
    Dim delay = 5000

    ' Call #1.
    ' Call an async method. Because you don't await it, its completion isn't
    ' coordinated with the current method, CallingMethodAsync.
    ' The following line causes the warning.
    CalledMethodAsync(delay)

    ' Call #2.
    ' To suppress the warning without awaiting, you can assign the
    ' returned task to a variable. The assignment doesn't change how
    ' the program runs. However, the recommended practice is always to
    ' await a call to an async method.
    ' Replace Call #1 with the following line.
    'Task delayTask = CalledMethodAsync(delay)

    ' Call #3
    ' To contrast with an awaited call, replace the unawaited call
    ' (Call #1 or Call #2) with the following awaited call. The best
    ' practice is to await the call.

    'Await CalledMethodAsync(delay)

    ' If the call to CalledMethodAsync isn't awaited, CallingMethodAsync
    ' continues to run and, in this example, finishes its work and returns
    ' to its caller.
    ResultsTextBox.Text &= vbCrL & " Returning from calling method."
End Function

Async Function CalledMethodAsync(howLong As Integer) As Task

    ResultsTextBox.Text &= vbCrL & " Entering called method, starting and awaiting Task.Delay."
    ' Slow the process down a little so you can distinguish between awaiting
    ' and not awaiting. Adjust the value for howLong if necessary.
    Await Task.Delay(howLong)
    ResultsTextBox.Text &= vbCrL & " Task.Delay is finished--returning from called method."
End Function

```

In the example, if you choose Call #1 or Call #2, the unawaited async method (`CalledMethodAsync`) finishes after both its caller (`CallingMethodAsync`) and the caller's caller (`StartButton_Click`) are complete. The last line in the following output shows you when the called method finishes. Entry to and exit from the event handler that calls `CallingMethodAsync` in the full example are marked in the output.

```

Entering the Click event handler.
Entering calling method.
    Entering called method, starting and awaiting Task.Delay.
    Returning from calling method.
Exiting the Click event handler.
    Task.Delay is finished--returning from called method.

```

Example

The following Windows Presentation Foundation (WPF) application contains the methods from the previous example. The following steps set up the application.

1. Create a WPF application, and name it `AsyncWarning`.
 2. In the Visual Studio Code Editor, choose the **MainWindow.xaml** tab.
- If the tab isn't visible, open the shortcut menu for `MainWindow.xaml` in **Solution Explorer**, and then choose **View Code**.
3. Replace the code in the **XAML** view of `MainWindow.xaml` with the following code.

```
<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button x:Name="StartButton" Content="Start" HorizontalAlignment="Left" Margin="214,28,0,0"
            VerticalAlignment="Top" Width="75" HorizontalContentAlignment="Center" FontWeight="Bold"
            FontFamily="Aharoni" Click="StartButton_Click" />
        <TextBox x:Name="ResultsTextBox" Margin="0,80,0,0" TextWrapping="Wrap" FontFamily="Lucida
            Console"/>
    </Grid>
</Window>
```

A simple window that contains a button and a text box appears in the **Design** view of `MainWindow.xaml`.

For more information about the XAML Designer, see [Creating a UI by using XAML Designer](#). For information about how to build your own simple UI, see the "To create a WPF application" and "To design a simple WPF MainWindow" sections of [Walkthrough: Accessing the Web by Using Async and Await](#).

4. Replace the code in `MainWindow.xaml.vb` with the following code.

```
Class MainWindow

    Private Async Sub StartButton_Click(sender As Object, e As RoutedEventArgs)

        ResultsTextBox.Text &= vbCrLf & "Entering the Click event handler."
        Await CallingMethodAsync()
        ResultsTextBox.Text &= vbCrLf & "Exiting the Click event handler."
    End Sub

    Async Function CallingMethodAsync() As Task

        ResultsTextBox.Text &= vbCrLf & " Entering calling method."

        ' Variable delay is used to slow down the called method so that you
        ' can distinguish between awaiting and not awaiting in the program's output.
        ' You can adjust the value to produce the output that this topic shows
        ' after the code.
        Dim delay = 5000

        ' Call #1.
        ' Call an async method. Because you don't await it, its completion isn't
        ' coordinated with the current method, CallingMethodAsync.
        ' The following line causes the warning.
        CalledMethodAsync(delay)

        ' Call #2.
        ' To suppress the warning without awaiting, you can assign the
        ' returned task to a variable. The assignment doesn't change how
        ' the program runs. However, the recommended practice is always to
        ' await a call to an async method.

        ' Replace Call #1 with the following line.
        'Task delayTask = CalledMethodAsync(delay)

        ' Call #3
```

```

' To contrast with an awaited call, replace the unawaited call
' (Call #1 or Call #2) with the following awaited call. The best
' practice is to await the call.

'Await CalledMethodAsync(delay)

    ' If the call to CalledMethodAsync isn't awaited, CallingMethodAsync
    ' continues to run and, in this example, finishes its work and returns
    ' to its caller.
    ResultsTextBox.Text &= vbCrLf & " Returning from calling method."
End Function

Async Function CalledMethodAsync(howLong As Integer) As Task

    ResultsTextBox.Text &= vbCrLf & " Entering called method, starting and awaiting Task.Delay."
    ' Slow the process down a little so you can distinguish between awaiting
    ' and not awaiting. Adjust the value for howLong if necessary.
    Await Task.Delay(howLong)
    ResultsTextBox.Text &= vbCrLf & " Task.Delay is finished--returning from called method."
End Function

End Class

' Output

' Entering the Click event handler.
'   Entering calling method.
'     Entering called method, starting and awaiting Task.Delay.
'     Returning from calling method.
'   Exiting the Click event handler.
'     Task.Delay is finished--returning from called method.

' Output

' Entering the Click event handler.
'   Entering calling method.
'     Entering called method, starting and awaiting Task.Delay.
'     Task.Delay is finished--returning from called method.
'   Returning from calling method.
'   Exiting the Click event handler.

```

5. Choose the F5 key to run the program, and then choose the **Start** button.

The expected output appears at the end of the code.

See also

- [Await Operator](#)
- [Asynchronous Programming with Async and Await](#)

Cannot convert anonymous type to expression tree because it contains a field that is used in the initialization of another field

10/1/2019 • 2 minutes to read • [Edit Online](#)

The compiler does not accept conversion of an anonymous to an expression tree when one property of the anonymous type is used to initialize another property of the anonymous type. For example, in the following code, `Prop1` is declared in the initialization list and then used as the initial value for `Prop2`.

```
Module M2

Sub ExpressionExample(Of T)(ByVal x As Expressions.Expression(Of Func(Of T)))
End Sub

Sub Main()
    ' The following line causes the error.
    ' ExpressionExample(Function() New With {.Prop1 = 2, .Prop2 = .Prop1})

    End Sub
End Module
```

Error ID: BC36548

To correct this error

- Assign the initial value for `Prop1` to a local variable. Assign that variable to both `Prop1` and `Prop2`, as shown in the following code.

```
Sub Main()

    Dim temp = 2
    ExpressionExample(Function() New With {.Prop1 = temp, .Prop2 = temp})

    End Sub
```

See also

- [Anonymous Types \(Visual Basic\)](#)
- [Expression Trees \(Visual Basic\)](#)
- [How to: Use Expression Trees to Build Dynamic Queries \(Visual Basic\)](#)

Cannot create ActiveX Component

4/28/2019 • 2 minutes to read • [Edit Online](#)

You tried to place an ActiveX control on a form at design time or add a form to a project with an ActiveX control on it, but the associated information in the registry could not be found.

To correct this error

- The information in the registry may have been deleted or corrupted. Reinstall the ActiveX control or contact the control vendor.

See also

- [Error Types](#)
- [Talk to Us](#)

Cannot refer to '<name>' because it is a member of the value-typed field '<name>' of class '<classname>' which has 'System.MarshalByRefObject' as a base class

4/9/2019 • 2 minutes to read • [Edit Online](#)

The `System.MarshalByRefObject` class enables applications that support remote access to objects across application domain boundaries. Types must inherit from the `MarshalByRejectObject` class when the type is used across application domain boundaries. The state of the object must not be copied because the members of the object are not usable outside the application domain in which they were created.

Error ID: BC30310

To correct this error

1. Check the reference to make sure the member being referred to is valid.
2. Explicitly qualify the member with the `Me` keyword.

See also

- [MarshalByRefObject](#)
- [Dim Statement](#)

Cannot refer to an instance member of a class from within a shared method or shared member initializer without an explicit instance of the class

10/10/2019 • 2 minutes to read • [Edit Online](#)

You have tried to refer to a non-shared member of a class from within a shared procedure. The following example demonstrates such a situation:

```
Class Sample
    Public x as Integer
    Public Shared Sub SetX()
        x = 10
    End Sub
End Class
```

In the preceding example, the assignment statement `x = 10` generates this error message. This is because a shared procedure is attempting to access an instance variable.

The variable `x` is an instance member because it is not declared as `Shared`. Each instance of class `Sample` contains its own individual variable `x`. When one instance sets or changes the value of `x`, it does not affect the value of `x` in any other instance.

However, the procedure `SetX` is `Shared` among all instances of class `Sample`. This means it is not associated with any one instance of the class, but rather operates independently of individual instances. Because it has no connection with a particular instance, `setx` cannot access an instance variable. It must operate only on `Shared` variables. When `setx` sets or changes the value of a shared variable, that new value is available to all instances of the class.

Error ID: BC30369

To correct this error

1. Decide whether you want the member to be shared among all instances of the class, or kept individual for each instance.
2. If you want a single copy of the member to be shared among all instances, add the `Shared` keyword to the member declaration. Retain the `Shared` keyword in the procedure declaration.
3. If you want each instance to have its own individual copy of the member, do not specify `Shared` for the member declaration. Remove the `Shared` keyword from the procedure declaration.

See also

- [Shared](#)

Can't create necessary temporary file

4/9/2019 • 2 minutes to read • [Edit Online](#)

Either the drive is full that contains the directory specified by the TEMP environment variable, or the TEMP environment variable specifies an invalid or read-only drive or directory.

To correct this error

1. Delete files from the drive, if full.
2. Specify a different drive in the TEMP environment variable.
3. Specify a valid drive for the TEMP environment variable.
4. Remove the read-only restriction from the currently specified drive or directory.

See also

- [Error Types](#)

Can't open '<filename>' for writing

4/9/2019 • 2 minutes to read • [Edit Online](#)

The specified file cannot be opened for writing, perhaps because it has already been opened.

Error ID: BC2012

To correct this error

1. Close the file and reopen it.
2. Check the file's permissions.

See also

- [WriteAllText](#)
- [WriteAllBytes](#)
- [Writing to Files](#)

Class '<classname>' cannot be found

4/9/2019 • 2 minutes to read • [Edit Online](#)

Class '<classname>' cannot be found. This condition is usually the result of a mismatched 'Microsoft.VisualBasic.dll'.

A defined member could not be located.

Error ID: BC31098

To correct this error

1. Compile the program again to see if the error recurs.
2. If the error recurs, save your work and restart Visual Studio.
3. If the error persists, reinstall Visual Basic.
4. If the error persists after reinstallation, notify Microsoft Product Support Services.

See also

- [Talk to Us](#)

Class does not support Automation or does not support expected interface

4/9/2019 • 2 minutes to read • [Edit Online](#)

Either the class you specified in the `GetObject` or `CreateObject` function call has not exposed a programmability interface, or you changed a project from .dll to .exe, or vice versa.

To correct this error

1. Check the documentation of the application that created the object for limitations on the use of automation with this class of object.
2. If you changed a project from .dll to .exe or vice versa, you must manually unregister the old .dll or .exe.

See also

- [Error Types](#)
- [Talk to Us](#)

'Class' statement must end with a matching 'End Class'

4/28/2019 • 2 minutes to read • [Edit Online](#)

`class` is used to initiate a `class` block; hence it can only appear at the beginning of the block, with a matching `End Class` statement ending the block. Either you have a redundant `class` statement, or you have not ended your `class` block with `End Class`.

Error ID: BC30481

To correct this error

- Locate and remove the unnecessary `class` statement.
- Conclude the `class` block with a matching `End Class`.

See also

- [End <keyword> Statement](#)
- [Class Statement](#)

Clipboard format is not valid

4/28/2019 • 2 minutes to read • [Edit Online](#)

The specified Clipboard format is incompatible with the method being executed. Among the possible causes for this error are:

- Using the Clipboard's `GetText` or `SetText` method with a Clipboard format other than `vbCFText` or `vbCFLink`.
- Using the Clipboard's `GetData` or `SetData` method with a Clipboard format other than `vbCFBitmap`, `vbCFDIB`, or `vbCFMetafile`.
- Using the `GetData` or `SetData` methods of a `DataObject` with a Clipboard format in the range reserved by Microsoft Windows for registered formats (&HC000-&HFFFF), when that Clipboard format has not been registered with Microsoft Windows.

To correct this error

- Remove the invalid format and specify a valid one.

See also

- [Clipboard: Adding Other Formats](#)

Constant expression not representable in type '`<typename>`'

4/9/2019 • 2 minutes to read • [Edit Online](#)

You are trying to evaluate a constant that will not fit into the target type, usually because it is overflowing the range.

Error ID: BC30439

To correct this error

1. Change the target type to one that can handle the constant.

See also

- [Constants Overview](#)
- [Constants and Enumerations](#)

Constants must be of an intrinsic or enumerated type, not a class, structure, type parameter, or array type

4/9/2019 • 2 minutes to read • [Edit Online](#)

You have attempted to declare a constant as a class, structure, or array type, or as a type parameter defined by a containing generic type.

Constants must be of an intrinsic type (`Boolean`, `Byte`, `Date`, `Decimal`, `Double`, `Integer`, `Long`, `Object`, `SByte`, `Short`, `Single`, `String`, `UInteger`, `ULong`, or `UShort`), or an `Enum` type based on one of the integral types.

Error ID: BC30424

To correct this error

1. Declare the constant as an intrinsic or `Enum` type.
2. A constant can also be a special value such as `True`, `False`, or `Nothing`. The compiler considers these predefined values to be of the appropriate intrinsic type.

See also

- [Constants and Enumerations](#)
- [Data Types](#)
- [Data Types](#)

Constructor '<name>' cannot call itself

4/9/2019 • 2 minutes to read • [Edit Online](#)

A `Sub New` procedure in a class or structure calls itself.

The purpose of a constructor is to initialize an instance of a class or structure when it is first created. A class or structure can have several constructors, provided they all have different parameter lists. A constructor is permitted to call another constructor to perform its functionality in addition to its own. But it is meaningless for a constructor to call itself, and in fact it would result in infinite recursion if permitted.

Error ID: BC30298

To correct this error

1. Check the parameter list of the constructor being called. It should be different from that of the constructor making the call.
2. If you do not intend to call a different constructor, remove the `Sub New` call entirely.

See also

- [Object Lifetime: How Objects Are Created and Destroyed](#)

Copying the value of 'ByRef' parameter '`<parametername>`' back to the matching argument narrows from type '`<typename1>`' to type '`<typename2>`'

4/28/2019 • 2 minutes to read • [Edit Online](#)

A procedure is called with an argument that widens to the corresponding parameter type, and the conversion from the parameter to the argument is narrowing.

When you define a class or structure, you can define one or more conversion operators to convert that class or structure type to other types. You can also define reverse conversion operators to convert those other types back to your class or structure type. When you use your class or structure type in a procedure call, Visual Basic can use these conversion operators to convert the type of an argument to the type of its corresponding parameter.

If you pass the argument [ByRef](#), Visual Basic sometimes copies the argument value into a local variable in the procedure instead of passing a reference. In such a case, when the procedure returns, Visual Basic must then copy the local variable value back into the argument in the calling code.

If a [ByRef](#) argument value is copied into the procedure and the argument and parameter are of the same type, no conversion is necessary. But if the types are different, Visual Basic must convert in both directions. If one of the types is your class or structure type, Visual Basic must convert it both to and from the other type. If one of these conversions is widening, the reverse conversion might be narrowing.

Error ID: BC32053

To correct this error

- If possible, use a calling argument of the same type as the procedure parameter, so Visual Basic does not need to do any conversion.
- If you need to call the procedure with an argument type different from the parameter type but do not need to return a value into the calling argument, define the parameter to be [ByVal](#) instead of [ByRef](#).
- If you need to return a value into the calling argument, define the reverse conversion operator as [Widening](#), if possible.

See also

- [Procedures](#)
- [Procedure Parameters and Arguments](#)
- [Passing Arguments by Value and by Reference](#)
- [Operator Procedures](#)
- [Operator Statement](#)
- [How to: Define an Operator](#)
- [How to: Define a Conversion Operator](#)
- [Type Conversions in Visual Basic](#)
- [Widening and Narrowing Conversions](#)

'Custom' modifier is not valid on events declared without explicit delegate types

4/9/2019 • 2 minutes to read • [Edit Online](#)

Unlike a non-custom event, a `Custom Event` declaration requires an `As` clause following the event name that explicitly specifies the delegate type for the event.

Non-custom events can be defined either with an `As` clause and an explicit delegate type, or with a parameter list immediately following the event name.

Error ID: BC31122

To correct this error

1. Define a delegate with the same parameter list as the custom event.

For example, if the `Custom Event` was defined by

`Custom Event Test(ByVal sender As Object, ByVal i As Integer)`, then the corresponding delegate would be the following.

```
Delegate Sub TestDelegate(ByVal sender As Object, ByVal i As Integer)
```

2. Replace the parameter list of the custom event with an `As` clause specifying the delegate type.

Continuing with the example, `Custom Event` declaration would be rewritten as follows.

```
Custom Event Test As TestDelegate
```

Example

This example declares a `Custom Event` and specifies the required `As` clause with a delegate type.

```
Delegate Sub TestDelegate(ByVal sender As Object, ByVal i As Integer)
Custom Event Test As TestDelegate
    AddHandler(ByVal value As TestDelegate)
        ' Code for adding an event handler goes here.
    End AddHandler

    RemoveHandler(ByVal value As TestDelegate)
        ' Code for removing an event handler goes here.
    End RemoveHandler

    RaiseEvent(ByVal sender As Object, ByVal i As Integer)
        ' Code for raising an event goes here.
    End RaiseEvent
End Event
```

See also

- [Event Statement](#)

- Delegate Statement
- Events

Data type(s) of the type parameter(s) cannot be inferred from these arguments

10/17/2019 • 2 minutes to read • [Edit Online](#)

Data type(s) of the type parameter(s) cannot be inferred from these arguments. Specifying the data type(s) explicitly might correct this error.

This error occurs when overload resolution has failed. It occurs as a subordinate message that states why a particular overload candidate has been eliminated. The error message explains that the compiler cannot use type inference to find data types for the type parameters.

NOTE

When specifying arguments is not an option (for example, for query operators in query expressions), the error message appears without the second sentence.

The following code demonstrates the error.

```
Module Module1

Sub Main()

    '' Not Valid.
    'OverloadedGenericMethod("Hello", "World")

End Sub

Sub OverloadedGenericMethod(Of T)(ByVal x As String,
                                ByVal y As InterfaceExample(Of T))
End Sub

Sub OverloadedGenericMethod(Of T, R)(ByVal x As T,
                                    ByVal y As InterfaceExample(Of R))
End Sub

End Module

Interface InterfaceExample(Of T)
End Interface
```

Error ID: BC36647 and BC36644

To correct this error

You may be able to specify a data type for the type parameter or parameters instead of relying on type inference.

See also

- [Relaxed Delegate Conversion](#)
- [Generic Procedures in Visual Basic](#)
- [Type Conversions in Visual Basic](#)

Declaration expected

4/28/2019 • 2 minutes to read • [Edit Online](#)

A nondeclarative statement, such as an assignment or loop statement, occurs outside any procedure. Only declarations are allowed outside procedures.

Alternatively, a programming element is declared without a declaration keyword such as `Dim` or `Const`.

Error ID: BC30188

To correct this error

- Move the nondeclarative statement to the body of a procedure.
- Begin the declaration with an appropriate declaration keyword.
- Ensure that a declaration keyword is not misspelled.

See also

- [Procedures](#)
- [Dim Statement](#)

Default property '<propertyname1>' conflicts with default property '<propertyname2>' in '<classname>' and so should be declared 'Shadows'

4/28/2019 • 2 minutes to read • [Edit Online](#)

A property is declared with the same name as a property defined in the base class. In this situation, the property in this class should shadow the base class property.

This message is a warning. `Shadows` is assumed by default. For more information about hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC40007

To correct this error

- Add the `Shadows` keyword to the declaration, or change the name of the property being declared.

See also

- [Shadows](#)
- [Shadowing in Visual Basic](#)

Default property access is ambiguous between the inherited interface members '`<defaultpropertyname>`' of interface '`<interfacename1>`' and '`<defaultpropertyname>`' of interface '`<interfacename2>`'

10/10/2019 • 2 minutes to read • [Edit Online](#)

An interface inherits from two interfaces, each of which declares a default property with the same name. The compiler cannot resolve an access to this default property without qualification. The following example illustrates this.

```
Public Interface Iface1
    Default Property prop(ByVal arg As Integer) As Integer
End Interface
Public Interface Iface2
    Default Property prop(ByVal arg As Integer) As Integer
End Interface
Public Interface Iface3
    Inherits Iface1, Iface2
End Interface
Public Class testClass
    Public Sub accessDefaultProperty()
        Dim testObj As Iface3
        Dim testInt As Integer = testObj(1)
    End Sub
End Class
```

When you specify `testObj(1)`, the compiler tries to resolve it to the default property. However, there are two possible default properties because of the inherited interfaces, so the compiler signals this error.

Error ID: BC30686

To correct this error

- Avoid inheriting any members with the same name. In the preceding example, if `testObj` does not need any of the members of, say, `Iface2`, then declare it as follows:

```
Dim testObj As Iface1
```

-or-

- Implement the inheriting interface in a class. Then you can implement each of the inherited properties with different names. However, only one of them can be the default property of the implementing class. The following example illustrates this.

```
Public Class useIface3
    Implements Iface3
    Default Public Property prop1(ByVal arg As Integer) As Integer Implements Iface1.prop
        ' Insert code to define Get and Set procedures for prop1.
    End Property
    Public Property prop2(ByVal arg As Integer) As Integer Implements Iface2.prop
        ' Insert code to define Get and Set procedures for prop2.
    End Property
End Class
```

See also

- [Interfaces](#)

Delegate class '<classname>' has no Invoke method, so an expression of this type cannot be the target of a method call

4/9/2019 • 2 minutes to read • [Edit Online](#)

A call to `Invoke` through a delegate has failed because `Invoke` is not implemented on the delegate class.

Error ID: BC30220

To correct this error

1. Ensure that an instance of the delegate class has been created with a `Dim` statement and that a procedure has been assigned to the delegate instance with the `AddressOf` operator.
2. Locate the code that implements the delegate class and make sure it implements the `Invoke` procedure.

See also

- [Delegates](#)
- [Delegate Statement](#)
- [AddressOf Operator](#)
- [Dim Statement](#)

Derived classes cannot raise base class events

4/28/2019 • 2 minutes to read • [Edit Online](#)

An event can be raised only from the declaration space in which it is declared. Therefore, a class cannot raise events from any other class, even one from which it is derived.

Error ID: BC30029

To correct this error

- Move the `Event` statement or the `RaiseEvent` statement so they are in the same class.

See also

- [Event Statement](#)
- [RaiseEvent Statement](#)

Device I/O error

4/28/2019 • 2 minutes to read • [Edit Online](#)

An input or output error occurred while your program was using a device such as a printer or disk drive.

To correct this error

- Make sure the device is operating properly, and then retry the operation.

See also

- [Error Types](#)

'Dir' function must first be called with a 'PathName' argument

4/9/2019 • 2 minutes to read • [Edit Online](#)

An initial call to the `Dir` function does not include the `PathName` argument. The first call to `Dir` must include a `PathName`, but subsequent calls to `Dir` do not need to include parameters to retrieve the next item.

To correct this error

1. Supply a `PathName` argument in the function call.

See also

- [Dir](#)

End of statement expected

4/9/2019 • 2 minutes to read • [Edit Online](#)

The statement is syntactically complete, but an additional programming element follows the element that completes the statement. A line terminator is required at the end of every statement.

A line terminator divides the characters of a Visual Basic source file into lines. Examples of line terminators are the Unicode carriage return character (&HD), the Unicode linefeed character (&HA), and the Unicode carriage return character followed by the Unicode linefeed character. For more information about line terminators, see the [Visual Basic Language Specification](#).

Error ID: BC30205

To correct this error

1. Check to see if two different statements have inadvertently been put on the same line.
2. Insert a line terminator after the element that completes the statement.

See also

- [How to: Break and Combine Statements in Code](#)
- [Statements](#)

Error creating assembly manifest: <error message>

9/13/2019 • 2 minutes to read • [Edit Online](#)

The Visual Basic compiler calls the Assembly Linker (Al.exe, also known as Alink) to generate an assembly with a manifest. The linker has reported an error in the pre-emission stage of creating the assembly.

This can occur if there are problems with the key file or the key container specified. To fully sign an assembly, you must provide a valid key file that contains information about the public and private keys. To delay sign an assembly, you must select the **Delay sign only** check box and provide a valid key file that contains information about the public key information. The private key is not necessary when an assembly is delay-signed. For more information, see [How to: Sign an Assembly with a Strong Name](#).

Error ID: BC30140

To correct this error

1. Examine the quoted error message and consult the topic [Al.exe](#) for error AL1019 further explanation and advice
2. If the error persists, gather information about the circumstances and notify Microsoft Product Support Services.

See also

- [How to: Sign an Assembly with a Strong Name](#)
- [Signing Page, Project Designer](#)
- [Al.exe](#)
- [Talk to Us](#)

Error creating Win32 resources: <error message>

4/9/2019 • 2 minutes to read • [Edit Online](#)

The Visual Basic compiler calls the Assembly Linker (Al.exe, also known as Alink) to generate an assembly with a manifest. The linker has reported an error creating an in-memory resource. This might be a problem with the environment, or your computer might be low on memory.

Error ID: BC30136

To correct this error

1. Examine the quoted error message and consult the topic [Al.exe](#) for further explanation and advice.
2. If the error persists, gather information about the circumstances and notify Microsoft Product Support Services.

See also

- [Al.exe](#)
- [Talk to Us](#)

Error in loading DLL (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

A dynamic-link library (DLL) is a library specified in the `Lib` clause of a `Declare` statement. Possible causes for this error include:

- The file is not DLL executable.
- The file is not a Microsoft Windows DLL.
- The DLL references another DLL that is not present.
- The DLL or referenced DLL is not in a directory specified in the path.

To correct this error

- If the file is a source-text file and therefore not DLL executable, it must be compiled and linked to a DLL-executable form.
- If the file is not a Microsoft Windows DLL, obtain the Microsoft Windows equivalent.
- If the DLL references another DLL that is not present, obtain the referenced DLL and make it available.
- If the DLL or referenced DLL is not in a directory specified by the path, move the DLL to a referenced directory.

See also

- [Declare Statement](#)

Error saving temporary Win32 resource file '': <error message>

4/9/2019 • 2 minutes to read • [Edit Online](#)

The Visual Basic compiler calls the Assembly Linker (Al.exe, also known as Alink) to generate an assembly with a manifest. The linker reported an error obtaining a file name for use in writing an in-memory resource.

Error ID: BC30137

To correct this error

1. Examine the quoted error message and consult the topic [Al.exe](#) for further explanation and advice.
2. If the error persists, gather information about the circumstances and notify Microsoft Product Support Services.

See also

- [Al.exe](#)
- [Talk to Us](#)

Errors occurred while compiling the XML schemas in the project

4/28/2019 • 2 minutes to read • [Edit Online](#)

Errors occurred while compiling the XML schemas in the project. Because of this, XML IntelliSense is not available.

There is an error in an XML Schema Definition (XSD) schema included in the project. This error occurs when you add an XSD schema (.xsd) file that conflicts with the existing XSD schema set for the project.

Error ID: BC36810

To correct this error

- Double-click the warning in the **Errors List** window. Visual Basic will take you to the location in the XSD file that is the source of the warning. Correct the error in the XSD schema.
- Ensure that all required XSD schema (.xsd) files are included in the project. You may need to click **Show All Files** on the **Project** menu to see your .xsd files in **Solution Explorer**. Right-click an .xsd file and then click **Include In Project** to include the file in your project.
- If you are using the XML to Schema Wizard, this error can occur if you infer schemas more than one time from the same source. In this case, you can remove the existing XSD schema files from the project, add a new XML to Schema item template, and then provide the XML to Schema Wizard with all the applicable XML sources for your project.
- If no error is identified in your XSD schema, the XML compiler may not have enough information to provide a detailed error message. You may be able to get more detailed error information if you ensure that the XML namespaces for the .xsd files included in your project match the XML namespaces identified for the XML Schema set in Visual Studio.

See also

- [Error List Window](#)
- [XML](#)

Evaluation of expression or statement timed out

4/9/2019 • 2 minutes to read • [Edit Online](#)

The evaluation of an expression did not complete in a timely manner.

Error ID: BC30722

To correct this error

1. Verify that the entered code is correct.
2. Simplify your expression so that it takes less time to execute.

See also

- [Debugging in Visual Studio](#)

Event '<eventname1>' cannot implement event '<eventname2>' on interface '<interface>' because their delegate types '<delegate1>' and '<delegate2>' do not match

4/28/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic cannot implement an event because the delegate type of the event does not match the delegate type of the event in the interface. This error can occur when you define multiple events in an interface and then attempt to implement them together with the same event. An event can implement two or more events only if all implemented events are declared using the `As` syntax and specify the same delegate type.

Error ID: BC31423

To correct this error

- Implement the events separately.

—or—
- Define the events in the interface using the `As` syntax and specify the same delegate type.

See also

- [Event Statement](#)
- [Delegate Statement](#)
- [Events](#)

Events cannot be declared with a delegate type that has a return type

4/28/2019 • 2 minutes to read • [Edit Online](#)

A delegate was specified for a function procedure.

Error ID: BC31084

To correct this error

- Specify a delegate for a `Sub` procedure.

See also

- [Events](#)

Events of shared WithEvents variables cannot be handled by non-shared methods

4/28/2019 • 2 minutes to read • [Edit Online](#)

A variable declared with the `Shared` modifier is a shared variable. A shared variable identifies exactly one storage location. A variable declared with the `WithEvents` modifier asserts that the type to which the variable belongs handles the set of events the variable raises. When a value is assigned to the variable, the property created by the `WithEvents` declaration unhooks any existing event handler and hooks up the new event handler via the `Add` method.

Error ID: BC30594

To correct this error

- Declare your event handler `Shared`.

See also

- [Shared](#)
- [WithEvents](#)

Expression does not produce a value

4/28/2019 • 2 minutes to read • [Edit Online](#)

You have tried to use an expression that does not produce a value in a value-producing context, such as calling a `Sub` in a context where a `Function` is expected.

Error ID: BC30491

To correct this error

- Change the expression to one that produces a value.

See also

- [Error Types](#)

Expression has the type '<typename>' which is a restricted type and cannot be used to access members inherited from 'Object' or 'ValueType'

4/9/2019 • 2 minutes to read • [Edit Online](#)

An expression evaluates to a type that cannot be boxed by the common language runtime (CLR) but accesses a member that requires boxing.

Boxing refers to the processing necessary to convert a type to `Object` or, on occasion, to `ValueType`. The common language runtime cannot box certain structure types, for example `ArgIterator`, `RuntimeArgumentHandle`, and `TypedReference`.

This expression attempts to use the restricted type to call a method inherited from `Object` or `ValueType`, such as `GetHashCode` or `ToString`. To access this method, Visual Basic has attempted an implicit boxing conversion that causes this error.

Error ID: BC31393

To correct this error

1. Locate the expression that evaluates to the cited type.
2. Locate the part of your statement that attempts to call the method inherited from `Object` or `ValueType`.
3. Rewrite the statement to avoid the method call.

See also

- [Implicit and Explicit Conversions](#)

Expression is a value and therefore cannot be the target of an assignment

7/26/2019 • 2 minutes to read • [Edit Online](#)

A statement attempts to assign a value to an expression. You can assign a value only to a writable variable, property, or array element at run time. The following example illustrates how this error can occur.

```
Dim yesterday As Integer
ReadOnly maximum As Integer = 45
yesterday + 1 = DatePart(DateInterval.Day, Now)
' The preceding line is an ERROR because of an expression on the left.
maximum = 50
' The preceding line is an ERROR because maximum is declared ReadOnly.
```

Similar examples could apply to properties and array elements.

Indirect Access. Indirect access through a value type can also generate this error. Consider the following code example, which attempts to set the value of [Point](#) by accessing it indirectly through [Location](#).

```
' Assume this code runs inside Form1.
Dim exitButton As New System.Windows.Forms.Button()
exitButton.Text = "Exit this form"
exitButton.Location.X = 140
' The preceding line is an ERROR because of no storage for Location.
```

The last statement of the preceding example fails because it creates only a temporary allocation for the [Point](#) structure returned by the [Location](#) property. A structure is a value type, and the temporary structure is not retained after the statement runs. The problem is resolved by declaring and using a variable for [Location](#), which creates a more permanent allocation for the [Point](#) structure. The following example shows code that can replace the last statement of the preceding example.

```
Dim exitLocation as New System.Drawing.Point(140, exitButton.Location.Y)
exitButton.Location = exitLocation
```

Error ID: BC30068

To correct this error

- If the statement assigns a value to an expression, replace the expression with a single writable variable, property, or array element.
- If the statement makes indirect access through a value type (usually a structure), create a variable to hold the value type.
- Assign the appropriate structure (or other value type) to the variable.
- Use the variable to access the property to assign it a value.

See also

- [Operators and Expressions](#)

- Statements
- Troubleshooting Procedures

Expression of type <type> is not queryable

4/9/2019 • 2 minutes to read • [Edit Online](#)

Expression of type <type> is not queryable. Make sure you are not missing an assembly reference and/or namespace import for the LINQ provider.

Queryable types are defined in the [System.Linq](#), [System.Data.Linq](#), and [System.Xml.Linq](#) namespaces. You must import one or more of these namespaces to perform LINQ queries.

The [System.Linq](#) namespace enables you to query objects such as collections and arrays by using LINQ.

The [System.Data.Linq](#) namespace enables you to query ADO.NET Datasets and SQL Server databases by using LINQ.

The [System.Xml.Linq](#) namespace enables you to query XML by using LINQ and to use XML features in Visual Basic.

Error ID: BC36593

To correct this error

1. Add an `Import` statement for the [System.Linq](#), [System.Data.Linq](#), or [System.Xml.Linq](#) namespace to your code file. You can also import namespaces for your project by using the **References** page of the Project Designer ([My Project](#)).
2. Ensure that the type that you have identified as the source of your query is a queryable type. That is, a type that implements `IEnumerable<T>` or `IQueryable<T>`.

See also

- [System.Linq](#)
- [System.Data.Linq](#)
- [System.Xml.Linq](#)
- [Introduction to LINQ in Visual Basic](#)
- [LINQ](#)
- [XML](#)
- [References and the Imports Statement](#)
- [Imports Statement \(.NET Namespace and Type\)](#)
- [References Page, Project Designer \(Visual Basic\)](#)

Expression recursively calls the containing property '`<propertyname>`'

10/1/2019 • 2 minutes to read • [Edit Online](#)

A statement in the `Set` procedure of a property definition stores a value into the name of the property.

The recommended approach to holding the value of a property is to define a `Private` variable in the property's container and use it in both the `Get` and `Set` procedures. The `Set` procedure should then store the incoming value in this `Private` variable.

The `Get` procedure behaves like a `Function` procedure, so it can assign a value to the property name and return control by encountering the `End Get` statement. The recommended approach, however, is to include the `Private` variable as the value in a [Return Statement](#).

The `Set` procedure behaves like a `Sub` procedure, which does not return a value. Therefore, the procedure or property name has no special meaning within a `Set` procedure, and you cannot store a value into it.

The following example illustrates the approach that can cause this error, followed by the recommended approach.

```
Public Class illustrateProperties
    ' The code in the following property causes this error.
    Public Property badProp() As Char
        Get
            Dim charValue As Char
            ' Insert code to update charValue.
            badProp = charValue
        End Get
        Set(ByVal Value As Char)
            ' The following statement causes this error.
            badProp = Value
            ' The value stored in the local variable badProp
            ' is not used by the Get procedure in this property.
        End Set
    End Property
    ' The following code uses the recommended approach.
    Private propValue As Char
    Public Property goodProp() As Char
        Get
            ' Insert code to update propValue.
            Return propValue
        End Get
        Set(ByVal Value As Char)
            propValue = Value
        End Set
    End Property
End Class
```

By default, this message is a warning. For more information about hiding warnings or treating warnings as errors, please see [Configuring Warnings in Visual Basic](#).

Error ID: BC42026

To correct this error

- Rewrite the property definition to use the recommended approach as illustrated in the preceding example.

See also

- [Property Procedures](#)
- [Property Statement](#)
- [Set Statement](#)

Expression too complex

4/28/2019 • 2 minutes to read • [Edit Online](#)

A floating-point expression contains too many nested subexpressions.

To correct this error

- Break the expression into as many separate expressions as necessary to prevent the error from occurring.

See also

- [Operators and Expressions](#)

'Extension' attribute can be applied only to 'Module', 'Sub', or 'Function' declarations

10/18/2019 • 2 minutes to read • [Edit Online](#)

The only way to extend a data type in Visual Basic is to define an extension method inside a standard module. The extension method can be a `Sub` procedure or a `Function` procedure. All extension methods must be marked with the extension attribute, `<Extension()>`, from the `System.Runtime.CompilerServices` namespace. Optionally, a module that contains an extension method may be marked in the same way. No other use of the extension attribute is valid.

Error ID: BC36550

To correct this error

- Remove the extension attribute.
- Redesign your extension as a method, defined in an enclosing module.

Example

The following example defines a `Print` method for the `String` data type.

```
Imports StringUtility
Imports System.Runtime.CompilerServices
Namespace StringUtility
    <Extension()
    Module StringExtensions
        <Extension()
        Public Sub Print (ByVal str As String)
            Console.WriteLine(str)
        End Sub
    End Module
End Namespace
```

See also

- [Attributes overview](#)
- [Extension Methods](#)
- [Module Statement](#)

File already open

4/28/2019 • 2 minutes to read • [Edit Online](#)

Sometimes a file must be closed before another `FileOpen` or other operation can occur. Among the possible causes of this error are:

- A sequential output mode `FileOpen` operation was executed for a file that is already open
- A statement refers to an open file.

To correct this error

1. Close the file before executing the statement.

See also

- [FileOpen](#)

File is too large to read into a byte array

4/28/2019 • 2 minutes to read • [Edit Online](#)

The size of the file you are attempting to read into a byte array exceeds 4 GB. The `My.Computer.FileSystem.ReadAllText` method cannot read a file that exceeds this size.

To correct this error

- Use a [StreamReader](#) to read the file. For more information, see [Basics of .NET Framework File I/O and the File System \(Visual Basic\)](#).

See also

- [ReadAllBytes](#)
- [StreamReader](#)
- [File Access with Visual Basic](#)
- [How to: Read Text from Files with a StreamReader](#)

File name or class name not found during Automation operation (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

The name specified for file name or class in a call to the `GetObject` function could not be found.

To correct this error

- Check the names and try again. Make sure the name used for the `class` parameter matches that registered with the system.

See also

- [Error Types](#)

File not found (Visual Basic Run-Time Error)

4/28/2019 • 2 minutes to read • [Edit Online](#)

The file was not found where specified. The error has the following possible causes:

- A statement refers to a file that does not exist.
- An attempt was made to call a procedure in a dynamic-link library (DLL), but the library specified in the `Lib` clause of the `Declare` statement cannot be found.
- You attempted to open a project or load a text file that does not exist.

To correct this error

1. Check the spelling of the file name and the path specification.

See also

- [Declare Statement](#)

First operand in a binary 'If' expression must be nullable or a reference type

4/28/2019 • 2 minutes to read • [Edit Online](#)

An `If` expression can take either two or three arguments. When you send only two arguments, the first argument must be a reference type or a nullable type. If the first argument evaluates to anything other than `Nothing`, its value is returned. If the first argument evaluates to `Nothing`, the second argument is evaluated and returned.

For example, the following code contains two `If` expressions, one with three arguments and one with two arguments. The expressions calculate and return the same value.

```
' firstChoice is a nullable value type.  
Dim firstChoice? As Integer = Nothing  
Dim secondChoice As Integer = 1128  
' If expression with three arguments.  
Console.WriteLine(If(firstChoice IsNot Nothing, firstChoice, secondChoice))  
' If expression with two arguments.  
Console.WriteLine(If(firstChoice, secondChoice))
```

The following expressions cause this error:

```
Dim choice1 = 4  
Dim choice2 = 5  
Dim booleanVar = True  
  
' Not valid.  
'Console.WriteLine(If(choice1 < choice2, 1))  
' Not valid.  
'Console.WriteLine(If(booleanVar, "Test returns True."))
```

Error ID: BC33107

To correct this error

- If you cannot change the code so that the first argument is a nullable type or reference type, consider converting to a three-argument `If` expression, or to an `If...Then...Else` statement.

```
Console.WriteLine(If(choice1 < choice2, 1, 2))  
Console.WriteLine(If(booleanVar, "Test returns True.", "Test returns False.))
```

See also

- [If Operator](#)
- [If...Then...Else Statement](#)
- [Nullable Value Types](#)

First statement of this 'Sub New' must be a call to ' MyBase.New' or ' MyClass.New' (No Accessible Constructor Without Parameters)

4/28/2019 • 2 minutes to read • [Edit Online](#)

First statement of this 'Sub New' must be a call to ' MyBase.New' or ' MyClass.New' because base class '<basename>' of '<derivedname>' does not have an accessible 'Sub New' that can be called with no arguments.

In a derived class, every constructor must call a base class constructor (`MyBase.New`). If the base class has a constructor with no parameters that is accessible to derived classes, `MyBase.New` can be called automatically. If not, a base class constructor must be called with parameters, and this cannot be done automatically. In this case, the first statement of every derived class constructor must call a parameterized constructor on the base class, or call another constructor in the derived class that makes a base class constructor call.

Error ID: BC30148

To correct this error

- Either call `MyBase.New` supplying the required parameters, or call a peer constructor that makes such a call.

For example, if the base class has a constructor that's declared as `Public Sub New(ByVal index As Integer)`, the first statement in the derived class constructor might be `MyBase.New(100)`.

See also

- [Inheritance Basics](#)

First statement of this 'Sub New' must be an explicit call to 'MyBase.New' or 'MyClass.New' because the '<constructorname>' in the base class '<baseclassname>' of '<derivedclassname>' is marked obsolete: '<errormessage>'

4/9/2019 • 2 minutes to read • [Edit Online](#)

A class constructor does not explicitly call a base class constructor, and the implicit base class constructor is marked with the [ObsoleteAttribute](#) attribute and the directive to treat it as an error.

When a derived class constructor does not call a base class constructor, Visual Basic attempts to generate an implicit call to a parameterless base class constructor. If there is no accessible constructor in the base class that can be called without arguments, Visual Basic cannot generate an implicit call. In this case, the required constructor is marked with the [ObsoleteAttribute](#) attribute, so Visual Basic cannot call it.

You can mark any programming element as being no longer in use by applying [ObsoleteAttribute](#) to it. If you do this, you can set the attribute's [IsError](#) property to either `True` or `False`. If you set it to `True`, the compiler treats an attempt to use the element as an error. If you set it to `False`, or let it default to `False`, the compiler issues a warning if there is an attempt to use the element.

Error ID: BC30920

To correct this error

1. Examine the quoted error message and take appropriate action.
2. Include a call to `MyBase.New()` or `MyClass.New()` as the first statement of the `Sub New` in the derived class.

See also

- [Attributes overview](#)

'For Each' on type '<typename>' is ambiguous because the type implements multiple instantiations of 'System.Collections.Generic.IEnumerable(Of T)'

5/14/2019 • 2 minutes to read • [Edit Online](#)

A `For Each` statement specifies an iterator variable that has more than one [GetEnumerator](#) method.

The iterator variable must be of a type that implements the [System.Collections.IEnumerable](#) or [System.Collections.Generic.IEnumerable<T>](#) interface in one of the [Collections](#) namespaces of the .NET Framework. It is possible for a class to implement more than one constructed generic interface, using a different type argument for each construction. If a class that does this is used for the iterator variable, that variable has more than one [GetEnumerator](#) method. In such a case, Visual Basic cannot choose which method to call.

Error ID: BC32096

To correct this error

- Use [DirectCast Operator](#) or [TryCast Operator](#) to cast the iterator variable type to the interface defining the [GetEnumerator](#) method you want to use.

See also

- [For Each...Next Statement](#)
- [Interfaces](#)

Friend assembly reference <reference> is invalid

9/13/2019 • 2 minutes to read • [Edit Online](#)

Friend assembly reference <reference> is invalid. Strong-name signed assemblies must specify a public key in their `InternalsVisibleTo` declarations.

The assembly name passed to the `InternalsVisibleToAttribute` attribute constructor identifies a strong-named assembly, but it does not include a `PublicKey` attribute.

Error ID: BC31535

To correct this error

1. Determine the public key for the strong-named friend assembly. Include the public key as part of the assembly name passed to the `InternalsVisibleToAttribute` attribute constructor by using the `PublicKey` attribute.

See also

- [AssemblyName](#)
- [Friend Assemblies](#)

Function '<procedurename>' doesn't return a value on all code paths

4/28/2019 • 2 minutes to read • [Edit Online](#)

Function '<procedurename>' doesn't return a value on all code paths. Are you missing a 'Return' statement?

A `Function` procedure has at least one possible path through its code that does not return a value.

You can return a value from a `Function` procedure in any of the following ways:

- Include the value in a [Return Statement](#).
- Assign the value to the `Function` procedure name and then perform an `Exit Function` statement.
- Assign the value to the `Function` procedure name and then perform the `End Function` statement.

If control passes to `Exit Function` or `End Function` and you have not assigned any value to the procedure name, the procedure returns the default value of the return data type. For more information, see "Behavior" in [Function Statement](#).

By default, this message is a warning. For more information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC42105

To correct this error

- Check your control flow logic and make sure you assign a value before every statement that causes a return.

It is easier to guarantee that every return from the procedure returns a value if you always use the `Return` statement. If you do this, the last statement before `End Function` should be a `Return` statement.

See also

- [Function Procedures](#)
- [Function Statement](#)
- [Compile Page, Project Designer \(Visual Basic\)](#)

Function evaluation is disabled because a previous function evaluation timed out

4/9/2019 • 2 minutes to read • [Edit Online](#)

Function evaluation is disabled because a previous function evaluation timed out. To re-enable function evaluation, step again or restart debugging.

In the Visual Studio debugger, an expression specifies a procedure call, but another evaluation has timed out.

Possible causes for a procedure call to time out include an infinite loop or *endless loop*. For more information, see [For...Next Statement](#).

A special case of an infinite loop is *recursion*. For more information, see [Recursive Procedures](#).

Error ID: BC30957

To correct this error

1. If possible, determine what the previous function evaluation was and what caused it to time out. Otherwise, you might encounter this error again.
2. Either step the debugger again, or terminate and restart debugging.

See also

- [Debugging in Visual Studio](#)
- [Navigating through Code with the Debugger](#)

Generic parameters used as optional parameter types must be class constrained

4/28/2019 • 2 minutes to read • [Edit Online](#)

A procedure is declared with an optional parameter that uses a type parameter that is not constrained to be a reference type.

You must always supply a default value for each optional parameter. If the parameter is of a reference type, the optional value must be `Nothing`, which is a valid value for any reference type. However, if the parameter is of a value type, that type must be an elementary data type predefined by Visual Basic. This is because a composite value type, such as a user-defined structure, has no valid default value.

When you use a type parameter for an optional parameter, you must guarantee that it is of a reference type to avoid the possibility of a value type with no valid default value. This means you must constrain the type parameter either with the `Class` keyword or with the name of a specific class.

Error ID: BC32124

To correct this error

- Constrain the type parameter to accept only a reference type, or do not use it for the optional parameter.

See also

- [Generic Types in Visual Basic](#)
- [Type List](#)
- [Class Statement](#)
- [Optional Parameters](#)
- [Structures](#)
- [Nothing](#)

'Get' accessor of property '<propertyname>' is not accessible

4/28/2019 • 2 minutes to read • [Edit Online](#)

A statement attempts to retrieve the value of a property when it does not have access to the property's `Get` procedure.

If the [Get Statement](#) is marked with a more restrictive access level than its [Property Statement](#), an attempt to read the property value could fail in the following cases:

- The `Get` statement is marked [Private](#) and the calling code is outside the class or structure in which the property is defined.
- The `Get` statement is marked [Protected](#) and the calling code is not in the class or structure in which the property is defined, nor in a derived class.
- The `Get` statement is marked [Friend](#) and the calling code is not in the same assembly in which the property is defined.

Error ID: BC31103

To correct this error

- If you have control of the source code defining the property, consider declaring the `Get` procedure with the same access level as the property itself.
- If you do not have control of the source code defining the property, or you must restrict the `Get` procedure access level more than the property itself, try to move the statement that reads the property value to a region of code that has better access to the property.

See also

- [Property Procedures](#)
- [How to: Declare a Property with Mixed Access Levels](#)

Handles clause requires a WithEvents variable defined in the containing type or one of its base types

10/18/2019 • 2 minutes to read • [Edit Online](#)

You did not supply a `WithEvents` variable in your `Handles` clause. The `Handles` keyword at the end of a procedure declaration causes it to handle events raised by an object variable declared using the `WithEvents` keyword.

Error ID: BC30506

To correct this error

Supply the necessary `WithEvents` variable.

Example

In the following example, Visual Basic generates compiler error `BC30506` because the `WithEvents` keyword is not used in the definition of the `System.Timers.Timer` instance.

```
Imports System.Timers

Module Module1
    Private _timer1 As New Timer() With {.Interval = 1000, .Enabled = True}

    Sub Main()
        Console.WriteLine("Press any key to start the timer...")
        Console.ReadKey()
        _timer1.Start()
        Console.ReadKey()
    End Sub

    Private Sub Timer1_Tick(sender As Object, args As EventArgs) Handles _timer1.Elapsed
        Console.WriteLine("Press any key to terminate...")
    End Sub
End Module
```

The following example compiles successfully because the `_timer1` variable is defined with the `WithEvents` keyword:

```
Imports System.Timers

Module Module1
    Private WithEvents _timer1 As New Timer() With {.Interval = 1000}

    Sub Main()
        Console.WriteLine("Press any key to start the timer...")
        Console.ReadKey()
        _timer1.Start()
        Console.ReadKey()
    End Sub

    Private Sub Timer1_Tick(sender As Object, args As EventArgs) Handles _timer1.Elapsed
        Console.WriteLine("Press any key to terminate...")
    End Sub

End Module
```

See also

- [Handles](#)

Identifier expected

4/28/2019 • 2 minutes to read • [Edit Online](#)

A programming element that is not a recognizable declared element name occurs where the context requires an element name. One possible cause is that an attribute has been specified somewhere other than at the beginning of the statement.

Error ID: BC30203

To correct this error

- Verify that any attributes in the statement are all placed at the beginning.
- Verify that all element names in the statement are spelled correctly.

See also

- [Declared Element Names](#)
- [Attributes overview](#)

Identifier is too long

4/28/2019 • 2 minutes to read • [Edit Online](#)

The name, or identifier, of every programming element is limited to 1023 characters. In addition, a fully qualified name cannot exceed 1023 characters. This means that the entire identifier string (`<namespace>.<...>.<namespace>.<class>.<element>`) cannot be more than 1023 characters long, including the member-access operator (`.`) characters.

Error ID: BC30033

To correct this error

- Reduce the length of the identifier.

See also

- [Declared Element Names](#)

Initializer expected

4/9/2019 • 2 minutes to read • [Edit Online](#)

You have tried to declare an instance of a class by using an object initializer in which the initialization list is empty, as shown in the following example.

```
' Not valid.
```

```
' Dim aStudent As New Student With {}
```

At least one field or property must be initialized in the initializer list, as shown in the following example.

```
Dim aStudent As New Student With {.year = "Senior"}
```

Error ID: BC30996

To correct this error

1. Initialize at least one field or property in the initializer, or do not use an object initializer.

See also

- [Object Initializers: Named and Anonymous Types](#)
- [How to: Declare an Object by Using an Object Initializer](#)

Input past end of file

4/9/2019 • 2 minutes to read • [Edit Online](#)

Either an `Input` statement is reading from a file that is empty or one in which all the data is used, or you used the `EOF` function with a file opened for binary access.

To correct this error

1. Use the `EOF` function immediately before the `Input` statement to detect the end of the file.
2. If the file is opened for binary access, use `Seek` and `Loc`.

See also

- [Input](#)
- [EOF](#)
- [Seek](#)
- [Loc](#)

Internal error happened at <location>

4/28/2019 • 2 minutes to read • [Edit Online](#)

An internal error has occurred. The line at which it occurred is contained in the error message.

To correct this error

- Make sure this error was not generated by the `Error` statement or `Raise` method; if it was not, contact Microsoft Product Support Services to report the conditions under which the message appeared.

See also

- [Debugger Basics](#)

Implicit conversion from '<typename1>' to '<typename2>' in copying the value of 'ByRef' parameter '<parametername>' back to the matching argument.

4/28/2019 • 2 minutes to read • [Edit Online](#)

A procedure is called with a [ByRef](#) argument of a different type than that of its corresponding parameter.

If you pass an argument [ByRef](#), Visual Basic sometimes copies the argument value into a local variable in the procedure instead of passing a reference. In such a case, when the procedure returns, Visual Basic must then copy the local variable value back into the argument in the calling code.

If a [ByRef](#) argument value is copied into the procedure and the argument and parameter are of the same type, no conversion is necessary. But if the types are different, Visual Basic must convert in both directions. Because you cannot use [CType](#) or any of the other conversion keywords on a procedure argument or parameter, such a conversion is always implicit.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC41999

To correct this error

- If possible, use a calling argument of the same type as the procedure parameter, so Visual Basic does not need to do any conversion.
- If you need to call the procedure with an argument type different from the parameter type but do not need to return a value into the calling argument, define the parameter to be [ByVal](#) instead of [ByRef](#).

See also

- [Procedures](#)
- [Procedure Parameters and Arguments](#)
- [Passing Arguments by Value and by Reference](#)
- [Implicit and Explicit Conversions](#)

'Is' requires operands that have reference types, but this operand has the value type '<typename>'

4/28/2019 • 2 minutes to read • [Edit Online](#)

The `Is` comparison operator determines whether two object variables refer to the same instance. This comparison is not defined for value types.

Error ID: BC30020

To correct this error

- Use the appropriate arithmetic comparison operator or the `Like` operator to compare two value types.

See also

- [Is Operator](#)
- [Like Operator](#)
- [Comparison Operators](#)

' IsNot' operand of type 'typename' can only be compared to 'Nothing', because 'typename' is a nullable type

10/17/2019 • 2 minutes to read • [Edit Online](#)

A variable declared as nullable has been compared to an expression other than `Nothing` using the `IsNot` operator.

Error ID: BC32128

To correct this error

To compare a nullable type to an expression other than `Nothing` by using the `IsNot` operator, call the `GetType` method on the nullable type and compare the result to the expression, as shown in the following example.

```
Dim number? As Integer = 5

If number IsNot Nothing Then
    If number.GetType() IsNot Type.GetType("System.Int32") Then

        End If
    End If
```

See also

- [Nullable Value Types](#)
- [IsNot Operator](#)

Labels that are numbers must be followed by colons

10/1/2019 • 2 minutes to read • [Edit Online](#)

Line numbers follow the same rules as other kinds of labels, and must contain a colon.

Error ID: BC30801

To correct this error

- Place the number followed by a colon at the start of a line of code; for example:

```
400:     X += 1
```

See also

- [GoTo Statement](#)

Lambda expression will not be removed from this event handler

10/18/2019 • 2 minutes to read • [Edit Online](#)

Lambda expression will not be removed from this event handler. Assign the lambda expression to a variable and use the variable to add and remove the event.

When lambda expressions are used with event handlers, you may not see the behavior you expect. The compiler generates a new method for each lambda expression definition, even if they are identical. Therefore, the following code displays `False`.

```
Module Module1

    Sub Main()
        Dim fun1 As ChangeInteger = Function(p As Integer) p + 1
        Dim fun2 As ChangeInteger = Function(p As Integer) p + 1
        Console.WriteLine(fun1 = fun2)
    End Sub

    Delegate Function ChangeInteger(ByVal x As Integer) As Integer

End Module
```

When lambda expressions are used with event handlers, this may cause unexpected results. In the following example, the lambda expression added by `AddHandler` is not removed by the `RemoveHandler` statement.

```
Module Module1

    Event ProcessInteger(ByVal x As Integer)

    Sub Main()

        ' The following line adds one listener to the event.
        AddHandler ProcessInteger, Function(m As Integer) m

        ' The following statement searches the current listeners
        ' for a match to remove. However, this lambda is not the same
        ' as the previous one, so nothing is removed.
        RemoveHandler ProcessInteger, Function(m As Integer) m

    End Sub
End Module
```

By default, this message is a warning. For more information about how to hide warnings or treat warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC42326

To correct this error

To avoid the warning and remove the lambda expression, assign the lambda expression to a variable and use the variable in both the `AddHandler` and `RemoveHandler` statements, as shown in the following example.

```
Module Module1

    Event ProcessInteger(ByVal x As Integer)

    Dim PrintHandler As ProcessIntegerEventHandler

    Sub Main()

        ' Assign the lambda expression to a variable.
        PrintHandler = Function(m As Integer) m

        ' Use the variable to add the listener.
        AddHandler ProcessInteger, PrintHandler

        ' Use the variable again when you want to remove the listener.
        RemoveHandler ProcessInteger, PrintHandler

    End Sub
End Module
```

See also

- [Lambda Expressions](#)
- [Relaxed Delegate Conversion](#)
- [Events](#)

Lambda expressions are not valid in the first expression of a 'Select Case' statement

4/28/2019 • 2 minutes to read • [Edit Online](#)

You cannot use a lambda expression for the test expression in a `Select Case` statement. Lambda expression definitions return functions, and the test expression of a `Select Case` statement must be an elementary data type.

The following code causes this error:

```
' Select Case (Function(arg) arg Is Nothing)
    ' List of the cases.
' End Select
```

Error ID: BC36635

To correct this error

- Examine your code to determine whether a different conditional construction, such as an `If...Then...Else` statement, would work for you.
- You may have intended to call the function, as shown in the following code:

```
Dim num? As Integer
Select Case ((Function(arg? As Integer) arg Is Nothing)(num))
    ' List of the cases
End Select
```

See also

- [Lambda Expressions](#)
- [If...Then...Else Statement](#)
- [Select...Case Statement](#)

Late bound resolution; runtime errors could occur

4/28/2019 • 2 minutes to read • [Edit Online](#)

An object is assigned to a variable declared to be of the [Object Data Type](#).

When you declare a variable as `Object`, the compiler must perform *late binding*, which causes extra operations at run time. It also exposes your application to potential run-time errors. For example, if you assign a [Form](#) to the `Object` variable and then try to access the [XmlDocument.NameTable](#) property, the runtime throws a [MemberAccessException](#) because the [Form](#) class does not expose a `NameTable` property.

If you declare the variable to be of a specific type, the compiler can perform *early binding* at compile time. This results in improved performance, controlled access to the members of the specific type, and better readability of your code.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC42017

To correct this error

- If possible, declare the variable to be of a specific type.

See also

- [Early and Late Binding](#)
- [Object Variable Declaration](#)

Latebound overload resolution cannot be applied to '`<procedurename>`' because the accessing instance is an interface type

7/26/2019 • 2 minutes to read • [Edit Online](#)

The compiler is attempting to resolve a reference to an overloaded property or procedure, but the reference fails because an argument is of type `Object` and the referring object has the data type of an interface. The `Object` argument forces the compiler to resolve the reference as late-bound.

In these circumstances, the compiler resolves the overload through the implementing class instead of through the underlying interface. If the class renames one of the overloaded versions, the compiler does not consider that version to be an overload because its name is different. This in turn causes the compiler to ignore the renamed version when it might have been the correct choice to resolve the reference.

Error ID: BC30933

To correct this error

- Use `CType` to cast the argument from `Object` to the type specified by the signature of the overload you want to call.

Note that it does not help to cast the referring object to the underlying interface. You must cast the argument to avoid this error.

Example

The following example shows a call to an overloaded `Sub` procedure that causes this error at compile time.

```
Module m1
    Interface i1
        Sub s1(ByVal p1 As Integer)
        Sub s1(ByVal p1 As Double)
    End Interface
    Class c1
        Implements i1
        Public Overloads Sub s1(ByVal p1 As Integer) Implements i1.s1
        End Sub
        Public Overloads Sub s2(ByVal p1 As Double) Implements i1.s1
        End Sub
    End Class
    Sub Main()
        Dim refer As i1 = New c1
        Dim o1 As Object = 3.1415
        ' The following reference is INVALID and causes a compiler error.
        refer.s1(o1)
    End Sub
End Module
```

In the preceding example, if the compiler allowed the call to `s1` as written, the resolution would take place through the class `c1` instead of the interface `i1`. This would mean that the compiler would not consider `s2` because its name is different in `c1`, even though it is the correct choice as defined by `i1`.

You can correct the error by changing the call to either of the following lines of code:

```
refer.s1(CType(o1, Integer))
refer.s1(CType(o1, Double))
```

Each of the preceding lines of code explicitly casts the `Object` variable `o1` to one of the parameter types defined for the overloads.

See also

- [Procedure Overloading](#)
- [Overload Resolution](#)
- [CType Function](#)

Leading '.' or '!' can only appear inside a 'With' statement

4/9/2019 • 2 minutes to read • [Edit Online](#)

A period (.) or exclamation point (!) that is not inside a `With` block occurs without an expression on the left.

Member access (`.`) and dictionary member access (`!`) require an expression specifying the element that contains the member. This must appear immediately to the left of the accessor or as the target of a `With` block containing the member access.

Error ID: BC30157

To correct this error

1. Ensure that the `With` block is correctly formatted.
2. If there is no `With` block, add an expression to the left of the accessor that evaluates to a defined element containing the member.

See also

- [Special Characters in Code](#)
- [With...End With Statement](#)

Line is too long

4/28/2019 • 2 minutes to read • [Edit Online](#)

Source text lines cannot exceed 65535 characters.

Error ID: BC30494

To correct this error

- Shorten the length of the line to 65535 characters or fewer.

See also

- [Error Types](#)

'Line' statements are no longer supported (Visual Basic Compiler Error)

4/9/2019 • 2 minutes to read • [Edit Online](#)

Line statements are no longer supported. File I/O functionality is available as

`Microsoft.VisualBasic.FileSystem.LineInput` and graphics functionality is available as

`System.Drawing.Graphics.DrawLine`.

Error ID: BC30830

To correct this error

1. If performing file access, use `Microsoft.VisualBasic.FileSystem.LineInput`.
2. If performing graphics, use `System.Drawing.Graphics.DrawLine`.

See also

- [System.IO](#)
- [System.Drawing](#)
- [File Access with Visual Basic](#)

Method does not have a signature compatible with the delegate

4/2/2019 • 2 minutes to read • [Edit Online](#)

There is an incompatibility between the signatures of the method and the delegate you are trying to use. The `Delegate` statement defines the parameter types and return types of a delegate class. Any procedure that has matching parameters of compatible types and return types can be used to create an instance of this delegate type.

Error ID: BC36563

See also

- [AddressOf Operator](#)
- [Delegate Statement](#)
- [Overload Resolution](#)
- [Generic Types in Visual Basic](#)

Methods of 'System.Nullable(Of T)' cannot be used as operands of the 'AddressOf' operator

4/28/2019 • 2 minutes to read • [Edit Online](#)

A statement uses the `AddressOf` operator with an operand that represents a procedure of the `Nullable<T>` structure.

Error ID: BC32126

To correct this error

- Replace the procedure name in the `AddressOf` clause with an operand that is not a member of `Nullable<T>`.
- Write a class that wraps the method of `Nullable<T>` that you want to use. In the following example, the `NullableWrapper` class defines a new method named `GetValueOrDefault`. Because this new method is not a member of `Nullable<T>`, it can be applied to `nullInstance`, an instance of a nullable type, to form an argument for `AddressOf`.

```
Module Module1

    Delegate Function Deleg() As Integer

    Sub Main()
        Dim nullInstance As New Nullable(Of Integer)(1)

        Dim del As Deleg

        ' GetValueOrDefault is a method of the Nullable generic
        ' type. It cannot be used as an operand of AddressOf.
        ' del = AddressOf nullInstance.GetValueOrDefault

        ' The following line uses the GetValueOrDefault method
        ' defined in the NullableWrapper class.
        del = AddressOf (New NullableWrapper(
            Of Integer)(nullInstance)).GetValueOrDefault

        Console.WriteLine(del.Invoke())
    End Sub

    Class NullableWrapper(Of T As Structure)
        Private m_Value As Nullable(Of T)

        Sub New(ByVal Value As Nullable(Of T))
            m_Value = Value
        End Sub

        Public Function GetValueOrDefault() As T
            Return m_Value.Value
        End Function
    End Class
End Module
```

See also

- [Nullable<T>](#)

- [AddressOf Operator](#)
- [Nullable Value Types](#)
- [Generic Types in Visual Basic](#)

'Module' statements can occur only at file or namespace level

4/28/2019 • 2 minutes to read • [Edit Online](#)

`Module` statements must appear at the top of your source file immediately after `Option` and `Imports` statements, global attributes, and namespace declarations, but before all other declarations.

Error ID: BC30617

To correct this error

- Move the `Module` statement to the top of your namespace declaration or source file.

See also

- [Module Statement](#)

Name <membername> is not CLS-compliant

4/28/2019 • 2 minutes to read • [Edit Online](#)

An assembly is marked as `<CLSCompliant(True)>` but exposes a member with a name that begins with an underscore (`_`).

A programming element can contain one or more underscores, but to be compliant with the [Language Independence and Language-Independent Components](#) (CLS), it must not begin with an underscore. See [Declared Element Names](#).

When you apply the [CLSCompliantAttribute](#) to a programming element, you set the attribute's `isCompliant` parameter to either `True` or `False` to indicate compliance or noncompliance. There is no default for this parameter, and you must supply a value.

If you do not apply the [CLSCompliantAttribute](#) to an element, it is considered to be noncompliant.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC40031

To correct this error

- If you have control over the source code, change the member name so that it does not begin with an underscore.
- If you require that the member name remain unchanged, remove the [CLSCompliantAttribute](#) from its definition or mark it as `<CLSCompliant(False)>`. You can still mark the assembly as `<CLSCompliant(True)>`.

See also

- [Declared Element Names](#)
- [Visual Basic Naming Conventions](#)

Name '<name>' is not declared

9/4/2019 • 2 minutes to read • [Edit Online](#)

A statement refers to a programming element, but the compiler cannot find an element with that exact name.

Error ID: BC30451

To correct this error

1. Check the spelling of the name in the referring statement. Visual Basic is case-insensitive, but any other variation in the spelling is regarded as a completely different name. Note that the underscore (`_`) is part of the name and therefore part of the spelling.
2. Check that you have the member access operator (`.`) between an object and its member. For example, if you have a `TextBox` control named `TextBox1`, to access its `Text` property you should type `TextBox1.Text`. If instead you type `TextBox1Text`, you have created a different name.
3. If the spelling is correct and the syntax of any object member access is correct, verify that the element has been declared. For more information, see [Declared Elements](#).
4. If the programming element has been declared, check that it is in scope. If the referring statement is outside the region declaring the programming element, you might need to qualify the element name. For more information, see [Scope in Visual Basic](#).
5. If you are not using a fully qualified type or type and member name (for example, your code refers to a property as `MethodInfo.Name` instead of `System.Reflection.MethodInfo.Name`), add an [Imports statement](#).
6. If you are attempting to compile an SDK-style project (a project with a *.vbproj file that begins with the line `<Project Sdk="Microsoft.NET.Sdk">`), and the error message refers to a type or member in the `Microsoft.VisualBasic.dll` assembly, configure your application to compile with a reference to the Visual Basic Runtime Library. By default, a subset of the library is embedded in your assembly in an SDK-style project.

For example, the following example fails to compile because the `Microsoft.VisualBasic.CompilerServices.ChangeType` method cannot be found. It is not embedded in the subset of the Visual Basic Runtime included with your application.

```
Imports Microsoft.VisualBasic.CompilerServices

Public Module Example
    Sub Main(args As String())
        Dim originalValue As String = args(0)
        Dim t As Type = GetType(Int32)
        Dim i As Int32 = Conversions.ChangeType(originalValue, t)
        Console.WriteLine($"'{originalValue}' --> {i}")
    End Sub
End Module
```

To address this error, add the `<VBRuntime>Default</VBRuntime>` element to the projects `<PropertyGroup>` section, as the following Visual Basic project file shows.

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <Reference Include="Microsoft.VisualBasic" />
  </ItemGroup>
  <PropertyGroup>
    <VBRuntime>Default</VBRuntime>
    <OutputType>Exe</OutputType>
    <RootNamespace>vbruntime</RootNamespace>
    <TargetFramework>net472</TargetFramework>
  </PropertyGroup>

</Project>
```

See also

- [Declarations and Constants Summary](#)
- [Visual Basic Naming Conventions](#)
- [Declared Element Names](#)
- [References to Declared Elements](#)

Name <namespacename> in the root namespace <fullnamespacename> is not CLS-compliant

4/28/2019 • 2 minutes to read • [Edit Online](#)

An assembly is marked as `<CLSCompliant(True)>`, but an element of the root namespace name begins with an underscore (`_`).

A programming element can contain one or more underscores, but to be compliant with the [Language Independence and Language-Independent Components](#) (CLS), it must not begin with an underscore. See [Declared Element Names](#).

When you apply the [CLSCompliantAttribute](#) to a programming element, you set the attribute's `isCompliant` parameter to either `True` or `False` to indicate compliance or noncompliance. There is no default for this parameter, and you must supply a value.

If you do not apply the [CLSCompliantAttribute](#) to an element, it is considered to be noncompliant.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC40039

To correct this error

- If you require CLS compliance, change the root namespace name so that none of its elements begins with an underscore.
- If you require that the namespace name remain unchanged, then remove the [CLSCompliantAttribute](#) from the assembly or mark it as `<CLSCompliant(False)>`.

See also

- [Namespace Statement](#)
- [Namespaces in Visual Basic](#)
- [/rootnamespace](#)
- [Application Page, Project Designer \(Visual Basic\)](#)
- [Declared Element Names](#)
- [Visual Basic Naming Conventions](#)

Namespace or type specified in the Imports '`<qualifiedelementname>`' doesn't contain any public member or cannot be found

3/5/2019 • 2 minutes to read • [Edit Online](#)

Namespace or type specified in the Imports '`<qualifiedelementname>`' doesn't contain any public member or cannot be found. Make sure the namespace or the type is defined and contains at least one public member. Make sure the alias name doesn't contain other aliases.

An `Imports` statement specifies a containing element that either cannot be found or does not define any `Public` members.

A *containing element* can be a namespace, class, structure, module, interface, or enumeration. The containing element contains members, such as variables, procedures, or other containing elements.

The purpose of importing is to allow your code to access namespace or type members without having to qualify them. Your project might also need to add a reference to the namespace or type. For more information, see "Importing Containing Elements" in [References to Declared Elements](#).

If the compiler cannot find the specified containing element, then it cannot resolve references that use it. If it finds the element but the element does not expose any `Public` members, then no reference can be successful. In either case it is meaningless to import the element.

Keep in mind that if you import a containing element and assign an import alias to it, then you cannot use that import alias to import another element. The following code generates a compiler error.

```
Imports winfrm = System.Windows.Forms  
' The following statement is INVALID because it reuses an import alias.  
Imports behave = winfrm.Design.Behavior'
```

Error ID: BC40056

To correct this error

1. Verify that the containing element is accessible from your project.
2. Verify that the specification of the containing element does not include any import alias from another import.
3. Verify that the containing element exposes at least one `Public` member.

See also

- [Imports Statement \(.NET Namespace and Type\)](#)
- [Namespace Statement](#)
- [Public](#)
- [Namespaces in Visual Basic](#)
- [References to Declared Elements](#)

Namespace or type specified in the project-level Imports '<qualifiedelementname>' doesn't contain any public member or cannot be found

4/9/2019 • 2 minutes to read • [Edit Online](#)

Namespace or type specified in the project-level Imports '<qualifiedelementname>' doesn't contain any public member or cannot be found. Make sure the namespace or the type is defined and contains at least one public member. Make sure the alias name doesn't contain other aliases.

An import property of a project specifies a containing element that either cannot be found or does not define any [Public](#) members.

A *containing element* can be a namespace, class, structure, module, interface, or enumeration. The containing element contains members, such as variables, procedures, or other containing elements.

The purpose of importing is to allow your code to access namespace or type members without having to qualify them. Your project might also need to add a reference to the namespace or type. For more information, see "Importing Containing Elements" in [References to Declared Elements](#).

If the compiler cannot find the specified containing element, then it cannot resolve references that use it. If it finds the element but the element does not expose any [Public](#) members, then no reference can be successful. In either case it is meaningless to import the element.

You use the **Project Designer** to specify elements to import. Use the **Imported namespaces** section of the **References** page. You can get to the **Project Designer** by double-clicking the **My Project** icon in **Solution Explorer**.

Error ID: BC40057

To correct this error

1. Open the **Project Designer** and switch to the **Reference** page.
2. In the **Imported namespaces** section, verify that the containing element is accessible from your project.
3. Verify that the containing element exposes at least one [Public](#) member.

See also

- [References Page, Project Designer \(Visual Basic\)](#)
- [Managing Project and Solution Properties](#)
- [Public](#)
- [Namespaces in Visual Basic](#)
- [References to Declared Elements](#)

Need property array index

4/28/2019 • 2 minutes to read • [Edit Online](#)

This property value consists of an array rather than a single value. You did not specify the index for the property array you tried to access.

To correct this error

- Check the component's documentation to find the range for the indexes appropriate for the array. Specify an appropriate index in your property access statement.

See also

- [Error Types](#)
- [Talk to Us](#)

Nested function does not have a signature that is compatible with delegate '<delegatename>'

10/18/2019 • 2 minutes to read • [Edit Online](#)

A lambda expression has been assigned to a delegate that has an incompatible signature. For example, in the following code, delegate `Del` has two integer parameters.

```
Delegate Function Del(ByVal p As Integer, ByVal q As Integer) As Integer
```

The error is raised if a lambda expression with one argument is declared as type `Del`:

```
' Neither of these is valid.  
' Dim lambda1 As Del = Function(n As Integer) n + 1  
' Dim lambda2 As Del = Function(n) n + 1
```

Error ID: BC36532

To correct this error

Adjust either the delegate definition or the assigned lambda expression so that the signatures are compatible.

See also

- [Relaxed Delegate Conversion](#)
- [Lambda Expressions](#)

No accessible 'Main' method with an appropriate signature was found in '<name>'

4/28/2019 • 2 minutes to read • [Edit Online](#)

Command-line applications must have a `Sub Main` defined. `Main` must be declared as `Public Shared` if it is defined in a class, or as `Public` if defined in a module.

Error ID: BC30737

To correct this error

- Define a `Public Sub Main` procedure for your project. Declare it as `Shared` if and only if you define it inside a class.

See also

- [Structure of a Visual Basic Program](#)
- [Procedures](#)

Non-CLS-compliant <membername> is not allowed in a CLS-compliant interface

4/28/2019 • 2 minutes to read • [Edit Online](#)

A property, procedure, or event in an interface is marked as `<CLSCompliant(True)>` when the interface itself is marked as `<CLSCompliant(False)>` or is not marked.

For an interface to be compliant with the [Language Independence and Language-Independent Components \(CLS\)](#), all its members must be compliant.

When you apply the [CLSCompliantAttribute](#) to a programming element, you set the attribute's `isCompliant` parameter to either `True` or `False` to indicate compliance or noncompliance. There is no default for this parameter, and you must supply a value.

If you do not apply the [CLSCompliantAttribute](#) to an element, it is considered to be noncompliant.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC40033

To correct this error

- If you require CLS compliance and have control over the interface source code, mark the interface as `<CLSCompliant(True)>` if all its members are compliant.
- If you require CLS compliance and do not have control over the interface source code, or if it does not qualify to be compliant, define this member within a different interface.
- If you require that this member remain within its current interface, remove the [CLSCompliantAttribute](#) from its definition or mark it as `<CLSCompliant(False)>`.

See also

- [Interface Statement](#)

Nullable type inference is not supported in this context

4/28/2019 • 2 minutes to read • [Edit Online](#)

Value types and structures can be declared nullable.

```
Dim a? As Integer  
Dim b As Integer?
```

However, you cannot use the nullable declaration in combination with type inference. The following examples cause this error.

```
' Not valid.  
' Dim c? = 10  
' Dim d? = a
```

Error ID: BC36629

To correct this error

- Use an `As` clause to declare the variable as nullable.

See also

- [Nullable Value Types](#)
- [Local Type Inference](#)

Number of indices exceeds the number of dimensions of the indexed array

4/28/2019 • 2 minutes to read • [Edit Online](#)

The number of indices used to access an array element must be exactly the same as the rank of the array, that is, the number of dimensions declared for it.

Error ID: BC30106

To correct this error

- Remove subscripts from the array reference until the total number of subscripts equals the rank of the array.
For example:

```
Dim gameBoard(3, 3) As String

' Incorrect code. The array has two dimensions.
gameBoard(1, 1, 1) = "X"
gameBoard(2, 1, 1) = "O"

' Correct code.
gameBoard(0, 0) = "X"
gameBoard(1, 0) = "O"
```

See also

- [Arrays](#)

Object or class does not support the set of events

4/9/2019 • 2 minutes to read • [Edit Online](#)

You tried to use a `WithEvents` variable with a component that cannot work as an event source for the specified set of events. For example, you wanted to sink the events of an object, then create another object that `Implements` the first object. Although you might think you could sink the events from the implemented object, this is not always the case. `Implements` only implements an interface for methods and properties. `WithEvents` is not supported for private `UserControls`, because the type info needed to raise the `ObjectEvent` is not available at run time.

To correct this error

1. You cannot sink events for a component that does not source events.

See also

- [WithEvents](#)
- [Implements Statement](#)

Object required (Visual Basic)

4/9/2019 • 2 minutes to read • [Edit Online](#)

References to properties and methods often require an explicit object qualifier. This is such a case.

To correct this error

1. Check that references to an object property or method have valid object qualifier. Specify an object qualifier if you didn't provide one.
2. Check the spelling of the object qualifier and make sure the object is visible in the part of the program in which you are referencing it.
3. If a path is supplied to a host application's **File Open** command, check that the arguments in it are correct.
4. Check the object's documentation and make sure the action is valid.

See also

- [Error Types](#)

Object variable or With block variable not set

8/27/2019 • 2 minutes to read • [Edit Online](#)

An invalid object variable is being referenced. This error can occur for several reasons:

- A variable was declared without specifying a type. If a variable is declared without specifying a type, it defaults to type `Object`.

For example, a variable declared with `Dim x` would be of type `Object`; a variable declared with `Dim x As String` would be of type `String`.

TIP

The `Option Strict` statement disallows implicit typing that results in an `Object` type. If you omit the type, a compile-time error will occur. See [Option Strict Statement](#).

- You are attempting to reference an object that has been set to `Nothing`.
- You are attempting to access an element of an array variable that wasn't properly declared.

For example, an array declared as `products() As String` will trigger the error if you try to reference an element of the array `products(3) = "Widget"`. The array has no elements and is treated as an object.

- You are attempting to access code within a `With...End With` block before the block has been initialized. A `With...End With` block must be initialized by executing the `With` statement entry point.

NOTE

In earlier versions of Visual Basic or VBA this error was also triggered by assigning a value to a variable without using the `Set` keyword (`x = "name"` instead of `Set x = "name"`). The `Set` keyword is no longer valid in Visual Basic .Net.

To correct this error

1. Set `Option Strict` to `On` by adding the following code to the beginning of the file:

```
Option Strict On
```

When you run the project, a compiler error will appear in the **Error List** for any variable that was specified without a type.

2. If you don't want to enable `Option Strict`, search your code for any variables that were specified without a type (`Dim x` instead of `Dim x As String`) and add the intended type to the declaration.
3. Make sure you aren't referring to an object variable that has been set to `Nothing`. Search your code for the keyword `Nothing`, and revise your code so that the object isn't set to `Nothing` until after you have referenced it.
4. Make sure that any array variables are dimensioned before you access them. You can either assign a dimension when you first create the array (`Dim x(5) As String` instead of `Dim x() As String`), or use the `ReDim` keyword to set the dimensions of the array before you first access it.

5. Make sure your `With` block is initialized by executing the `With` statement entry point.

See also

- [Object Variable Declaration](#)
- [ReDim Statement](#)
- [With...End With Statement](#)

Operator declaration must be one of: +,-,*,,/,^, &, Like, Mod, And, Or, Xor, Not, <<, >>...

4/9/2019 • 2 minutes to read • [Edit Online](#)

You can declare only an operator that is eligible for overloading. The following table lists the operators you can declare.

TYPE	OPERATORS
Unary	+ , - , IsFalse , IsTrue , Not
Binary	+ , - , * , / , \ , & , ^ , >> , << , = , <> , > , >= , < , <= , And , Like , Mod , Or , Xor
Conversion (unary)	CType

Note that the `=` operator in the binary list is the comparison operator, not the assignment operator.

Error ID: BC33000

To correct this error

1. Select an operator from the set of overloadable operators.
2. If you need the functionality of overloading an operator that you cannot overload directly, create a `Function` procedure that takes the appropriate parameters and returns the appropriate value.

See also

- [Operator Statement](#)
- [Operator Procedures](#)
- [How to: Define an Operator](#)
- [How to: Define a Conversion Operator](#)
- [Function Statement](#)

'Optional' expected

4/9/2019 • 2 minutes to read • [Edit Online](#)

An optional argument in a procedure declaration is followed by a required argument. Every argument following an optional argument must also be optional.

Error ID: BC30202

To correct this error

1. If the argument is intended to be required, move it to precede the first optional argument in the argument list.
2. If the argument is intended to be optional, use the `Optional` keyword.

See also

- [Optional Parameters](#)

Optional parameters must specify a default value

10/18/2019 • 2 minutes to read • [Edit Online](#)

Optional parameters must provide default values that can be used if no parameter is supplied by a calling procedure.

Error ID: BC30812

To correct this error

Specify default values for optional parameters; for example:

```
Sub Proc1(ByVal X As Integer,  
          Optional ByVal Y As String = "Default Value")  
    MsgBox("Default argument is: " & Y)  
End Sub
```

See also

- [Optional](#)

Ordinal is not valid

4/28/2019 • 2 minutes to read • [Edit Online](#)

Your call to a dynamic-link library (DLL) indicated to use a number instead of a procedure name, using the `#num` syntax. This error has the following possible causes:

- An attempt to convert the `#num` expression to an ordinal failed.
- The `#num` specified does not specify any function in the DLL.
- A type library has an invalid declaration resulting in internal use of an invalid ordinal number.

To correct this error

1. Make sure the expression represents a valid number, or call the procedure by name.
2. Make sure `#num` identifies a valid function in the DLL.
3. Isolate the procedure call causing the problem by commenting out the code. Write a `Declare` statement for the procedure, and report the problem to the type library vendor.

See also

- [Declare Statement](#)

Out of memory (Visual Basic Compiler Error)

4/28/2019 • 2 minutes to read • [Edit Online](#)

More memory was required than is available.

Error ID: BC2004

To correct this error

- Close unnecessary applications, documents and source files.
- Eliminate unnecessary controls and forms so fewer are loaded at one time
- Reduce the number of `Public` variables.
- Check available disk space.
- Increase the available RAM by installing additional memory or reallocating memory.
- Make sure that memory is freed when it is no longer needed.

See also

- [Error Types](#)

Out of stack space (Visual Basic)

4/9/2019 • 2 minutes to read • [Edit Online](#)

The stack is a working area of memory that grows and shrinks dynamically with the demands of your executing program. Its limits have been exceeded.

To correct this error

1. Check that procedures are not nested too deeply.
2. Make sure recursive procedures terminate properly.
3. If local variables require more local variable space than is available, try declaring some variables at the module level. You can also declare all variables in the procedure static by preceding the `Property`, `Sub`, or `Function` keyword with `Static`. Or you can use the `Static` statement to declare individual static variables within procedures.
4. Redefine some of your fixed-length strings as variable-length strings, as fixed-length strings use more stack space than variable-length strings. You can also define the string at module level where it requires no stack space.
5. Check the number of nested `DoEvents` function calls, by using the `Calls` dialog box to view which procedures are active on the stack.
6. Make sure you did not cause an "event cascade" by triggering an event that calls an event procedure already on the stack. An event cascade is similar to an unterminated recursive procedure call, but it is less obvious, since the call is made by Visual Basic rather than an explicit call in the code. Use the `Calls` dialog box to view which procedures are active on the stack.

See also

- [Memory Windows](#)

Out of string space (Visual Basic)

4/9/2019 • 2 minutes to read • [Edit Online](#)

With Visual Basic, you can use very large strings. However, the requirements of other programs and the way you work with your strings can still cause this error.

To correct this error

1. Make sure that an expression requiring temporary string creation during evaluation is not causing the error.
2. Remove any unnecessary applications from memory to create more space.

See also

- [Error Types](#)
- [String Manipulation Summary](#)

Overflow (Visual Basic Error)

4/28/2019 • 2 minutes to read • [Edit Online](#)

A literal represents a value outside the limits of the data type to which it is being assigned.

Error ID: BC30036

To correct this error

- Consult the value range for the target data type and rewrite the literal to conform to that range.

See also

- [Data Types](#)

Overflow (Visual Basic Run-Time Error)

4/9/2019 • 2 minutes to read • [Edit Online](#)

An overflow results when you attempt an assignment that exceeds the limits of the assignment's target.

To correct this error

1. Make sure that results of assignments, calculations, and data type conversions are not too large to be represented within the range of variables allowed for that type of value, and assign the value to a variable of a type that can hold a larger range of values, if necessary.
2. Make sure assignments to properties fit the range of the property to which they are made.
3. Make sure that numbers used in calculations that are coerced into integers do not have results larger than integers.

See also

- [Int32.MaxValue](#)
- [Double.MaxValue](#)
- [Data Types](#)
- [Error Types](#)

Path not found

4/28/2019 • 2 minutes to read • [Edit Online](#)

During a file-access or disk-access operation, the operating system was unable to find the specified path. The path to a file includes the drive specification plus the directories and subdirectories that must be traversed to locate the file. A path can be relative or absolute.

To correct this error

- Verify and respecify the path.

See also

- [Error Types](#)

Path/File access error

4/9/2019 • 2 minutes to read • [Edit Online](#)

During a file-access or disk-access operation, the operating system could not make a connection between the path and the file name.

To correct this error

1. Make sure the file specification is correctly formatted. A file name can contain a fully qualified (absolute) or relative path. A fully qualified path starts with the drive name (if the path is on another drive) and lists the explicit path from the root to the file. Any path that is not fully qualified is relative to the current drive and directory.
2. Make sure that you did not attempt to save a file that would replace an existing read-only file. If this is the case, change the read-only attribute of the target file, or save the file with a different file name.
3. Make sure you did not attempt to open a read-only file in sequential `Output` or `Append` mode. If this is the case, open the file in `Input` mode or change the read-only attribute of the file.
4. Make sure you did not attempt to change a Visual Basic project within a database or document.

See also

- [Error Types](#)

Permission denied (Visual Basic)

4/9/2019 • 2 minutes to read • [Edit Online](#)

An attempt was made to write to a write-protected disk or to access a locked file.

To correct this error

1. To open a write-protected file, change the write-protection attribute of the file.
2. Make sure that another process has not locked the file, and wait to open the file until the other process releases it.
3. To access the registry, check that your user permissions include this type of registry access.

See also

- [Error Types](#)

Procedure call or argument is not valid (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

Some part of the call cannot be completed.

To correct this error

- Check the permitted ranges for arguments to make sure no arrangement exceeds the permitted values.

See also

- [Error Types](#)

Property '<propertyname>' doesn't return a value on all code paths

4/28/2019 • 2 minutes to read • [Edit Online](#)

Property '<propertyname>' doesn't return a value on all code paths. A null reference exception could occur at run time when the result is used.

A property `Get` procedure has at least one possible path through its code that does not return a value.

You can return a value from a property `Get` procedure in any of the following ways:

- Assign the value to the property name and then perform an `Exit Property` statement.
- Assign the value to the property name and then perform the `End Get` statement.
- Include the value in a [Return Statement](#).

If control passes to `Exit Property` or `End Get` and you have not assigned any value to the property name, the `Get` procedure returns the default value of the property's data type. For more information, see "Behavior" in [Function Statement](#).

By default, this message is a warning. For more information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC42107

To correct this error

- Check your control flow logic and make sure you assign a value before every statement that causes a return.

It is easier to guarantee that every return from the procedure returns a value if you always use the `Return` statement. If you do this, the last statement before `End Get` should be a `Return` statement.

See also

- [Property Procedures](#)
- [Property Statement](#)
- [Get Statement](#)

Property array index is not valid

4/28/2019 • 2 minutes to read • [Edit Online](#)

The supplied value is not valid for a property array index.

To correct this error

- Check the component's documentation to make sure your index is within the valid range for the specified property.

See also

- [Arrays](#)

Property let procedure not defined and property get procedure did not return an object

4/9/2019 • 2 minutes to read • [Edit Online](#)

Certain properties, methods, and operations can only apply to `Collection` objects. You specified an operation or property that is exclusive to collections, but the object is not a collection.

To correct this error

1. Check the spelling of the object or property name, or verify that the object is a `Collection` object.
2. Look at the `Add` method used to add the object to the collection to be sure the syntax is correct and that any identifiers were spelled correctly.

See also

- [Collection](#)

Property not found

4/9/2019 • 2 minutes to read • [Edit Online](#)

This object does not support the specified property.

To correct this error

1. Check the spelling of the property's name.
2. Check the object's documentation to make sure you are not trying to access something like a "text" property when the object actually supports a "caption" or similarly named property.

See also

- [Error Types](#)

Property or method not found

4/28/2019 • 2 minutes to read • [Edit Online](#)

The referenced object method or object property is not defined.

To correct this error

- You may have misspelled the name of the object. To see what properties and methods are defined for an object, display the Object Browser. Select the appropriate object library to view a list of available properties and methods.

See also

- [Error Types](#)

Range variable <variable> hides a variable in an enclosing block, a previously defined range variable, or an implicitly declared variable in a query expression

4/28/2019 • 2 minutes to read • [Edit Online](#)

A range variable name specified in a `Select`, `From`, `Aggregate`, or `Let` clause duplicates the name of a range variable already specified previously in the query, or the name of a variable that is implicitly declared by the query, such as a field name or the name of an aggregate function.

Error ID: BC36633

To correct this error

- Ensure that all range variables in a particular query scope have unique names. You can enclose a query in parentheses to ensure that nested queries have a unique scope.

See also

- [Introduction to LINQ in Visual Basic](#)
- [From Clause](#)
- [Let Clause](#)
- [Aggregate Clause](#)
- [Select Clause](#)

Range variable name can be inferred only from a simple or qualified name with no arguments

10/18/2019 • 2 minutes to read • [Edit Online](#)

A programming element that takes one or more arguments is included in a LINQ query. The compiler is unable to infer a range variable from that programming element.

Error ID: BC36599

To correct this error

Supply an explicit variable name for the programming element, as shown in the following code:

```
Dim query = From var1 In collection1  
            Select VariableAlias= SampleFunction(var1), var1
```

See also

- [Introduction to LINQ in Visual Basic](#)
- [Select Clause](#)

Reference required to assembly '<assemblyidentity>' containing type '<typename>', but a suitable reference could not be found due to ambiguity between projects '<projectname1>' and '<projectname2>'

4/9/2019 • 2 minutes to read • [Edit Online](#)

An expression uses a type, such as a class, structure, interface, enumeration, or delegate, that is defined outside your project. However, you have project references to more than one assembly defining that type.

The cited projects produce assemblies with the same name. Therefore, the compiler cannot determine which assembly to use for the type you are accessing.

To access a type defined in another assembly, the Visual Basic compiler must have a reference to that assembly. This must be a single, unambiguous reference that does not cause circular references among projects.

Error ID: BC30969

To correct this error

1. Determine which project produces the best assembly for your project to reference. For this decision, you might use criteria such as ease of file access and frequency of updates.
2. In your project properties, add a reference to the file that contains the assembly that defines the type you are using.

See also

- [Managing references in a project](#)
- [References to Declared Elements](#)
- [Managing Project and Solution Properties](#)
- [Troubleshooting Broken References](#)

Reference required to assembly '<assemblyname>' containing the base class '<classname>'

4/28/2019 • 2 minutes to read • [Edit Online](#)

Reference required to assembly '<assemblyname>' containing the base class '<classname>'. Add one to your project.

The class is defined in a dynamic-link library (DLL) or assembly that is not directly referenced in your project. The Visual Basic compiler requires a reference to avoid ambiguity in case the class is defined in more than one DLL or assembly.

Error ID: BC30007

To correct this error

- Include the name of the unreferenced DLL or assembly in your project references.

See also

- [Managing references in a project](#)
- [Troubleshooting Broken References](#)

Requested operation is not available because the runtime library function '<function>' is not defined.

3/5/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic is attempting to make an internal call to a function in the Visual Basic Runtime (Microsoft.VisualBasic.dll) that cannot be found.

Error ID: BC35000

This error occurs in SDK-style projects (projects with a *.vbproj file that begins with the line `<Project Sdk="Microsoft.NET.Sdk">`). By default, only a subset of the Microsoft.VisualBasic.dll assembly is embedded in the application assembly, and *<function>* is not included in that subset.

To correct this error

Rather than embedding a subset of the Visual Basic Runtime in your assembly, you must compile with a reference to it. You do this by adding the following element to the `<PropertyGroup>` section of your *.vbproj file:

```
<VBRuntime>Default</VBRuntime>
```

See also

- [-vbruntime compiler option](#)

Resume without error

4/9/2019 • 2 minutes to read • [Edit Online](#)

A `Resume` statement appeared outside error-handling code, or the code jumped into an error handler even though there was no error.

To correct this error

1. Move the `Resume` statement into an error handler, or delete it.
2. Jumps to labels cannot occur across procedures, so search the procedure for the label that identifies the error handler. If you find a duplicate label specified as the target of a `GoTo` statement that isn't an `On Error GoTo` statement, change the line label to agree with its intended target.

See also

- [Resume Statement](#)
- [On Error Statement](#)

Return type of function '<procedurename>' is not CLS-compliant

5/15/2019 • 2 minutes to read • [Edit Online](#)

A `Function` procedure is marked as `<CLSCompliant(True)>` but returns a type that is marked as `<CLSCompliant(False)>`, is not marked, or does not qualify because it is a noncompliant type.

For a procedure to be compliant with the [Language Independence and Language-Independent Components \(CLS\)](#), it must use only CLS-compliant types. This applies to the types of the parameters, the return type, and the types of all its local variables.

The following Visual Basic data types are not CLS-compliant:

- [SByte Data Type](#)
- [UInteger Data Type](#)
- [ULong Data Type](#)
- [UShort Data Type](#)

When you apply the [CLSCompliantAttribute](#) to a programming element, you set the attribute's `isCompliant` parameter to either `True` or `False` to indicate compliance or noncompliance. There is no default for this parameter, and you must supply a value.

If you do not apply the [CLSCompliantAttribute](#) to an element, it is considered to be noncompliant.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC40027

To correct this error

- If the `Function` procedure must return this particular type, remove the [CLSCompliantAttribute](#). The procedure cannot be CLS-compliant.
- If the `Function` procedure must be CLS-compliant, change the return type to the closest CLS-compliant type. For example, in place of `UInteger` you might be able to use `Integer` if you do not need the value range above 2,147,483,647. If you do need the extended range, you can replace `UInteger` with `Long`.
- If you are interfacing with Automation or COM objects, keep in mind that some types have different data widths than in the .NET Framework. For example, `int` is often 16 bits in other environments. If you are returning a 16-bit integer to such a component, declare it as `Short` instead of `Integer` in your managed Visual Basic code.

'Set' accessor of property '<propertyname>' is not accessible

4/28/2019 • 2 minutes to read • [Edit Online](#)

A statement attempts to store the value of a property when it does not have access to the property's `Set` procedure.

If the [Set Statement](#) is marked with a more restrictive access level than its [Property Statement](#), an attempt to set the property value could fail in the following cases:

- The `Set` statement is marked [Private](#) and the calling code is outside the class or structure in which the property is defined.
- The `Set` statement is marked [Protected](#) and the calling code is not in the class or structure in which the property is defined, nor in a derived class.
- The `Set` statement is marked [Friend](#) and the calling code is not in the same assembly in which the property is defined.

Error ID: BC31102

To correct this error

- If you have control of the source code defining the property, consider declaring the `Set` procedure with the same access level as the property itself.
- If you do not have control of the source code defining the property, or you must restrict the `Set` procedure access level more than the property itself, try to move the statement that sets the property value to a region of code that has better access to the property.

See also

- [Property Procedures](#)
- [How to: Declare a Property with Mixed Access Levels](#)

Some subkeys cannot be deleted

4/28/2019 • 2 minutes to read • [Edit Online](#)

An attempt has been made to delete a registry key, but the operation failed because some subkeys cannot be deleted. Usually this is due to a lack of permissions.

To correct this error

- Make sure you have sufficient permissions to delete the specified subkeys.

See also

- [Microsoft.VisualBasic.MyServices.RegistryProxy](#)
- [DeleteSubKey](#)
- [RegistryPermission](#)

Statement cannot end a block outside of a line 'If' statement

4/28/2019 • 2 minutes to read • [Edit Online](#)

A single-line `If` statement contains several statements separated by colons (:), one of which is an `End` statement for a control block outside the single-line `If`. Single-line `If` statements do not use the `End If` statement.

Error ID: BC32005

To correct this error

- Move the single-line `If` statement outside the control block that contains the `End If` statement.

See also

- [If...Then...Else Statement](#)

Statement is not valid in a namespace

4/28/2019 • 2 minutes to read • [Edit Online](#)

The statement cannot appear at the level of a namespace. The only declarations allowed at namespace level are module, interface, class, delegate, enumeration, and structure declarations.

Error ID: BC30001

To correct this error

- Move the statement to a location within a module, class, interface, structure, enumeration, or delegate definition.

See also

- [Scope in Visual Basic](#)
- [Namespaces in Visual Basic](#)

Statement is not valid inside a method/multiline lambda

4/28/2019 • 2 minutes to read • [Edit Online](#)

The statement is not valid within a `Sub`, `Function`, property `Get`, or property `Set` procedure. Some statements can be placed at the module or class level. Others, such as `Option Strict`, must be at namespace level and precede all other declarations.

Error ID: BC30024

To correct this error

- Remove the statement from the procedure.

See also

- [Sub Statement](#)
- [Function Statement](#)
- [Get Statement](#)
- [Set Statement](#)

String constants must end with a double quote

4/28/2019 • 2 minutes to read • [Edit Online](#)

String constants must begin and end with quotation marks.

ErrorID: BC30648

To correct this error

- Make sure the string literal ends with a quotation mark ("). If you paste values from other text editors, make sure the pasted character is a valid quotation mark and not one of the characters that resemble it, such as "smart" or "curly" quotation marks (" or ") or two single quotation marks ('').

See also

- [Strings](#)

Structure '<structurename>' must contain at least one instance member variable or at least one instance event declaration not marked 'Custom'

4/28/2019 • 2 minutes to read • [Edit Online](#)

A structure definition does not include any nonshared variables or nonshared noncustom events.

Every structure must have either a variable or an event that applies to each specific instance (nonshared) instead of to all instances collectively ([Shared](#)). Nonshared constants, properties, and procedures do not satisfy this requirement. In addition, if there are no nonshared variables and only one nonshared event, that event cannot be a [Custom](#) event.

Error ID: BC30941

To correct this error

- Define at least one variable or event that is not [Shared](#). If you define only one event, it must be noncustom as well as nonshared.

See also

- [Structures](#)
- [How to: Declare a Structure](#)
- [Structure Statement](#)

'Sub Main' was not found in '<name>'

4/9/2019 • 2 minutes to read • [Edit Online](#)

`Sub Main` is missing, or the wrong location has been specified for it.

Error ID: BC30420

To correct this error

1. Supply the missing `Sub Main` statement, or if it exists, move it to the appropriate location in the code. For more information on `Sub Main`, see [Main Procedure in Visual Basic](#).
2. Specify the location of the project's startup object in the **Startup form** box of the **Project Designer**.

See also

- [Sub Statement](#)
- [Main Procedure in Visual Basic](#)

Sub or Function not defined (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

A `Sub` or `Function` must be defined in order to be called. Possible causes of this error include:

- Misspelling the procedure name.
- Trying to call a procedure from another project without explicitly adding a reference to that project in the **References** dialog box.
- Specifying a procedure that is not visible to the calling procedure.
- Declaring a Windows dynamic-link library (DLL) routine or Macintosh code-resource routine that is not in the specified library or code resource.

To correct this error

1. Make sure that the procedure name is spelled correctly.
2. Find the name of the project containing the procedure you want to call in the **References** dialog box. If it does not appear, click the **Browse** button to search for it. Select the check box to the left of the project name, and then click **OK**.
3. Check the name of the routine.

See also

- [Error Types](#)
- [Managing references in a project](#)
- [Sub Statement](#)
- [Function Statement](#)

Subscript out of range (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

An array subscript is not valid because it falls outside the allowable range. The lowest subscript value for a dimension is always 0, and the highest subscript value is returned by the `GetUpperBound` method for that dimension.

To correct this error

- Change the subscript so it falls within the valid range.

See also

- [Array.GetUpperBound](#)
- [Arrays](#)

TextFieldParser is unable to complete the read operation because maximum buffer size has been exceeded

4/28/2019 • 2 minutes to read • [Edit Online](#)

The operation cannot be completed because the maximum buffer size (10,000,000 bytes) has been exceeded.

To correct this error

- Make sure there are no malformed fields in the file.

See also

- [OpenTextFieldParser](#)
- [TextFieldParser](#)
- [How to: Read From Text Files with Multiple Formats](#)
- [Parsing Text Files with the TextFieldParser Object](#)

The type for variable '<variablename>' will not be inferred because it is bound to a field in an enclosing scope

7/26/2019 • 2 minutes to read • [Edit Online](#)

The type for variable '<variablename>' will not be inferred because it is bound to a field in an enclosing scope. Either change the name of '<variablename>', or use the fully qualified name (for example, 'Me.variablename' or 'MyBase.variablename').

A loop control variable in your code has the same name as a field of the class or other enclosing scope. Because the control variable is used without an `As` clause, it is bound to the field in the enclosing scope, and the compiler does not create a new variable for it or infer its type.

In the following example, `Index`, the control variable in the `For` statement, is bound to the `Index` field in the `Customer` class. The compiler does not create a new variable for the control variable `Index` or infer its type.

```
Class Customer

    ' The class has a field named Index.
    Private Index As Integer

    Sub Main()

        ' The following line will raise this warning.
        For Index = 1 To 10
            ' ...
        Next

    End Sub
End Class
```

By default, this message is a warning. For information about how to hide warnings or how to treat warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC42110

To address this warning

- Make the loop control variable local by changing its name to an identifier that is not also the name of a field of the class.

```
For I = 1 To 10
```

- Clarify that the loop control variable binds to the class field by prefixing `Me.` to the variable name.

```
For Me.Index = 1 To 10
```

- Instead of relying on local type inference, use an `As` clause to specify a type for the loop control variable.

```
For Index As Integer = 1 To 10
```

Example

The following code shows the earlier example with the first correction in place.

```
Class Customer

    ' The class has a field named Index.
    Private Index As Integer

    Sub Main()

        For I = 1 To 10
            ' ...
        Next

    End Sub
End Class
```

See also

- [Option Infer Statement](#)
- [For Each...Next Statement](#)
- [For...Next Statement](#)
- [How to: Refer to the Current Instance of an Object](#)
- [Local Type Inference](#)
- [Me, My, MyBase, and MyClass](#)

This array is fixed or temporarily locked (Visual Basic)

4/28/2019 • 2 minutes to read • [Edit Online](#)

This error has the following possible causes:

- Using `ReDim` to change the number of elements of a fixed-size array.
- Redimensioning a module-level dynamic array, in which one element has been passed as an argument to a procedure. If an element is passed, the array is locked to prevent deallocating memory for the reference parameter within the procedure.
- Attempting to assign a value to a `Variant` variable containing an array, but the `Variant` is currently locked.

To correct this error

1. Make the original array dynamic rather than fixed by declaring it with `ReDim` (if the array is declared within a procedure), or by declaring it without specifying the number of elements (if the array is declared at the module level).
2. Determine whether you really need to pass the element, since it is visible within all procedures in the module.
3. Determine what is locking the `Variant` and remedy it.

See also

- [Arrays](#)

This key is already associated with an element of this collection

4/28/2019 • 2 minutes to read • [Edit Online](#)

The specified a key for a collection member already identifies another member of the collection. A key is a string specified in the `Add` method that uniquely identifies a specific member of a collection.

To correct this error

- Use a different key for this member.

See also

- [Error Types](#)

Too many files

4/9/2019 • 2 minutes to read • [Edit Online](#)

Either more files have been created in the root directory than the operating system permits, or more files have been opened than the number specified in the **files=** setting in your CONFIG.SYS file.

To correct this error

1. If your program is opening, closing, or saving files in the root directory, change your program so that it uses a subdirectory.
2. Increase the number of files specified in your **files=** setting in your CONFIG.SYS file, and restart your computer.

See also

- [Error Types](#)

Type '<typename>' has no constructors

4/9/2019 • 2 minutes to read • [Edit Online](#)

A type does not support a call to `Sub New()`. One possible cause is a corrupted compiler or binary file.

Error ID: BC30251

To correct this error

1. If the type is in a different project or in a referenced file, reinstall the project or file.
2. If the type is in the same project, recompile the assembly containing the type.
3. If the error recurs, reinstall the Visual Basic compiler.
4. If the error persists, gather information about the circumstances and notify Microsoft Product Support Services.

See also

- [Objects and Classes](#)
- [Talk to Us](#)

Type <typename> is not CLS-compliant

5/15/2019 • 2 minutes to read • [Edit Online](#)

A variable, property, or function return is declared with a data type that is not CLS-compliant.

For an application to be compliant with the [Language Independence and Language-Independent Components](#) (CLS), it must use only CLS-compliant types.

The following Visual Basic data types are not CLS-compliant:

- [SByte Data Type](#)
- [UInteger Data Type](#)
- [ULong Data Type](#)
- [UShort Data Type](#)

Error ID: BC40041

To correct this error

- If your application needs to be CLS-compliant, change the data type of this element to the closest CLS-compliant type. For example, in place of `UInteger` you might be able to use `Integer` if you do not need the value range above 2,147,483,647. If you do need the extended range, you can replace `UInteger` with `Long`.
- If your application does not need to be CLS-compliant, you do not need to change anything. You should be aware of its noncompliance, however.

Type '<typename>' is not defined

4/28/2019 • 2 minutes to read • [Edit Online](#)

The statement has made reference to a type that has not been defined. You can define a type in a declaration statement such as `Enum`, `Structure`, `Class`, or `Interface`.

Error ID: BC30002

To correct this error

- Ensure that the type definition and its reference both use the same spelling.
- Ensure that the type definition is accessible to the reference. For example, if the type is in another module and has been declared `Private`, move the type definition to the referencing module or declare it `Public`.
- Ensure that the namespace of the type is not redefined within your project. If it is, use the `Global` keyword to fully qualify the type name. For example, if a project defines a namespace named `System`, the `System.Object` type cannot be accessed unless it is fully qualified with the `Global` keyword:
`Global.System.Object`.
- If the type is defined, but the object library or type library in which it is defined is not registered in Visual Basic, click **Add Reference** on the **Project** menu, and then select the appropriate object library or type library.
- Ensure that the type is in an assembly that is part of the targeted .NET Framework profile. For more information, see [Troubleshooting .NET Framework Targeting Errors](#).

See also

- [Namespaces in Visual Basic](#)
- [Enum Statement](#)
- [Structure Statement](#)
- [Class Statement](#)
- [Interface Statement](#)
- [Managing references in a project](#)

Type arguments could not be inferred from the delegate

4/28/2019 • 2 minutes to read • [Edit Online](#)

An assignment statement uses `AddressOf` to assign the address of a generic procedure to a delegate, but it does not supply any type arguments to the generic procedure.

Normally, when you invoke a generic type, you supply a type argument for each type parameter that the generic type defines. If you do not supply any type arguments, the compiler attempts to infer the types to be passed to the type parameters. If the context does not provide enough information for the compiler to infer the types, an error is generated.

Error ID: BC36564

To correct this error

- Specify the type arguments for the generic procedure in the `AddressOf` expression.

See also

- [Generic Types in Visual Basic](#)
- [AddressOf Operator](#)
- [Generic Procedures in Visual Basic](#)
- [Type List](#)
- [Extension Methods](#)

Type mismatch (Visual Basic)

4/9/2019 • 2 minutes to read • [Edit Online](#)

You attempted to convert a value to another type in a way that is not valid.

To correct this error

1. Check the assignment to make sure it is valid.
2. Make sure you did not pass an object to a procedure that expects a single property or value.
3. Make sure you did not use a module or project name where an expression was expected.

See also

- [Error Types](#)

Type of '<variablename>' cannot be inferred because the loop bounds and the step variable do not widen to the same type

7/26/2019 • 2 minutes to read • [Edit Online](#)

You have written a `For...Next` loop in which the compiler cannot infer a data type for the loop control variable because the following conditions are true:

- The data type of the loop control variable is not specified with an `As` clause.
- The loop bounds and step variable contain at least two data types.
- No standard conversions exist between the data types.

Therefore, the compiler cannot infer the data type of a loop's control variable.

In the following example, the step variable is a character and the loop bounds are both integers. Because there is no standard conversion between characters and integers, this error is reported.

```
Dim stepVar = "1"c
Dim m = 0
Dim n = 20

' Not valid.
' For i = 1 To 10 Step stepVar
    ' Loop processing
' Next
```

Error ID: BC30982

To correct this error

- Change the types of the loop bounds and step variable as necessary so that at least one of them is a type that the others widen to. In the preceding example, change the type of `stepVar` to `Integer`.

```
Dim stepVar = 1
```

-or-

```
Dim stepVar As Integer = 1
```

- Use explicit conversion functions to convert the loop bounds and step variable to the appropriate types. In the preceding example, apply the `Val` function to `stepVar`.

```
For i = 1 To 10 Step Val(stepVar)
    ' Loop processing
Next
```

See also

- [Val](#)
- [For...Next Statement](#)
- [Implicit and Explicit Conversions](#)
- [Local Type Inference](#)
- [Option Infer Statement](#)
- [Type Conversion Functions](#)
- [Widening and Narrowing Conversions](#)

Type of member '<membername>' is not CLS-compliant

5/14/2019 • 2 minutes to read • [Edit Online](#)

The data type specified for this member is not part of the [Language Independence and Language-Independent Components](#) (CLS). This is not an error within your component, because the .NET Framework and Visual Basic support this data type. However, another component written in strictly CLS-compliant code might not support this data type. Such a component might not be able to interact successfully with your component.

The following Visual Basic data types are not CLS-compliant:

- [SByte Data Type](#)
- [UInteger Data Type](#)
- [ULong Data Type](#)
- [UShort Data Type](#)

By default, this message is a warning. For more information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC40025

To correct this error

- If your component interfaces only with other .NET Framework components, or does not interface with any other components, you do not need to change anything.
- If you are interfacing with a component not written for the .NET Framework, you might be able to determine, either through reflection or from documentation, whether it supports this data type. If it does, you do not need to change anything.
- If you are interfacing with a component that does not support this data type, you must replace it with the closest CLS-compliant type. For example, in place of `UInteger` you might be able to use `Integer` if you do not need the value range above 2,147,483,647. If you do need the extended range, you can replace `UInteger` with `Long`.
- If you are interfacing with Automation or COM objects, keep in mind that some types have different data widths than in the .NET Framework. For example, `uint` is often 16 bits in other environments. If you are passing a 16-bit argument to such a component, declare it as `ushort` instead of `UInteger` in your managed Visual Basic code.

See also

- [Reflection](#)

Type of optional value for optional parameter <parametername> is not CLS-compliant

5/15/2019 • 2 minutes to read • [Edit Online](#)

A procedure is marked as `<CLSCompliant(True)>` but declares an [Optional](#) parameter with default value of a noncompliant type.

For a procedure to be compliant with the [Language Independence and Language-Independent Components \(CLS\)](#), it must use only CLS-compliant types. This applies to the types of the parameters, the return type, and the types of all its local variables. It also applies to the default values of optional parameters.

The following Visual Basic data types are not CLS-compliant:

- [SByte Data Type](#)
- [UInteger Data Type](#)
- [ULong Data Type](#)
- [UShort Data Type](#)

When you apply the [CLSCompliantAttribute](#) attribute to a programming element, you set the attribute's `isCompliant` parameter to either `True` or `False` to indicate compliance or noncompliance. There is no default for this parameter, and you must supply a value.

If you do not apply [CLSCompliantAttribute](#) to an element, it is considered to be noncompliant.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC40042

To correct this error

- If the optional parameter must have a default value of this particular type, remove [CLSCompliantAttribute](#). The procedure cannot be CLS-compliant.
- If the procedure must be CLS-compliant, change the type of this default value to the closest CLS-compliant type. For example, in place of `UInteger` you might be able to use `Integer` if you do not need the value range above 2,147,483,647. If you do need the extended range, you can replace `UInteger` with `Long`.
- If you are interfacing with Automation or COM objects, keep in mind that some types have different data widths than in the .NET Framework. For example, `int` is often 16 bits in other environments. If you are accepting a 16-bit integer from such a component, declare it as `short` instead of `Integer` in your managed Visual Basic code.

Type of parameter '<parametername>' is not CLS-compliant

5/15/2019 • 2 minutes to read • [Edit Online](#)

A procedure is marked as `<CLSCompliant(True)>` but declares a parameter with a type that is marked as `<CLSCompliant(False)>`, is not marked, or does not qualify because it is a noncompliant type.

For a procedure to be compliant with the [Language Independence and Language-Independent Components \(CLS\)](#), it must use only CLS-compliant types. This applies to the types of the parameters, the return type, and the types of all its local variables.

The following Visual Basic data types are not CLS-compliant:

- [SByte Data Type](#)
- [UInteger Data Type](#)
- [ULong Data Type](#)
- [UShort Data Type](#)

When you apply the [CLSCompliantAttribute](#) to a programming element, you set the attribute's `isCompliant` parameter to either `True` or `False` to indicate compliance or noncompliance. There is no default for this parameter, and you must supply a value.

If you do not apply the [CLSCompliantAttribute](#) to an element, it is considered to be noncompliant.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC40028

To correct this error

- If the procedure must take a parameter of this particular type, remove the [CLSCompliantAttribute](#). The procedure cannot be CLS-compliant.
- If the procedure must be CLS-compliant, change the type of this parameter to the closest CLS-compliant type. For example, in place of `UInteger` you might be able to use `Integer` if you do not need the value range above 2,147,483,647. If you do need the extended range, you can replace `UInteger` with `Long`.
- If you are interfacing with Automation or COM objects, keep in mind that some types have different data widths than in the .NET Framework. For example, `int` is often 16 bits in other environments. If you are accepting a 16-bit integer from such a component, declare it as `short` instead of `Integer` in your managed Visual Basic code.

Type parameters cannot be used as qualifiers

10/10/2019 • 2 minutes to read • [Edit Online](#)

A programming element is qualified with a qualification string that includes a type parameter.

A type parameter represents a requirement for a type that is to be supplied when the generic type is constructed. It does not represent a specific defined type. A qualification string must include only elements that are defined at compile time.

The following code can generate this error:

```
Public Function CheckText(Of c As System.Windows.Forms.Control)()
    badText As String) As Boolean

    Dim saveText As c.Text
    ' Insert code to look for badText within saveText.
End Function
```

Error ID: BC32098

To correct this error

1. Remove the type parameter from the qualification string, or replace it with a defined type.
2. If you need to use a constructed type to locate the programming element being qualified, you must use additional program logic.

See also

- [References to Declared Elements](#)
- [Generic Types in Visual Basic](#)
- [Type List](#)

Unable to create strong-named assembly from key file '<filename>': <error>

4/9/2019 • 2 minutes to read • [Edit Online](#)

A strong-named assembly could not be created from the specified key file.

Error ID: BC31026

To correct this error

1. Verify that the correct key file has been specified, and that it is not locked by another application.

See also

- [Sn.exe \(Strong Name Tool\)](#)

Unable to embed resource file '<filename>': <error message>

4/9/2019 • 2 minutes to read • [Edit Online](#)

The Visual Basic compiler calls the Assembly Linker (Al.exe, also known as Alink) to generate an assembly with a manifest. The linker has reported an error embedding a native COM+ resource file directly into the assembly.

Error ID: BC30143

To correct this error

1. Examine the quoted error message and consult the topic [Al.exe](#) for further explanation and advice.
2. If the error persists, gather information about the circumstances and notify Microsoft Product Support Services.

See also

- [Al.exe](#)
- [Talk to Us](#)

Unable to emit assembly: <error message>

9/13/2019 • 2 minutes to read • [Edit Online](#)

The Visual Basic compiler calls the Assembly Linker (`Al.exe`, also known as Alink) to generate an assembly with a manifest, and the linker reports an error in the emission stage of creating the assembly.

Error ID: BC30145

To correct this error

1. Examine the quoted error message and consult the topic [Al.exe](#) for further explanation and advice.
2. Try signing the assembly manually, using either the [Al.exe](#) or the [Sn.exe \(Strong Name Tool\)](#).
3. If the error persists, gather information about the circumstances and notify Microsoft Product Support Services.

To sign the assembly manually

1. Use the [Sn.exe \(Strong Name Tool\)](#) to create a public/private key pair file.
This file has an `.snk` extension.
2. Delete the COM reference that is generating the error from your project.
3. Open the [Developer Command Prompt for Visual Studio](#).

In Windows 10, enter **Developer command prompt** into the search box on the task bar. Then, select **Developer Command Prompt for VS 2017** from the results list.

4. Change the directory to the directory where you want to place your assembly wrapper.
5. Enter the following command:

```
tlbimp <path to COM reference file> /out:<output assembly name> /keyfile:<path to .snk file>
```

An example of the actual command you might enter is:

```
tlbimp c:\windows\system32\msi.dll /out:Interop.WindowsInstaller.dll /keyfile:"c:\documents and settings\mykey.snk"
```

TIP

Use double quotation marks if a path or file contains spaces.

6. In Visual Studio, add a .NET Assembly reference to the file you just created.

See also

- [Al.exe](#)
- [Sn.exe \(Strong Name Tool\)](#)
- [How to: Create a Public-Private Key Pair](#)

- [Talk to Us](#)

Unable to find required file '<filename>'

4/28/2019 • 2 minutes to read • [Edit Online](#)

A file that is required by Visual Studio is missing or damaged.

Error ID: BC30655

To correct this error

- Reinstall Visual Studio.

See also

- [Talk to Us](#)

Unable to get serial port names because of an internal system error

4/9/2019 • 2 minutes to read • [Edit Online](#)

An internal error occurred when the `My.Computer.Ports.SerialPortNames` property was called.

To correct this error

1. See [Debugger Basics](#) for more troubleshooting information.
2. Note the circumstances under which the error occurred, and call Microsoft Product Support Services.

See also

- [SerialPortNames](#)
- [Debugger Basics](#)
- [Talk to Us](#)

Unable to link to resource file '<filename>': <error message>

4/9/2019 • 2 minutes to read • [Edit Online](#)

The Visual Basic compiler calls the Assembly Linker (Al.exe, also known as Alink) to generate an assembly with a manifest. The linker has reported an error linking to a native COM+ resource file from the assembly.

Error ID: BC30144

To correct this error

1. Examine the quoted error message and consult the topic [Al.exe](#) for further explanation and advice.
2. If the error persists, gather information about the circumstances and notify Microsoft Product Support Services.

See also

- [Al.exe](#)
- [Talk to Us](#)

Unable to load information for class '<classname>'

4/9/2019 • 2 minutes to read • [Edit Online](#)

A reference was made to a class that is not available.

Error ID: BC30712

To correct this error

1. Verify that the class is defined and that you spelled the name correctly.
2. Try accessing one of the members declared in the module. In some cases, the debugging environment cannot locate members because the modules where they are declared have not been loaded yet.

See also

- [Debugging in Visual Studio](#)

Unable to write output to memory

4/9/2019 • 2 minutes to read • [Edit Online](#)

There was a problem writing output to memory.

Error ID: BC31020

To correct this error

1. Compile the program again to see if the error reoccurs.
2. If the error continues, save your work and restart Visual Studio.
3. If the error recurs, reinstall Visual Basic.
4. If the error persists after reinstallation, notify Microsoft Product Support Services.

See also

- [Talk to Us](#)

Unable to write temporary file because temporary path is not available

4/9/2019 • 2 minutes to read • [Edit Online](#)

Visual Basic could not determine the path where temporary files are stored.

Error ID: BC30698

To correct this error

1. Restart Visual Studio.
2. If the problem persists, reinstall Visual Studio.

See also

- [Talk to Us](#)

Unable to write to output file '<filename>': <error>

4/28/2019 • 2 minutes to read • [Edit Online](#)

There was a problem creating the file.

An output file cannot be opened for writing. The file (or the folder containing the file) may be opened for exclusive use by another process, or it may have its read-only attribute set.

Common situations where a file is opened exclusively are:

- The application is already running and using its files. To solve this problem, make sure that the application is not running.
- Another application has opened the file. To solve this problem, make sure that no other application is accessing the files. It is not always obvious which application is accessing your files; in that case, restarting the computer might be the easiest way to terminate the application.

If even one of the project output files is marked as read-only, this exception will be thrown.

Error ID: BC31019

To correct this error

1. Compile the program again to see if the error recurs.
2. If the error continues, save your work and restart Visual Studio.
3. If the error continues, restart the computer.
4. If the error recurs, reinstall Visual Basic.
5. If the error persists after reinstallation, notify Microsoft Product Support Services.

To check file attributes in File Explorer

1. Open the folder you are interested in.
2. Click the **Views** icon and choose **Details**.
3. Right-click the column header, and choose **Attributes** from the drop-down list.

To change the attributes of a file or folder

1. In **File Explorer**, right-click the file or folder and choose **Properties**.
2. In the **Attributes** section of the **General** tab, clear the **Read-only** box.
3. Press **OK**.

See also

- [Talk to Us](#)

Underlying type <typename> of Enum is not CLS-compliant

5/14/2019 • 2 minutes to read • [Edit Online](#)

The data type specified for this enumeration is not part of the [Language Independence and Language-Independent Components](#) (CLS). This is not an error within your component, because the .NET Framework and Visual Basic support this data type. However, another component written in strictly CLS-compliant code might not support this data type. Such a component might not be able to interact successfully with your component.

The following Visual Basic data types are not CLS-compliant:

- [SByte Data Type](#)
- [UInteger Data Type](#)
- [ULong Data Type](#)
- [UShort Data Type](#)

By default, this message is a warning. For more information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC40032

To correct this error

- If your component interfaces only with other .NET Framework components, or does not interface with any other components, you do not need to change anything.
- If you are interfacing with a component not written for the .NET Framework, you might be able to determine, either through reflection or from documentation, whether it supports this data type. If it does, you do not need to change anything.
- If you are interfacing with a component that does not support this data type, you must replace it with the closest CLS-compliant type. For example, in place of `UInteger` you might be able to use `Integer` if you do not need the value range above 2,147,483,647. If you do need the extended range, you can replace `UInteger` with `Long`.
- If you are interfacing with Automation or COM objects, keep in mind that some types have different data widths than in the .NET Framework. For example, `uint` is often 16 bits in other environments. If you are passing a 16-bit argument to such a component, declare it as `ushort` instead of `UInteger` in your managed Visual Basic code.

See also

- [Reflection \(Visual Basic\)](#)
- [Reflection](#)

Using the iteration variable in a lambda expression may have unexpected results

4/28/2019 • 2 minutes to read • [Edit Online](#)

Using the iteration variable in a lambda expression may have unexpected results. Instead, create a local variable within the loop and assign it the value of the iteration variable.

This warning appears when you use a loop iteration variable in a lambda expression that is declared inside the loop. For example, the following example causes the warning to appear.

```
For i As Integer = 1 To 10
    ' The warning is given for the use of i.
    Dim exampleFunc As Func(Of Integer) = Function() i
Next
```

The following example shows the unexpected results that might occur.

```
Module Module1
    Sub Main()
        Dim array1 As Func(Of Integer)() = New Func(Of Integer)(4) {}

        For i As Integer = 0 To 4
            array1(i) = Function() i
        Next

        For Each funcElement In array1
            System.Console.WriteLine(funcElement())
        Next

    End Sub
End Module
```

The `For` loop creates an array of lambda expressions, each of which returns the value of the loop iteration variable `i`. When the lambda expressions are evaluated in the `For Each` loop, you might expect to see 0, 1, 2, 3, and 4 displayed, the successive values of `i` in the `For` loop. Instead, you see the final value of `i` displayed five times:

5
5
5
5
5

By default, this message is a warning. For more information about hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC42324

To correct this error

- Assign the value of the iteration variable to a local variable, and use the local variable in the lambda expression.

```
Module Module1
Sub Main()
    Dim array1 As Func(Of Integer)() = New Func(Of Integer)(4) {}

    For i As Integer = 0 To 4
        Dim j = i
        array1(i) = Function() j
    Next

    For Each funcElement In array1
        System.Console.WriteLine(funcElement())
    Next

End Sub
End Module
```

See also

- [Lambda Expressions](#)

Value of type '<typename1>' cannot be converted to '<typename2>'

4/28/2019 • 2 minutes to read • [Edit Online](#)

Value of type '<typename1>' cannot be converted to '<typename2>'. Type mismatch could be due to the mixing of a file reference with a project reference to assembly '<assemblyname>'. Try replacing the file reference to '<filepath>' in project '<projectname1>' with a project reference to '<projectname2>'.

In a situation where a project makes both a project reference and a file reference, the compiler cannot guarantee that one type can be converted to another.

The following pseudo-code illustrates a situation that can generate this error.

```
' ===== Visual Basic project P1 =====

' P1 makes a PROJECT REFERENCE to project P2

' and a FILE REFERENCE to project P3.

Public commonObject As P3.commonClass

commonObject = P2.getCommonClass()

' ===== Visual Basic project P2 =====

' P2 makes a PROJECT REFERENCE to project P3

Public Function getCommonClass() As P3.commonClass

Return New P3.commonClass

End Function

' ===== Visual Basic project P3 =====

Public Class commonClass

End Class
```

Project `P1` makes an indirect project reference through project `P2` to project `P3`, and also a direct file reference to `P3`. The declaration of `commonObject` uses the file reference to `P3`, while the call to `P2.getCommonClass` uses the project reference to `P3`.

The problem in this situation is that the file reference specifies a file path and name for the output file of `P3` (typically `p3.dll`), while the project references identify the source project (`P3`) by project name. Because of this, the compiler cannot guarantee that the type `P3.commonClass` comes from the same source code through the two different references.

This situation typically occurs when project references and file references are mixed. In the preceding illustration, the problem would not occur if `P1` made a direct project reference to `P3` instead of a file reference.

Error ID: BC30955

To correct this error

- Change the file reference to a project reference.

See also

- [Type Conversions in Visual Basic](#)
- [Managing references in a project](#)

Value of type '<typename1>' cannot be converted to '<typename2>' (Multiple file references)

4/28/2019 • 2 minutes to read • [Edit Online](#)

Value of type '<typename1>' cannot be converted to '<typename2>'. Type mismatch could be due to mixing a file reference to '<filepath1>' in project '<projectname1>' with a file reference to '<filepath2>' in project '<projectname2>'. If both assemblies are identical, try replacing these references so both references are from the same location.

In a situation where a project makes more than one file reference to an assembly, the compiler cannot guarantee that one type can be converted to another.

Each file reference specifies a file path and name for the output file of a project (typically a DLL file). The compiler cannot guarantee that the output files come from the same source, or that they represent the same version of the same assembly. Therefore, it cannot guarantee that the types in the different references are the same type, or even that one can be converted to the other.

You can use a single file reference if you know that the referenced assemblies have the same assembly identity. The *assembly identity* includes the assembly's name, version, public key if any, and culture. This information uniquely identifies the assembly.

Error ID: BC30961

To correct this error

- If the referenced assemblies have the same assembly identity, then remove or replace one of the file references so that there is only a single file reference.
- If the referenced assemblies do not have the same assembly identity, then change your code so that it does not attempt to convert a type in one to a type in the other.

See also

- [Type Conversions in Visual Basic](#)
- [Managing references in a project](#)

Value of type 'type1' cannot be converted to 'type2'

4/28/2019 • 2 minutes to read • [Edit Online](#)

Value of type 'type1' cannot be converted to 'type2'. You can use the 'Value' property to get the string value of the first element of '<parentElement>'.

An attempt has been made to implicitly cast an XML literal to a specific type. The XML literal cannot be implicitly cast to the specified type.

Error ID: BC31194

To correct this error

- Use the `Value` property of the XML literal to reference its value as a `String`. Use the `CType` function, another type conversion function, or the `Convert` class to cast the value as the specified type.

See also

- [Convert](#)
- [Type Conversion Functions](#)
- [XML Literals](#)
- [XML](#)

Variable '<variablename>' hides a variable in an enclosing block

9/28/2019 • 2 minutes to read • [Edit Online](#)

A variable enclosed in a block has the same name as another local variable.

Error ID: BC30616

To correct this error

- Rename the variable in the enclosed block so that it is not the same as any other local variables. For example:

```
Dim a, b, x As Integer
If a = b Then
    Dim y As Integer = 20 ' Uniquely named block variable.
End If
```

- A common cause for this error is the use of `Catch e As Exception` inside an event handler. If this is the case, name the `Catch` block variable `ex` rather than `e`.
- Another common source of this error is an attempt to access a local variable declared within a `Try` block in a separate `Catch` block. To correct this, declare the variable outside the `Try...Catch...Finally` structure.

See also

- [Try...Catch...Finally Statement](#)
- [Variable Declaration](#)

Variable '<variablename>' is used before it has been assigned a value

4/28/2019 • 2 minutes to read • [Edit Online](#)

Variable '<variablename>' is used before it has been assigned a value. A null reference exception could result at run time.

An application has at least one possible path through its code that reads a variable before any value is assigned to it.

If a variable has never been assigned a value, it holds the default value for its data type. For a reference data type, that default value is [Nothing](#). Reading a reference variable that has a value of [Nothing](#) can cause a [NullReferenceException](#) in some circumstances.

By default, this message is a warning. For more information on hiding warnings or treating warnings as errors, see [Configuring Warnings in Visual Basic](#).

Error ID: BC42104

To correct this error

- Check your control flow logic and make sure the variable has a valid value before control passes to any statement that reads it.
- One way to guarantee that the variable always has a valid value is to initialize it as part of its declaration. See "Initialization" in [Dim Statement](#).

See also

- [Dim Statement](#)
- [Variable Declaration](#)
- [Troubleshooting Variables](#)

Variable uses an Automation type not supported in Visual Basic

3/5/2019 • 2 minutes to read • [Edit Online](#)

You tried to use a variable defined in a type library or object library that has a data type not supported by Visual Basic.

To correct this error

- Use a variable of a type recognized by Visual Basic.
-or-
- If you encounter this error while using `FileGet` or `FileGetObject`, make sure the file you are trying to use was written to with `FilePut` or `FilePutObject`.

See also

- [Data Types](#)

XML axis properties do not support late binding

4/28/2019 • 2 minutes to read • [Edit Online](#)

An XML axis property has been referenced for an untyped object.

Error ID: BC31168

To correct this error

- Ensure that the object is a strong-typed [XElement](#) object before referencing the XML axis property.

See also

- [XML Axis Properties](#)
- [XML](#)

XML comment exception must have a 'cref' attribute

10/15/2019 • 2 minutes to read • [Edit Online](#)

The `<exception>` tag provides a way to document the exceptions that may be thrown by a method. The required `cref` attribute designates the name of a member, which is checked by the documentation generator. If the member exists, it is translated to the canonical element name in the documentation file.

Error ID: BC42319

To correct this error

Add the `cref` attribute to the exception as follows:

```
<exception cref="member">description</exception>
```

See also

- [<exception>](#)
- [How to: Create XML Documentation](#)
- [XML Comment Tags](#)

XML entity references are not supported

4/28/2019 • 2 minutes to read • [Edit Online](#)

An entity reference (for example, `•`) that is not defined in the XML 1.0 specification is included as a value for an XML literal. Only `&`, `"`, `<`, `>`, and `'` XML entity references are supported in XML literals.

Error ID: BC31180

To correct this error

- Remove the unsupported entity reference.

See also

- [XML Literals and the XML 1.0 Specification](#)
- [XML Literals](#)
- [XML](#)

XML literals and XML properties are not supported in embedded code within ASP.NET

4/28/2019 • 2 minutes to read • [Edit Online](#)

XML literals and XML properties are not supported in embedded code within ASP.NET. To use XML features, move the code to code-behind.

An XML literal or XML axis property is defined within embedded code (`<%= %>`) in an ASP.NET file.

Error ID: BC31200

To correct this error

- Move the code that includes the XML literal or XML axis property to an ASP.NET code-behind file.

See also

- [XML Literals](#)
- [XML Axis Properties](#)
- [XML](#)

XML namespace URI

`http://www.w3.org/XML/1998/namespace`; can be bound only to 'xmlns'

1/30/2019 • 2 minutes to read • [Edit Online](#)

The URI `http://www.w3.org/XML/1998/namespace` is used in an XML namespace declaration. This URI is a reserved namespace and cannot be included in an XML namespace declaration.

Error ID: BC31183

To correct this error

Remove the XML namespace declaration or replace the URI `http://www.w3.org/XML/1998/namespace` with a valid namespace URI.

See also

- [Imports Statement \(XML Namespace\)](#)
- [XML Literals](#)
- [XML](#)

Reference (Visual Basic)

5/14/2019 • 2 minutes to read • [Edit Online](#)

This section provides links to reference information about various aspects of Visual Basic programming.

In This Section

[Visual Basic Language Reference](#)

Provides reference information for various aspects of the Visual Basic language.

[Visual Basic Command-Line Compiler](#)

Provides links to information about the command-line compiler, which provides an alternative to compiling programs from the Visual Studio IDE.

[.NET Framework Reference Information](#)

Provides links to information on working with the .NET Framework class library.

[Visual Basic Language Specification](#)

Provides links to the complete Visual Basic language specification, which contains detailed information on all aspects of the language.

Related Sections

[General User Interface Elements \(Visual Studio\)](#)

Contains topics for dialog boxes and windows used in Visual Studio.

[XML Tools in Visual Studio](#)

Provides links to topics on various XML tools available in Visual Studio.

[Automation and Extensibility Reference](#)

Provides links to topics covering automation and extensibility in Visual Studio, for both shared and language-specific components.

Visual Basic command-line compiler

10/24/2018 • 2 minutes to read • [Edit Online](#)

The Visual Basic command-line compiler provides an alternative to compiling programs from within the Visual Studio integrated development environment (IDE). This section contains descriptions for the Visual Basic compiler options.

Every compiler option is available in two forms: **-option** and **/option**. The documentation only shows the -option form.

In this section

[Building from the Command Line](#)

Describes the Visual Basic command-line compiler, which is provided as an alternative to compiling programs from within the Visual Studio IDE.

[Visual Basic Compiler Options Listed Alphabetically](#)

Lists compiler options in an alphabetical table

[Visual Basic Compiler Options Listed by Category](#)

Presents compiler options in functional groups.

Related sections

[Visual Basic Guide](#)

The starting point for the Visual Basic documentation.

Building from the Command Line (Visual Basic)

8/27/2019 • 2 minutes to read • [Edit Online](#)

A Visual Basic project is made up of one or more separate source files. During the process known as compilation, these files are brought together into one package—a single executable file that can be run as an application.

Visual Basic provides a command-line compiler as an alternative to compiling programs from within the Visual Studio integrated development environment (IDE). The command-line compiler is designed for situations in which you do not require the full set of features in the IDE—for example, when you are using or writing for computers with limited system memory or storage space.

To compile source files from within the Visual Studio IDE, choose the **Build** command from the **Build** menu.

TIP

When you build project files by using the Visual Studio IDE, you can display information about the associated **vbc** command and its switches in the output window. To display this information, open the [Options Dialog Box, Projects and Solutions, Build and Run](#), and then set the **MSBuild project build output verbosity** to **Normal** or a higher level of verbosity. For more information, see [How to: View, Save, and Configure Build Log Files](#).

You can compile project (.vbproj) files at a command prompt by using MSBuild. For more information, see [Command-Line Reference](#) and [Walkthrough: Using MSBuild](#).

In This Section

[How to: Invoke the Command-Line Compiler](#)

Describes how to invoke the command-line compiler at the MS-DOS prompt or from a specific subdirectory.

[Sample Compilation Command Lines](#)

Provides a list of sample command lines that you can modify for your own use.

Related Sections

[Visual Basic Command-Line Compiler](#)

Provides lists of compiler options, organized alphabetically or by purpose.

[Conditional Compilation](#)

Describes how to compile particular sections of code.

[Building and Cleaning Projects and Solutions in Visual Studio](#)

Describes how to organize what will be included in different builds, choose project properties, and ensure that projects build in the correct order.

How to: Invoke the Command-Line Compiler (Visual Basic)

9/17/2019 • 2 minutes to read • [Edit Online](#)

You can invoke the command-line compiler by typing the name of its executable file into the command line, also known as the MS-DOS prompt. If you compile from the default Windows Command Prompt, you must type the fully qualified path to the executable file. To override this default behavior, you can either use the Developer Command Prompt for Visual Studio, or modify the PATH environment variable. Both allow you to compile from any directory by simply typing the compiler name.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

To invoke the compiler using the Developer Command Prompt for Visual Studio

1. Open the Visual Studio Tools program folder within the Microsoft Visual Studio program group.
2. You can use the Developer Command Prompt for Visual Studio to access the compiler from any directory on your machine, if Visual Studio is installed.
3. Invoke the Developer Command Prompt for Visual Studio.
4. At the command line, type `vbc.exe sourceFileName` and then press ENTER.

For example, if you stored your source code in a directory called `SourceFiles`, you would open the Command Prompt and type `cd SourceFiles` to change to that directory. If the directory contained a source file named `Source.vb`, you could compile it by typing `vbc.exe Source.vb`.

To set the PATH environment variable to the compiler for the Windows Command Prompt

1. Use the Windows Search feature to find Vbc.exe on your local disk.

The exact name of the directory where the compiler is located depends on the location of the Windows directory and the version of the ".NET Framework" installed. If you have more than one version of the ".NET Framework" installed, you must determine which version to use (typically the latest version).

2. From your **Start** Menu, right-click **My Computer**, and then click **Properties** from the shortcut menu.
3. Click the **Advanced** tab, and then click **Environment Variables**.
4. In the **System** variables pane, select **Path** from the list and click **Edit**.
5. In the **Edit System** Variable dialog box, move the insertion point to the end of the string in the **Variable Value** field and type a semicolon (;) followed by the full directory name found in Step 1.
6. Click **OK** to confirm your edits and close the dialog boxes.

After you change the PATH environment variable, you can run the Visual Basic compiler at the Windows Command Prompt from any directory on the computer.

To invoke the compiler using the Windows Command Prompt

1. From the **Start** menu, click on the **Accessories** folder, and then open the **Windows Command Prompt**.
2. At the command line, type `vbc.exe sourceFileName` and then press ENTER.

For example, if you stored your source code in a directory called `SourceFiles`, you would open the Command Prompt and type `cd SourceFiles` to change to that directory. If the directory contained a source file named `Source.vb`, you could compile it by typing `vbc.exe Source.vb`.

See also

- [Visual Basic Command-Line Compiler](#)
- [Conditional Compilation](#)

Sample compilation command lines (Visual Basic)

8/27/2019 • 2 minutes to read • [Edit Online](#)

As an alternative to compiling Visual Basic programs from within Visual Studio, you can compile from the command line to produce executable (.exe) files or dynamic-link library (.dll) files.

The Visual Basic command-line compiler supports a complete set of options that control input and output files, assemblies, and debug and preprocessor options. Each option is available in two interchangeable forms: `-option` and `/option`. This documentation shows only the `-option` form.

The following table lists some sample command lines you can modify for your own use.

TO	USE
Compile File.vb and create File.exe	<code>vbc -reference:Microsoft.VisualBasic.dll File.vb</code>
Compile File.vb and create File.dll	<code>vbc -target:library File.vb</code>
Compile File.vb and create My.exe	<code>vbc -out:My.exe File.vb</code>
Compile File.vb and create both a library and a reference assembly named File.dll	<code>vbc -target:library -ref:.\debug\bin\ref\file.dll File.vb</code>
Compile all Visual Basic files in the current directory, with optimizations on and the <code>DEBUG</code> symbol defined, producing File2.exe	<code>vbc -define:DEBUG=1 -optimize -out:File2.exe *.vb</code>
Compile all Visual Basic files in the current directory, producing a debug version of File2.dll without displaying the logo or warnings	<code>vbc -target:library -out:File2.dll -nowarn -nologo -debug *.vb</code>
Compile all Visual Basic files in the current directory to Something.dll	<code>vbc -target:library -out:Something.dll *.vb</code>

TIP

When you build a project by using the Visual Studio IDE, you can display information about the associated `vbc` command with its compiler options in the output window. To display this information, open the [Options Dialog Box, Projects and Solutions, Build and Run](#), and then set the **MSBuild project build output verbosity** to **Normal** or a higher level of verbosity.

See also

- [Visual Basic Command-Line Compiler](#)
- [Conditional Compilation](#)

Visual Basic compiler options listed alphabetically

9/23/2019 • 3 minutes to read • [Edit Online](#)

The Visual Basic command-line compiler is provided as an alternative to compiling programs from the Visual Studio integrated development environment (IDE). The following is a list of the Visual Basic command-line compiler options sorted alphabetically.

Every compiler option is available in two forms: **-option** and **/option**. The documentation only shows the -option form.

OPTION	PURPOSE
@ (Specify Response File)	Specifies a response file.
-?	Displays compiler options. This command is the same as specifying the -help option. No compilation occurs.
-additionalfile	Names additional files that don't directly affect code generation but may be used by analyzers for producing errors or warnings.
-addmodule	Causes the compiler to make all type information from the specified file(s) available to the project you are currently compiling.
-analyzer	Run the analyzers from this assembly (Short form: -a)
-baseaddress	Specifies the base address of a DLL.
-bugreport	Creates a file that contains information that makes it easy to report a bug.
-checksumalgorithm:<alg>	Specify the algorithm for calculating the source file checksum stored in PDB. Supported values are: SHA1 (default) or SHA256. Due to collision problems with SHA1, Microsoft recommends SHA256 or better.
-codepage	Specifies the code page to use for all source code files in the compilation.
-debug	Produces debugging information.
-define	Defines symbols for conditional compilation.
-delaysign	Specifies whether the assembly will be fully or partially signed.
-deterministic	Causes the compiler to output an assembly whose binary content is identical across compilations if inputs are identical.
-doc	Processes documentation comments to an XML file.

OPTION	PURPOSE
-errorreport	Specifies how the Visual Basic compiler should report internal compiler errors.
-filealign	Specifies where to align the sections of the output file.
-help	Displays compiler options. This command is the same as specifying the <code>-?</code> option. No compilation occurs.
-highentropyva	Indicates whether a particular executable supports high entropy Address Space Layout Randomization (ASLR).
-imports	Imports a namespace from a specified assembly.
-keycontainer	Specifies a key container name for a key pair to give an assembly a strong name.
-keyfile	Specifies a file that contains a key or key pair to give an assembly a strong name.
-langversion	Specify language version: 9 9.0 10 10.0 11 11.0.
-libpath	Specifies the location of assemblies referenced by the <code>-reference</code> option.
-linkresource	Creates a link to a managed resource.
-main	Specifies the class that contains the <code>Sub Main</code> procedure to use at startup.
-moduleassemblyname	Specifies the name of the assembly that a module will be a part of.
<code>-modulename:<string></code>	Specify the name of the source module
-netcf	Sets the compiler to target the .NET Compact Framework.
-noconfig	Do not compile with Vbc.rsp.
-nologo	Suppresses compiler banner information.
-nostdlib	Causes the compiler not to reference the standard libraries.
-nowarn	Suppresses the compiler's ability to generate warnings.
-nowin32manifest	Instructs the compiler not to embed any application manifest into the executable file.
-optimize	Enables/disables code optimization.
-optioncompare	Specifies whether string comparisons should be binary or use locale-specific text semantics.

OPTION	PURPOSE
-optionexplicit	Enforces explicit declaration of variables.
-optioninfer	Enables the use of local type inference in variable declarations.
-optionstrict	Enforces strict language semantics.
-out	Specifies an output file.
-parallel[+ -]	Specifies whether to use concurrent build (+).
-platform	Specifies the processor platform the compiler targets for the output file.
-preferreuilang	Specify the preferred output language name.
-quiet	Prevents the compiler from displaying code for syntax-related errors and warnings.
-recurse	Searches subdirectories for source files to compile.
-reference	Imports metadata from an assembly.
-refonly	Outputs only a reference assembly.
-refout	Specifies the output path of a reference assembly.
-removeintchecks	Disables integer overflow checking.
-resource	Embeds a managed resource in an assembly.
-rootnamespace	Specifies a namespace for all type declarations.
-ruleset:<file>	Specify a ruleset file that disables specific diagnostics.
-sdkpath	Specifies the location of MsCorlib.dll and Microsoft.VisualBasic.dll.
-subsystemversion	Specifies the minimum version of the subsystem that the generated executable file can use.
-target	Specifies the format of the output file.
-utf8output	Displays compiler output using UTF-8 encoding.
-vbruntime	Specifies that the compiler should compile without a reference to the Visual Basic Runtime Library, or with a reference to a specific runtime library.
-verbose	Outputs extra information during compilation.
-warnaserror	Promotes warnings to errors.

OPTION	PURPOSE
-win32icon	Inserts an .ico file into the output file.
-win32manifest	Identifies a user-defined Win32 application manifest file to be embedded into a project's portable executable (PE) file.
-win32resource	Inserts a Win32 resource into the output file.

See also

- [Visual Basic Compiler Options Listed by Category](#)
- [Manage project and solution properties](#)

@ (Specify Response File) (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Specifies a file that contains compiler options and source-code files to compile.

Syntax

```
@response_file
```

Arguments

`response_file`

Required. A file that lists compiler options or source-code files to compile. Enclose the file name in quotation marks (" ") if it contains a space.

Remarks

The compiler processes the compiler options and source-code files specified in a response file as if they had been specified on the command line.

To specify more than one response file in a compilation, specify multiple response-file options, such as the following.

```
@file1.rsp @file2.rsp
```

In a response file, multiple compiler options and source-code files can appear on one line. A single compiler-option specification must appear on one line (cannot span multiple lines). Response files can have comments that begin with the `#` symbol.

You can combine options specified on the command line with options specified in one or more response files. The compiler processes the command options as it encounters them. Therefore, command-line arguments can override previously listed options in response files. Conversely, options in a response file override options listed previously on the command line or in other response files.

Visual Basic provides the Vbc.rsp file, which is located in the same directory as the Vbc.exe file. The Vbc.rsp file is included by default unless the `-noconfig` option is used. For more information, see [-noconfig](#).

NOTE

The `@` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following lines are from a sample response file.

```
# build the first output file
-target:exe
-out:MyExe.exe
source1.vb
source2.vb
```

Example

The following example demonstrates how to use the `@` option with the response file named `File1.rsp`.

```
vbc @file1.rsp
```

See also

- [Visual Basic Command-Line Compiler](#)
- [-noconfig](#)
- [Sample Compilation Command Lines](#)

-addmodule

10/17/2019 • 2 minutes to read • [Edit Online](#)

Causes the compiler to make all type information from the specified file(s) available to the project you are currently compiling.

Syntax

```
-addmodule:fileList
```

Arguments

`fileList`

Required. Comma-delimited list of files that contain metadata but do not contain assembly manifests. File names containing spaces should be surrounded by quotation marks (" ").

Remarks

The files listed by the `fileList` parameter must be created with the `-target:module` option, or with another compiler's equivalent to `-target:module`.

All modules added with `-addmodule` must be in the same directory as the output file at run time. That is, you can specify a module in any directory at compile time, but the module must be in the application directory at run time. If it is not, you get a [TypeLoadException](#) error.

If you specify (implicitly or explicitly) any [-target \(Visual Basic\)](#) option other than `-target:module` with `-addmodule`, the files you pass to `-addmodule` become part of the project's assembly. An assembly is required to run an output file that has one or more files added with `-addmodule`.

Use [-reference \(Visual Basic\)](#) to import metadata from a file that contains an assembly.

NOTE

The `-addmodule` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code creates a module.

```
' t1.vb
' Compile with vbc /target:module t1.vb.
' Outputs t1.netmodule.

Public Class TestClass
    Public i As Integer
End Class
```

The following code imports the module's types.

```
' t2.vb
' Compile with vbc /addmodule:t1.netmodule t2.vb.
Option Strict Off

Namespace NetmoduleTest
    Module Module1
        Sub Main()
            Dim x As TestClass
            x = New TestClass
            x.i = 802
            System.Console.WriteLine(x.i)
        End Sub
    End Module
End Namespace
```

When you run `t1`, it outputs `802`.

See also

- [Visual Basic Command-Line Compiler](#)
- [-target \(Visual Basic\)](#)
- [-reference \(Visual Basic\)](#)
- [Sample Compilation Command Lines](#)

-baseaddress

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies a default base address when creating a DLL.

Syntax

```
-baseaddress:address
```

Arguments

TERM	DEFINITION
address	Required. The base address for the DLL. This address must be specified as a hexadecimal number.

Remarks

The default base address for a DLL is set by the .NET Framework.

Be aware that the lower-order word in this address is rounded. For example, if you specify 0x11110001, it is rounded to 0x11110000.

To complete the signing process for a DLL, use the `-R` option of the Strong Naming tool (Sn.exe).

This option is ignored if the target is not a DLL.

TO SET -BASEADDRESS IN THE VISUAL STUDIO IDE

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Click **Advanced**.
4. Modify the value in the **DLL base address:** box. **Note:** The **DLL base address:** box is read-only unless the target is a DLL.

See also

- [Visual Basic Command-Line Compiler](#)
- [-target \(Visual Basic\)](#)
- [Sample Compilation Command Lines](#)
- [Sn.exe \(Strong Name Tool\)\)](#)

-bugreport

10/7/2019 • 2 minutes to read • [Edit Online](#)

Creates a file that you can use when you file a bug report.

Syntax

```
-bugreport:file
```

Arguments

TERM	DEFINITION
<code>file</code>	Required. The name of the file that will contain your bug report. Enclose the file name in quotation marks (" ") if the name contains a space.

Remarks

The following information is added to `file`:

- A copy of all source-code files in the compilation.
- A list of the compiler options used in the compilation.
- Version information about your compiler, common language runtime, and operating system.
- Compiler output, if any.
- A description of the problem, for which you are prompted.
- A description of how you think the problem should be fixed, for which you are prompted.

Because a copy of all source-code files is included in `file`, you may want to reproduce the (suspected) code defect in the shortest possible program.

IMPORTANT

The `-bugreport` option produces a file that contains potentially sensitive information. This includes current time, compiler version, .NET Framework version, OS version, user name, the command-line arguments with which the compiler was run, all source code, and the binary form of any referenced assembly. This option can be accessed by specifying command-line options in the Web.config file for a server-side compilation of an ASP.NET application. To prevent this, modify the Machine.config file to disallow users from compiling on the server.

If this option is used with `-errorreport:prompt`, `-errorreport:queue`, or `-errorreport:send`, and your application encounters an internal compiler error, the information in `file` is sent to Microsoft Corporation. That information will help Microsoft engineers identify the cause of the error and may help improve the next release of Visual Basic. By default, no information is sent to Microsoft. However, when you compile an application by using `-errorreport:queue`, which is enabled by default, the application collects its error reports. Then, when the computer's administrator logs in, the error reporting system displays a pop-up window that enables the

administrator to forward to Microsoft any error reports that occurred since the logon.

NOTE

The `/bugreport` option is not available from within the Visual Studio development environment; it is available only when you compile from the command line.

Example

The following example compiles `t2.vb` and puts all bug-reporting information in the file `Problem.txt`.

```
vbc -bugreport:problem.txt t2.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [-debug \(Visual Basic\)](#)
- [-errorreport](#)
- [Sample Compilation Command Lines](#)
- [trustLevel Element for securityPolicy \(ASP.NET Settings Schema\)](#)

-codepage (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies the code page to use for all source-code files in the compilation.

Syntax

```
-codepage:id
```

Arguments

TERM	DEFINITION
<code>id</code>	Required. The compiler uses the code page specified by <code>id</code> to interpret the encoding of the source files.

Remarks

To compile source code saved with a specific encoding, you can use `-codepage` to specify which code page should be used. The `-codepage` option applies to all source-code files in your compilation. For more information, see [Character Encoding in the .NET Framework](#).

The `-codepage` option is not needed if the source-code files were saved using the current ANSI code page, Unicode, or UTF-8 with a signature. Visual Studio saves all source-code files with the current ANSI code page by default, unless the user specifies another encoding in the **Encoding** dialog box. Visual Studio uses the **Encoding** dialog box to open source-code files saved with a different code page.

NOTE

The `-codepage` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

See also

- [Visual Basic Command-Line Compiler](#)

-debug (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

Causes the compiler to generate debugging information and place it in the output file(s).

Syntax

```
-debug[+ | -]
```

or

```
-debug:[full | pdbonly]
```

Arguments

TERM	DEFINITION
<code>+ -</code>	Optional. Specifying <code>+</code> or <code>/debug</code> causes the compiler to generate debugging information and place it in a .pdb file. Specifying <code>-</code> has the same effect as not specifying <code>/debug</code> .
<code>full pdbonly</code>	Optional. Specifies the type of debugging information generated by the compiler. If you do not specify <code>/debug:pdbonly</code> , the default is <code>full</code> , which enables you to attach a debugger to the running program. The <code>pdbonly</code> argument allows source-code debugging when the program is started in the debugger, but it displays assembly-language code only when the running program is attached to the debugger.

Remarks

Use this option to create debug builds. If you do not specify `/debug`, `/debug+`, or `/debug:full`, you will be unable to debug the output file of your program.

By default, debugging information is not emitted (`/debug-`). To emit debugging information, specify `/debug` or `/debug+`.

For information on how to configure the debug performance of an application, see [Making an Image Easier to Debug](#).

TO SET -DEBUG IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Click **Advanced Compile Options**.
4. Modify the value in the **Generate Debug Info** box.

Example

The following example puts debugging information in output file `App.exe`.

```
vbc -debug -out:app.exe test.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [/bugreport](#)
- [Sample Compilation Command Lines](#)

-define (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Defines conditional compiler constants.

Syntax

```
-define:["]symbol[=value][,symbol[=value]]["]
```

or

```
-d:["]symbol[=value][,symbol[=value]]["]
```

Arguments

TERM	DEFINITION
<code>symbol</code>	Required. The symbol to define.
<code>value</code>	Optional. The value to assign <code>symbol</code> . If <code>value</code> is a string, it must be surrounded by backslash/quotations-mark sequences (\") instead of quotation marks. If no value is specified, then it is taken to be True.

Remarks

The `-define` option has an effect similar to using a `#Const` preprocessor directive in your source file, except that constants defined with `-define` are public and apply to all files in the project.

You can use symbols created by this option with the `#If ... Then ... #Else` directive to compile source files conditionally.

`-d` is the short form of `-define`.

You can define multiple symbols with `-define` by using a comma to separate symbol definitions.

TO SET /DEFINE IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Click **Advanced**.
4. Modify the value in the **Custom Constants** box.

Example

The following code defines and then uses two conditional compiler constants.

```
' Vbc /define:DEBUGMODE=True,TRAPERRORS=False test.vb
Sub mysub()
#If debugmode Then
    ' Insert debug statements here.
    MsgBox("debug mode")
#Else
    ' Insert default statements here.
#End If
End Sub
```

See also

- [Visual Basic Command-Line Compiler](#)
- [#If...Then...#Else Directives](#)
- [#Const Directive](#)
- [Sample Compilation Command Lines](#)

-delaysign

10/18/2019 • 2 minutes to read • [Edit Online](#)

Specifies whether the assembly will be fully or partially signed.

Syntax

```
-delaysign[+ | -]
```

Arguments

+ | -

Optional. Use `-delaysign-` if you want a fully signed assembly. Use `-delaysign+` if you want to place the public key in the assembly and reserve space for the signed hash. The default is `-delaysign-`.

Remarks

The `-delaysign` option has no effect unless used with [-keyfile](#) or [-keycontainer](#).

When you request a fully signed assembly, the compiler hashes the file that contains the manifest (assembly metadata) and signs that hash with the private key. The resulting digital signature is stored in the file that contains the manifest. When an assembly is delay signed, the compiler does not compute and store the signature but reserves space in the file so the signature can be added later.

For example, by using `-delaysign+`, a developer in an organization can distribute unsigned test versions of an assembly that testers can register with the global assembly cache and use. When work on the assembly is completed, the person responsible for the organization's private key can fully sign the assembly. This compartmentalization protects the organization's private key from disclosure, while allowing all developers to work on the assemblies.

See [Creating and Using Strong-Named Assemblies](#) for more information on signing an assembly.

To set `-delaysign` in the Visual Studio integrated development environment

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Signing** tab.
3. Set the value in the **Delay sign only** box.

See also

- [Visual Basic Command-Line Compiler](#)
- [-keyfile](#)
- [-keycontainer](#)
- [Sample Compilation Command Lines](#)

-deterministic

10/7/2019 • 2 minutes to read • [Edit Online](#)

Causes the compiler to produce an assembly whose byte-for-byte output is identical across compilations for identical inputs.

Syntax

```
-deterministic
```

Remarks

By default, compiler output from a given set of inputs is unique, since the compiler adds a timestamp and a GUID that is generated from random numbers. You use the `-deterministic` option to produce a *deterministic assembly*, one whose binary content is identical across compilations as long as the input remains the same.

The compiler considers the following inputs for the purpose of determinism:

- The sequence of command-line parameters.
- The contents of the compiler's .rsp response file.
- The precise version of the compiler used, and its referenced assemblies.
- The current directory path.
- The binary contents of all files explicitly passed to the compiler either directly or indirectly, including:
 - Source files
 - Referenced assemblies
 - Referenced modules
 - Resources
 - The strong name key file
 - @ response files
 - Analyzers
 - Rulesets
 - Additional files that may be used by analyzers
- The current culture (for the language in which diagnostics and exception messages are produced).
- The default encoding (or the current code page) if the encoding is not specified.
- The existence, non-existence, and contents of files on the compiler's search paths (specified, for example, by `/lib` or `/recurse`).
- The CLR platform on which the compiler is run.
- The value of `%LIBPATH%`, which can affect analyzer dependency loading.

When sources are publicly available, deterministic compilation can be used for establishing whether a binary is compiled from a trusted source. It can also be useful in a continuous build system for determining whether build steps that are dependent on changes to a binary need to be executed.

See also

- [Visual Basic Command-Line Compiler](#)

- Sample Compilation Command Lines

-doc

10/7/2019 • 2 minutes to read • [Edit Online](#)

Processes documentation comments to an XML file.

Syntax

```
-doc[+ | -]
```

or

```
-doc:file
```

Arguments

TERM	DEFINITION
+ -	Optional. Specifying +, or just -doc, causes the compiler to generate documentation information and place it in an XML file. Specifying - is the equivalent of not specifying -doc, causing no documentation information to be created.
file	Required if -doc: is used. Specifies the output XML file, which is populated with the comments from the source-code files of the compilation. If the file name contains a space, surround the name with quotation marks ("").

Remarks

The -doc option controls whether the compiler generates an XML file containing the documentation comments. If you use the -doc:file syntax, the file parameter specifies the name of the XML file. If you use -doc or -doc+, the compiler takes the XML file name from the executable file or library that the compiler is creating. If you use -doc- or do not specify the -doc option, the compiler does not create an XML file.

In source-code files, documentation comments can precede the following definitions:

- User-defined types, such as a [class](#) or [interface](#)
- Members, such as a field, [event](#), [property](#), [function](#), or [subroutine](#).

To use the generated XML file with the Visual Studio [IntelliSense](#) feature, let the file name of the XML file be the same as the assembly you want to support. Make sure the XML file is in the same directory as the assembly so that when the assembly is referenced in the Visual Studio project, the .xml file is found as well. XML documentation files are not required for IntelliSense to work for code within a project or within projects referenced by a project.

Unless you compile with /target:module, the XML file contains the tags <assembly></assembly>. These tags specify the name of the file containing the assembly manifest for the output file of the compilation.

See [XML Comment Tags](#) for ways to generate documentation from comments in your code.

TO SET -DOC IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Set the value in the **Generate XML documentation file** box.

Example

See [Documenting Your Code with XML](#) for a sample.

See also

- [Visual Basic Command-Line Compiler](#)
- [Documenting Your Code with XML](#)

-errorreport

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies how the Visual Basic compiler should report internal compiler errors.

Syntax

```
-errorreport:{ prompt | queue | send | none }
```

Remarks

This option provides a convenient way to report a Visual Basic internal compiler error (ICE) to the Visual Basic team at Microsoft. By default, the compiler sends no information to Microsoft. However, if you do encounter an internal compiler error, this option allows you to report the error to Microsoft. That information will help Microsoft engineers identify the cause and may help improve the next release of Visual Basic.

A user's ability to send reports depends on machine and user policy permissions.

The following table summarizes the effect of the `-errorreport` option.

OPTION	BEHAVIOR
<code>prompt</code>	If an internal compiler error occurs, a dialog box comes up so that you can view the exact data that the compiler collected. You can determine if there is any sensitive information in the error report and make a decision on whether to send it to Microsoft. If you decide to send it, and the machine and user policy settings allow it, the compiler sends the data to Microsoft.
<code>queue</code>	Queues the error report. When you log in with administrator privileges, you can report any failures since the last time you were logged in (you will not be prompted to send reports for failures more than once every three days). This is the default behavior when the <code>-errorreport</code> option is not specified.
<code>send</code>	If an internal compiler error occurs, and the machine and user policy settings allow it, the compiler sends the data to Microsoft. The option <code>-errorreport:send</code> attempts to automatically send error information to Microsoft if reporting is enabled by the Windows Error Reporting system settings.
<code>none</code>	If an internal compiler error occurs, it will not be collected or sent to Microsoft.

The compiler sends data that includes the stack at the time of the error, which usually includes some source code.

If `-errorreport` is used with the `-bugreport` option, then the entire source file is sent.

This option is best used with the `/bugreport` option, because it allows Microsoft engineers to more easily reproduce the error.

NOTE

The `-errorreport` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code attempts to compile `t2.vb`, and if the compiler encounters an internal compiler error, it prompts you to send the error report to Microsoft.

```
vbc -errorreport:prompt t2.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)
- [-bugreport](#)

-filealign

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies where to align the sections of the output file.

Syntax

```
-filealign:number
```

Arguments

`number`

Required. A value that specifies the alignment of sections in the output file. Valid values are 512, 1024, 2048, 4096, and 8192. These values are in bytes.

Remarks

You can use the `-filealign` option to specify the alignment of sections in your output file. Sections are blocks of contiguous memory in a Portable Executable (PE) file that contains either code or data. The `-filealign` option lets you compile your application with a nonstandard alignment; most developers do not need to use this option.

Each section is aligned on a boundary that is a multiple of the `-filealign` value. There is no fixed default. If `-filealign` is not specified, the compiler picks a default at compile time.

By specifying the section size, you can change the size of the output file. Modifying section size may be useful for programs that will run on smaller devices.

NOTE

The `-filealign` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

See also

- [Visual Basic Command-Line Compiler](#)

-help, -? (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Displays the compiler options.

Syntax

```
-help
```

or

```
-?
```

Remarks

If you include this option in a compilation, no output file is created and no compilation takes place.

NOTE

The `-help` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code displays help from the command line.

```
vbc -help
```

See also

- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)

-highentropyva (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Indicates whether a 64-bit executable or an executable that's marked by the [/platform:anycpu](#) compiler option supports high entropy Address Space Layout Randomization (ASLR).

Syntax

```
-highentropyva[+ | -]
```

Arguments

+ | -

Optional. The option is off by default or if you specify `-highentropyva-`. The option is on if you specify `-highentropyva` or `-highentropyva+`.

Remarks

If you specify this option, compatible versions of the Windows kernel can use higher degrees of entropy when the kernel randomizes the address space layout of a process as part of ASLR. If the kernel uses higher degrees of entropy, a larger number of addresses can be allocated to memory regions such as stacks and heaps. As a result, it is more difficult to guess the location of a particular memory region.

When the option is on, the target executable and any modules on which it depends must be able to handle pointer values that are larger than 4 gigabytes (GB) when those modules are running as 64-bit processes.

See also

- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)

-imports (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Imports namespaces from a specified assembly.

Syntax

```
-imports:namespaceList
```

Arguments

TERM	DEFINITION
namespaceList	Required. Comma-delimited list of namespaces to be imported.

Remarks

The `-imports` option imports any namespace defined within the current set of source files or from any referenced assembly.

The members in a namespace specified with `-imports` are available to all source-code files in the compilation. Use the [Imports Statement \(.NET Namespace and Type\)](#) to use a namespace in a single source-code file.

TO SET /IMPORTS IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **References** tab.
3. Enter the namespace name in the box beside the **Add User Import** button.
4. Click the **Add User Import** button.

Example

The following code compiles when `/imports:system.globalization` is specified. Without it, successful compilation requires either that an `Imports System.Globalization` statement be included at the beginning of the source code file, or that the property be fully qualified as `System.Globalization.CultureInfo.CurrentCulture.Name`.

```
Module Example
    Public Sub Main()
        Console.WriteLine($"The current culture is {CultureInfo.CurrentCulture.Name}")
    End Sub
End Module
```

See also

- [Visual Basic Command-Line Compiler](#)
- [References and the Imports Statement](#)
- [Sample Compilation Command Lines](#)

-keycontainer

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies a key container name for a key pair to give an assembly a strong name.

Syntax

```
-keycontainer:container
```

Arguments

TERM	DEFINITION
container	Required. Container file that contains the key. Enclose the file name in quotation marks ("") if the name contains a space.

Remarks

The compiler creates the sharable component by inserting a public key into the assembly manifest and by signing the final assembly with the private key. To generate a key file, type `sn -k file` at the command line. The `-i` option installs the key pair into a container. For more information, see [Sn.exe \(Strong Name Tool\)](#)).

If you compile with `-target:module`, the name of the key file is held in the module and incorporated into the assembly that is created when you compile an assembly with [-addmodule](#).

You can also specify this option as a custom attribute ([AssemblyKeyNameAttribute](#)) in the source code for any Microsoft intermediate language (MSIL) module.

You can also pass your encryption information to the compiler with [-keyfile](#). Use [-delaysign](#) if you want a partially signed assembly.

See [Creating and Using Strong-Named Assemblies](#) for more information on signing an assembly.

NOTE

The `-keycontainer` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code compiles source file `Input.vb` and specifies a key container.

```
vbc -keycontainer:key1 input.vb
```

See also

- [Assemblies in .NET](#)
- [Visual Basic Command-Line Compiler](#)

- [-keyfile](#)
- [Sample Compilation Command Lines](#)

-keyfile

10/18/2019 • 2 minutes to read • [Edit Online](#)

Specifies a file containing a key or key pair to give an assembly a strong name.

Syntax

```
-keyfile:file
```

Arguments

`file`

Required. File that contains the key. If the file name contains a space, enclose the name in quotation marks ("").

Remarks

The compiler inserts the public key into the assembly manifest and then signs the final assembly with the private key. To generate a key file, type `sn -k file` at the command line. For more information, see [Sn.exe \(Strong Name Tool\)](#).

If you compile with `-target:module`, the name of the key file is held in the module and incorporated into the assembly that is created when you compile an assembly with [/addmodule](#).

You can also pass your encryption information to the compiler with [-keycontainer](#). Use [-delaysign](#) if you want a partially signed assembly.

You can also specify this option as a custom attribute ([AssemblyKeyFileAttribute](#)) in the source code for any Microsoft intermediate language module.

In case both [-keyfile](#) and [-keycontainer](#) are specified (either by command-line option or by custom attribute) in the same compilation, the compiler first tries the key container. If that succeeds, then the assembly is signed with the information in the key container. If the compiler does not find the key container, it tries the file specified with [-keyfile](#). If this succeeds, the assembly is signed with the information in the key file, and the key information is installed in the key container (similar to `sn -i`) so that on the next compilation, the key container will be valid.

Note that a key file might contain only the public key.

See [Creating and Using Strong-Named Assemblies](#) for more information on signing an assembly.

NOTE

The [-keyfile](#) option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code compiles source file `Input.vb` and specifies a key file.

```
vbc -keyfile:myfile.sn input.vb
```

See also

- [Assemblies in .NET](#)
- [Visual Basic Command-Line Compiler](#)
- [-reference \(Visual Basic\)](#)
- [Sample Compilation Command Lines](#)

-langversion (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Causes the compiler to accept only syntax that is included in the specified Visual Basic language version.

Syntax

```
-langversion:version
```

Arguments

version

Required. The language version to be used during the compilation. Accepted values are 9, 10, 11, 12, 14, 15, 15.3, 15.5, default and latest.

Any of the whole numbers may also be specified using .0 as the minor version, for example, 11.0.

You can see the list of all possible values by specifying -langversion:? on the command line.

Remarks

The -langversion option specifies what syntax the compiler accepts. For example, if you specify that the language version is 9.0, the compiler generates errors for syntax that is valid only in version 10.0 and later.

You can use this option when you develop applications that target different versions of the .NET Framework. For example, if you are targeting .NET Framework 3.5, you could use this option to ensure that you do not use syntax from language version 10.0.

You can set -langversion directly only by using the command line. For more information, see [Targeting a Specific .NET Framework Version](#).

Example

The following code compiles sample.vb for Visual Basic 9.0.

```
vbc -langversion:9.0 sample.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)
- [Targeting a Specific .NET Framework Version](#)

-libpath

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies the location of referenced assemblies.

Syntax

```
-libpath:dirList
```

Arguments

TERM	DEFINITION
dirList	Required. Semicolon-delimited list of directories for the compiler to look in if a referenced assembly is not found in either the current working directory (the directory from which you are invoking the compiler) or the common language runtime's system directory. If the directory name contains a space, enclose the name in quotation marks ("").

Remarks

The `-libpath` option specifies the location of assemblies referenced by the `-reference` option.

The compiler searches for assembly references that are not fully qualified in the following order:

1. Current working directory. This is the directory from which the compiler is invoked.
2. The common language runtime system directory.
3. Directories specified by `/libpath`.
4. Directories specified by the LIB environment variable.

The `-libpath` option is additive; specifying it more than once appends to any prior values.

Use `-reference` to specify an assembly reference.

TO SET /LIBPATH IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **References** tab.
3. Click the **Reference Paths...** button.
4. In the **Reference Paths** dialog box, enter the directory name in the **Folder:** box.
5. Click **Add Folder**.

Example

The following code compiles `T2.vb` to create an .exe file. The compiler looks in the working directory, in the root directory of the C: drive, and in the New Assemblies directory of the C: drive for assembly references.

```
vbc -libpath:c:\;"c:\New Assemblies" -reference:t2.dll t2.vb
```

See also

- [Assemblies in .NET](#)
- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)

-link (Visual Basic)

10/17/2019 • 3 minutes to read • [Edit Online](#)

Causes the compiler to make COM type information in the specified assemblies available to the project that you are currently compiling.

Syntax

```
-link:fileList
```

or

```
-l:fileList
```

Arguments

TERM	DEFINITION
fileList	Required. Comma-delimited list of assembly file names. If the file name contains a space, enclose the name in quotation marks.

Remarks

The `-link` option enables you to deploy an application that has embedded type information. The application can then use types in a runtime assembly that implement the embedded type information without requiring a reference to the runtime assembly. If various versions of the runtime assembly are published, the application that contains the embedded type information can work with the various versions without having to be recompiled. For an example, see [Walkthrough: Embedding Types from Managed Assemblies](#).

Using the `-link` option is especially useful when you are working with COM interop. You can embed COM types so that your application no longer requires a primary interop assembly (PIA) on the target computer. The `-link` option instructs the compiler to embed the COM type information from the referenced interop assembly into the resulting compiled code. The COM type is identified by the CLSID (GUID) value. As a result, your application can run on a target computer that has installed the same COM types with the same CLSID values. Applications that automate Microsoft Office are a good example. Because applications like Office usually keep the same CLSID value across different versions, your application can use the referenced COM types as long as .NET Framework 4 or later is installed on the target computer and your application uses methods, properties, or events that are included in the referenced COM types.

The `-link` option embeds only interfaces, structures, and delegates. Embedding COM classes is not supported.

NOTE

When you create an instance of an embedded COM type in your code, you must create the instance by using the appropriate interface. Attempting to create an instance of an embedded COM type by using the CoClass causes an error.

To set the `-link` option in Visual Studio, add an assembly reference and set the `Embed Interop Types` property to **true**. The default for the `Embed Interop Types` property is **false**.

If you link to a COM assembly (Assembly A) which itself references another COM assembly (Assembly B), you also have to link to Assembly B if either of the following is true:

- A type from Assembly A inherits from a type or implements an interface from Assembly B.
- A field, property, event, or method that has a return type or parameter type from Assembly B is invoked.

Use `-libpath` to specify the directory in which one or more of your assembly references is located.

Like the `-reference` compiler option, the `-link` compiler option uses the Vbc.rsp response file, which references frequently used .NET Framework assemblies. Use the `-noconfig` compiler option if you do not want the compiler to use the Vbc.rsp file.

The short form of `-link` is `-l`.

Generics and Embedded Types

The following sections describe the limitations on using generic types in applications that embed interop types.

Generic Interfaces

Generic interfaces that are embedded from an interop assembly cannot be used. This is shown in the following example.

```
' The following code causes an error if ISampleInterface is an embedded interop type.  
Dim sample As ISampleInterface(Of SampleType)
```

Types That Have Generic Parameters

Types that have a generic parameter whose type is embedded from an interop assembly cannot be used if that type is from an external assembly. This restriction does not apply to interfaces. For example, consider the `Range` interface that is defined in the `Microsoft.Office.Interop.Excel` assembly. If a library embeds interop types from the `Microsoft.Office.Interop.Excel` assembly and exposes a method that returns a generic type that has a parameter whose type is the `Range` interface, that method must return a generic interface, as shown in the following code example.

```
Imports System.Collections.Generic  
Imports Microsoft.Office.Interop.Excel  
  
Class Utility  
    ' The following code causes an error when called by a client assembly.  
    Public Function GetRange1() As List(Of Range)
```

```
End Function  
  
' The following code is valid for calls from a client assembly.  
Public Function GetRange2() As IList(Of Range)
```

```
End Function  
End Class
```

In the following example, client code can call the method that returns the `IList` generic interface without error.

```
Module Client
    Public Sub Main()
        Dim util As New Utility()

        ' The following code causes an error.
        Dim rangeList1 As List(Of Range) = util.GetRange1()

        ' The following code is valid.
        Dim rangeList2 As List(Of Range) = CType(util.GetRange2(), List(Of Range))
    End Sub
End Module
```

Example

The following command line compiles source file `OfficeApp.vb` and reference assemblies from `COMData1.dll` and `COMData2.dll` to produce `OfficeApp.exe`.

```
vbc -link:COMData1.dll,COMData2.dll /out:OfficeApp.exe OfficeApp.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [Walkthrough: Embedding Types from Managed Assemblies](#)
- [-reference \(Visual Basic\)](#)
- [-noconfig](#)
- [-libpath](#)
- [Sample Compilation Command Lines](#)
- [Introduction to COM Interop](#)

-linkresource (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Creates a link to a managed resource.

Syntax

```
-linkresource:filename[,identifier[,public|private]]
```

or

```
-linkres:filename[,identifier[,public|private]]
```

Arguments

`filename`

Required. The resource file to link to the assembly. If the file name contains a space, enclose the name in quotation marks (" ").

`identifier`

Optional. The logical name for the resource. The name that is used to load the resource. The default is the name of the file. Optionally, you can specify whether the file is public or private in the assembly manifest, for example:

```
-linkres:filename.res,myname.res,public
```

Remarks

The `-linkresource` option does not embed the resource file in the output file; use the `-resource` option to do this.

The `-linkresource` option requires one of the `-target` options other than `-target:module`.

If `filename` is a .NET Framework resource file created, for example, by the [Resgen.exe \(Resource File Generator\)](#) or in the development environment, it can be accessed with members in the [System.Resources](#) namespace. (For more information, see [ResourceManager](#).) To access all other resources at run time, use the methods that begin with `GetManifestResource` in the [Assembly](#) class.

The file name can be any file format. For example, you may want to make a native DLL part of the assembly, so that it can be installed into the global assembly cache and accessed from managed code in the assembly.

The short form of `-linkresource` is `-linkres`.

NOTE

The `-linkresource` option is not available from the Visual Studio development environment; it is available only when you compile from the command line.

Example

The following code compiles `in.vb` and links to resource file `rf.resource`.

```
vbc -linkresource:rf.resource in.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [-target \(Visual Basic\)](#)
- [-resource \(Visual Basic\)](#)
- [Sample Compilation Command Lines](#)

-main

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies the class or module that contains the `Sub Main` procedure.

Syntax

```
-main:location
```

Arguments

`location`

Required. The name of the class or module that contains the `Sub Main` procedure to be called when the program starts. This may be in the form **-main:module** or **-main:namespace.module**.

Remarks

Use this option when you create an executable file or Windows executable program. If the **-main** option is omitted, the compiler searches for a valid shared `Sub Main` in all public classes and modules.

See [Main Procedure in Visual Basic](#) for a discussion of the various forms of the `Main` procedure.

When `location` is a class that inherits from `Form`, the compiler provides a default `Main` procedure that starts the application if the class has no `Main` procedure. This lets you compile code at the command line that was created in the development environment.

```
' Compile with /r:System.dll,SYSTEM.WINDOWS.FORMS.DLL /main:MyC
Public Class MyC
    Inherits System.Windows.Forms.Form
End Class
```

To set -main in the Visual Studio integrated development environment

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Application** tab.
3. Make sure the **Enable application framework** check box is not checked.
4. Modify the value in the **Startup object** box.

Example

The following code compiles `T2.vb` and `T3.vb`, specifying that the `Sub Main` procedure will be found in the `Test2` class.

```
vbc t2.vb t3.vb -main:Test2
```

See also

- [Visual Basic Command-Line Compiler](#)
- [-target \(Visual Basic\)](#)
- [Sample Compilation Command Lines](#)
- [Main Procedure in Visual Basic](#)

-moduleassemblyname

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies the name of the assembly that this module will be a part of.

Syntax

```
-moduleassemblyname:assembly_name
```

Arguments

TERM	DEFINITION
assembly_name	The name of the assembly that this module will be a part of.

Remarks

The compiler processes the `-moduleassemblyname` option only if the `-target:module` option has been specified. This causes the compiler to create a module. The module created by the compiler is valid only for the assembly specified with the `-moduleassemblyname` option. If you place the module in a different assembly, run-time errors will occur.

The `-moduleassemblyname` option is needed only when the following are true:

- A data type in the module needs access to a `Friend` type in a referenced assembly.
- The referenced assembly has granted friend assembly access to the assembly into which the module will be built.

For more information about creating a module, see [/target \(Visual Basic\)](#). For more information about friend assemblies, see [Friend Assemblies](#).

NOTE

The `-moduleassemblyname` option is not available from within the Visual Studio development environment; it is available only when you compile from a command prompt.

See also

- [How to: Build a Multifile Assembly](#)
- [Visual Basic Command-Line Compiler](#)
- [-target \(Visual Basic\)](#)
- [-main](#)
- [-reference \(Visual Basic\)](#)
- [-addmodule](#)
- [Assemblies in .NET](#)
- [Sample Compilation Command Lines](#)

- Friend Assemblies

-netcf

10/7/2019 • 2 minutes to read • [Edit Online](#)

Sets the compiler to target the .NET Compact Framework.

Syntax

```
-netcf
```

Remarks

The `-netcf` option causes the Visual Basic compiler to target the .NET Compact Framework rather than the full .NET Framework. Language functionality that is present only in the full .NET Framework is disabled.

The `-netcf` option is designed to be used with `-sdkpath`. The language features disabled by `-netcf` are the same language features not present in the files targeted with `-sdkpath`.

NOTE

The `-netcf` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. The `-netcf` option is set when a Visual Basic device project is loaded.

The `-netcf` option changes the following language features:

- The `End <keyword> Statement` keyword, which terminates execution of a program, is disabled. The following program compiles and runs without `-netcf` but fails at compile time with `-netcf`.

```
Module Module1
    Sub Main()
        End      ' not valid to terminate execution with /netcf
    End Sub
End Module
```

- Late binding, in all forms, is disabled. Compile-time errors are generated when recognized late-binding scenarios are encountered. The following program compiles and runs without `-netcf` but fails at compile time with `-netcf`.

```

Class LateBoundClass
    Sub S1()
        End Sub

        Default Property P1(ByVal s As String) As Integer
            Get
            End Get
            Set(ByVal Value As Integer)
            End Set
        End Property
    End Class

Module Module1
    Sub Main()
        Dim o1 As Object
        Dim o2 As Object
        Dim o3 As Object
        Dim IntArr(3) As Integer

        o1 = New LateBoundClass
        o2 = 1
        o3 = IntArr

        ' Late-bound calls
        o1.S1()
        o1.P1("member") = 1

        ' Dictionary member access
        o1!member = 1

        ' Late-bound overload resolution
        LateBoundSub(o2)

        ' Late-bound array
        o3(1) = 1
    End Sub

    Sub LateBoundSub(ByVal n As Integer)
    End Sub

    Sub LateBoundSub(ByVal s As String)
    End Sub
End Module

```

- The **Auto**, **Ansi**, and **Unicode** modifiers are disabled. The syntax of the **Declare Statement** statement is also modified to `Declare Sub|Function name Lib "library" [Alias "alias"] [(arglist)]`. The following code shows the effect of `-netcf` on a compilation.

```

' compile with: /target:library
Module Module1
    ' valid with or without /netcf
    Declare Sub DllSub Lib "SomeLib.dll" ()

    ' not valid with /netcf
    Declare Auto Sub DllSub1 Lib "SomeLib.dll" ()
    Declare Ansi Sub DllSub2 Lib "SomeLib.dll" ()
    Declare Unicode Sub DllSub3 Lib "SomeLib.dll" ()
End Module

```

- Using Visual Basic 6.0 keywords that were removed from Visual Basic generates a different error when `-netcf` is used. This affects the error messages for the following keywords:

○ [Open](#)

- `Close`
- `Put`
- `Print`
- `Write`
- `Input`
- `Lock`
- `Unlock`
- `Seek`
- `Width`
- `Name`
- `FreeFile`
- `EOF`
- `Loc`
- `LOF`
- `Line`

Example

The following code compiles `Myfile.vb` with the .NET Compact Framework, using the versions of mscorlib.dll and Microsoft.VisualBasic.dll found in the default installation directory of the .NET Compact Framework on the C drive. Typically, you would use the most recent version of the .NET Compact Framework.

```
vbc -netcf -sdkpath:"c:\Program Files\Microsoft Visual Studio .NET 2003\CompactFrameworkSDK\v1.0.5000\Windows  
CE " myfile.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)
- [-sdkpath](#)

-noconfig

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies that the compiler should not automatically reference the commonly used .NET Framework assemblies or import the `System` and `Microsoft.VisualBasic` namespaces.

Syntax

```
-noconfig
```

Remarks

The `-noconfig` option tells the compiler not to compile with the Vbc.rsp file, which is located in the same directory as the Vbc.exe file. The Vbc.rsp file references the commonly used .NET Framework assemblies and imports the `System` and `Microsoft.VisualBasic` namespaces. The compiler implicitly references the System.dll assembly unless the `-nostdlib` option is specified. The `-nostdlib` option tells the compiler not to compile with Vbc.rsp or automatically reference the System.dll assembly.

NOTE

The MsCorlib.dll and Microsoft.VisualBasic.dll assemblies are always referenced.

You can modify the Vbc.rsp file to specify additional compiler options that should be included in every Vbc.exe compilation (except when specifying the `-noconfig` option). For more information, see [@\(Specify Response File\)](#).

The compiler processes the options passed to the `vbc` command last. Therefore, any option on the command line overrides the setting of the same option in the Vbc.rsp file.

NOTE

The `-noconfig` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

See also

- [-nostdlib \(Visual Basic\)](#)
- [Visual Basic Command-Line Compiler](#)
- [@\(Specify Response File\)](#)
- [-reference \(Visual Basic\)](#)

-nologo (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Suppresses display of the copyright banner and informational messages during compilation.

Syntax

```
-nologo
```

Remarks

If you specify `-nologo`, the compiler does not display a copyright banner. By default, `-nologo` is not in effect.

NOTE

The `-nologo` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code compiles `t2.vb` and does not display a copyright banner.

```
vbc -nologo t2.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)

-nostdlib (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Causes the compiler not to automatically reference the standard libraries.

Syntax

```
-nostdlib
```

Remarks

The `-nostdlib` option removes the automatic reference to the System.dll assembly and prevents the compiler from reading the Vbc.rsp file. The Vbc.rsp file, which is located in the same directory as the Vbc.exe file, references the commonly used .NET Framework assemblies and imports the `System` and `Microsoft.VisualBasic` namespaces.

NOTE

The `Mscorlib.dll` and `Microsoft.VisualBasic.dll` assemblies are always referenced.

NOTE

The `-nostdlib` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code compiles `T2.vb` without referencing the standard libraries. You must set the `_MYTYPE` conditional-compilation constant to the string "Empty" to remove the `My` object.

```
vbc -nostdlib -define:_MYTYPE=\"Empty\" T2.vb
```

See also

- [-noconfig](#)
- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)
- [Customizing Which Objects are Available in My](#)

-nowarn

10/7/2019 • 2 minutes to read • [Edit Online](#)

Suppresses the compiler's ability to generate warnings.

Syntax

```
-nowarn[:numberList]
```

Arguments

TERM	DEFINITION
numberList	Optional. Comma-delimited list of the warning ID numbers that the compiler should suppress. If the warning IDs are not specified, all warnings are suppressed.

Remarks

The `-nowarn` option causes the compiler to not generate warnings. To suppress an individual warning, supply the warning ID to the `-nowarn` option following the colon. Separate multiple warning numbers with commas.

You need to specify only the numeric part of the warning identifier. For example, if you want to suppress BC42024, the warning for unused local variables, specify `-nowarn:42024`.

For more information on the warning ID numbers, see [Configuring Warnings in Visual Basic](#).

TO SET -NOWARN IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Select the **Disable all warnings** check box to disable all warnings.
- or -

To disable a particular warning, click **None** from the drop-down list adjacent to the warning.

Example

The following code compiles `T2.vb` and does not display any warnings.

```
vbc -nowarn t2.vb
```

Example

The following code compiles `T2.vb` and does not display the warnings for unused local variables (42024).

```
vbc -nowarn:42024 t2.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)
- [Configuring Warnings in Visual Basic](#)

-nowin32manifest (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Instructs the compiler not to embed any application manifest into the executable file.

Syntax

```
-nowin32manifest
```

Remarks

When this option is used, the application will be subject to virtualization on Windows Vista unless you provide an application manifest in a Win32 Resource file or during a later build step. For more information about virtualization, see [ClickOnce Deployment on Windows Vista](#).

For more information about manifest creation, see [-win32manifest \(Visual Basic\)](#).

See also

- [Visual Basic Command-Line Compiler](#)
- [Application Page, Project Designer \(Visual Basic\)](#)

-optimize

10/7/2019 • 2 minutes to read • [Edit Online](#)

Enables or disables compiler optimizations.

Syntax

```
-optimize[ + | - ]
```

Arguments

TERM	DEFINITION
+ -	Optional. The <code>-optimize-</code> option disables compiler optimizations. The <code>-optimize+</code> option enables optimizations. By default, optimizations are disabled.

Remarks

Compiler optimizations make your output file smaller, faster, and more efficient. However, because optimizations result in code rearrangement in the output file, `-optimize+` can make debugging difficult.

All modules generated with `-target:module` for an assembly must use the same `-optimize` settings as the assembly. For more information, see [-target \(Visual Basic\)](#).

You can combine the `-optimize` and `-debug` options.

TO SET -OPTIMIZE IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Click the **Advanced** button.
4. Modify the **Enable optimizations** check box.

Example

The following code compiles `t2.vb` and enables compiler optimizations.

```
vbc t2.vb -optimize
```

See also

- [Visual Basic Command-Line Compiler](#)
- [-debug \(Visual Basic\)](#)
- [Sample Compilation Command Lines](#)
- [-target \(Visual Basic\)](#)

-optioncompare

10/18/2019 • 2 minutes to read • [Edit Online](#)

Specifies how string comparisons are made.

Syntax

```
-optioncompare:{binary | text}
```

Remarks

You can specify `-optioncompare` in one of two forms: `-optioncompare:binary` to use binary string comparisons, and `-optioncompare:text` to use text string comparisons. By default, the compiler uses `-optioncompare:binary`.

In Microsoft Windows, the current code page determines the binary sort order. A typical binary sort order is as follows:

```
A < B < E < Z < a < b < e < z < À < È < Ø < à < è < ø
```

Text-based string comparisons are based on a case-insensitive text sort order determined by your system's locale. A typical text sort order is as follows:

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

To set -optioncompare in the Visual Studio IDE

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Modify the value in the **Option Compare** box.

To set -optioncompare programmatically

See [Option Compare Statement](#).

Example

The following code compiles `ProjFile.vb` and uses binary string comparisons.

```
vbc -optioncompare:binary projFile.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [-optionexplicit](#)
- [-optionstrict](#)
- [-optioninfer](#)
- [Sample Compilation Command Lines](#)
- [Option Compare Statement](#)
- [Visual Basic Defaults, Projects, Options Dialog Box](#)

-optionexplicit

10/7/2019 • 2 minutes to read • [Edit Online](#)

Causes the compiler to report errors if variables are not declared before they are used.

Syntax

```
-optionexplicit[+ | -]
```

Arguments

+ | -

Optional. Specify `-optionexplicit+` to require explicit declaration of variables. The `-optionexplicit+` option is the default and is the same as `-optionexplicit`. The `-optionexplicit-` option enables implicit declaration of variables.

Remarks

If the source code file contains an [Option Explicit statement](#), the statement overrides the `-optionexplicit` command-line compiler setting.

To set `-optionexplicit` in the Visual Studio IDE

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Modify the value in the **Option Explicit** box.

Example

The following code compiles when `-optionexplicit-` is used.

```
Module Module1
    Sub Main()
        i = 99
        System.Console.WriteLine(i)
    End Sub
End Module
```

See also

- [Visual Basic Command-Line Compiler](#)
- [-optioncompare](#)
- [-optionstrict](#)
- [-optioninfer](#)
- [Sample Compilation Command Lines](#)
- [Option Explicit Statement](#)
- [Visual Basic Defaults, Projects, Options Dialog Box](#)

-optioninfer

10/7/2019 • 2 minutes to read • [Edit Online](#)

Enables the use of local type inference in variable declarations.

Syntax

```
-optioninfer[+ | -]
```

Arguments

TERM	DEFINITION
+ -	Optional. Specify <code>-optioninfer+</code> to enable local type inference, or <code>-optioninfer-</code> to block it. The <code>-optioninfer</code> option, with no value specified, is the same as <code>-optioninfer+</code> . The default value when the <code>-optioninfer</code> switch is not present is also <code>-optioninfer+</code> . The default value is set in the Vbc.rsp response file.

NOTE

You can use the `-noconfig` option to retain the compiler's internal defaults instead of those specified in vbc.rsp. The compiler default for this option is `-optioninfer-`.

Remarks

If the source code file contains an [Option Infer Statement](#), the statement overrides the `-optioninfer` command-line compiler setting.

To set `-optioninfer` in the Visual Studio IDE

1. Select a project in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. On the **Compile** tab, modify the value in the **Option infer** box.

Example

The following code compiles `test.vb` with local type inference enabled.

```
vbc -optioninfer+ test.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [-optioncompare](#)
- [-optionexplicit](#)
- [-optionstrict](#)

- [Sample Compilation Command Lines](#)
- [Option Infer Statement](#)
- [Local Type Inference](#)
- [Visual Basic Defaults, Projects, Options Dialog Box](#)
- [Compile Page, Project Designer \(Visual Basic\)](#)
- [/noconfig](#)
- [Building from the Command Line](#)

-optionstrict

10/18/2019 • 2 minutes to read • [Edit Online](#)

Enforces strict type semantics to restrict implicit type conversions.

Syntax

```
-optionstrict[+ | -]  
-optionstrict[:custom]
```

Arguments

+ | -

Optional. The `-optionstrict+` option restricts implicit type conversion. The default for this option is `-optionstrict-`. The `-optionstrict+` option is the same as `-optionstrict`. You can use both for permissive type semantics.

custom

Required. Warn when strict language semantics are not respected.

Remarks

When `-optionstrict+` is in effect, only widening type conversions can be made implicitly. Implicit narrowing type conversions, such as assigning a `Decimal` type object to an integer type object, are reported as errors.

To generate warnings for implicit narrowing type conversions, use `-optionstrict:custom`. Use `-nowarn:numberlist` to ignore particular warnings and `-warnaserror:numberlist` to treat particular warnings as errors.

To set `-optionstrict` in the Visual Studio IDE

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Modify the value in the **Option Strict** box.

To set `-optionstrict` programmatically

See [Option Strict Statement](#).

Example

The following code compiles `Test.vb` using strict type semantics.

```
vbc -optionstrict+ test.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [-optioncompare](#)

- [-optionexplicit](#)
- [-optioninfer](#)
- [-nowarn](#)
- [-warnaserror \(Visual Basic\)](#)
- [Sample Compilation Command Lines](#)
- [Option Strict Statement](#)
- [Visual Basic Defaults, Projects, Options Dialog Box](#)

-out (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies the name of the output file.

Syntax

```
-out:filename
```

Arguments

TERM	DEFINITION
<code>filename</code>	Required. The name of the output file the compiler creates. If the file name contains a space, enclose the name in quotation marks ("").

Remarks

Specify the full name and extension of the file to create. If you do not, the .exe file takes its name from the source-code file containing the `Sub Main` procedure, and the .dll file takes its name from the first source-code file.

If you specify a file name without an .exe or .dll extension, the compiler automatically adds the extension for you, depending on the value specified for the `-target` compiler option.

TO SET -OUT IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Application** tab.
3. Modify the value in the **Assembly Name** box.

Example

The following code compiles `T2.vb` and creates output file `T2.exe`.

```
vbc t2.vb -out:t3.exe
```

See also

- [Visual Basic Command-Line Compiler](#)
- [-target \(Visual Basic\)](#)
- [Sample Compilation Command Lines](#)

-platform (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies which platform version of common language runtime (CLR) can run the output file.

Syntax

```
-platform:{ x86 | x64 | Itanium | arm | anycpu | anycpu32bitpreferred }
```

Arguments

TERM	DEFINITION
x86	Compiles your assembly to be run by the 32-bit, x86-compatible CLR.
x64	Compiles your assembly to be run by the 64-bit CLR on a computer that supports the AMD64 or EM64T instruction set.
Itanium	Compiles your assembly to be run by the 64-bit CLR on a computer with an Itanium processor.
arm	Compiles your assembly to be run on a computer with an ARM (Advanced RISC Machine) processor.
anycpu	Compiles your assembly to run on any platform. The application will run as a 32-bit application on 32-bit versions of Windows and as a 64-bit application on 64-bit versions of Windows. This flag is the default value.
anycpu32bitpreferred	Compiles your assembly to run on any platform. The application will run as a 32-bit application on both 32-bit and 64-bit versions of Windows. This flag is valid only for executables (.EXE) and requires .NET Framework 4.5.

Remarks

Use the `-platform` option to specify the type of processor targeted by the output file.

In general, .NET Framework assemblies written in Visual Basic will run the same regardless of the platform. However, there are some cases that behave differently on different platforms. These common cases are:

- Structures that contain members that change size depending on the platform, such as any pointer type.
- Pointer arithmetic that includes constant sizes.
- Incorrect platform invoke or COM declarations that use `Integer` for handles instead of `IntPtr`.
- Casting `IntPtr` to `Integer`.
- Using platform invoke or COM interop with components that do not exist on all platforms.

The **-platform** option will mitigate some issues if you know you have made assumptions about the architecture your code will run on. Specifically:

- If you decide to target a 64-bit platform, and the application is run on a 32-bit machine, the error message comes much earlier and is more targeted at the problem than the error that occurs without using this switch.
- If you set the `x86` flag on the option and the application is subsequently run on a 64-bit machine, the application will run in the WOW subsystem instead of running natively.

On a 64-bit Windows operating system:

- Assemblies compiled with `-platform:x86` will execute on the 32-bit CLR running under WOW64.
- Executables compiled with the `-platform:anycpu` will execute on the 64-bit CLR.
- A DLL compiled with the `-platform:anycpu` will execute on the same CLR as the process into which it loaded.
- Executables that are compiled with `-platform:anycpu32bitpreferred` will execute on the 32-bit CLR.

For more information about how to develop an application to run on a 64-bit version of Windows, see [64-bit Applications](#).

To set **-platform** in the Visual Studio IDE

1. In **Solution Explorer**, choose the project, open the **Project** menu, and then click **Properties**.
2. On the **Compile** tab, select or clear the **Prefer 32-bit** check box, or, in the **Target CPU** list, choose a value.

For more information, see [Compile Page, Project Designer \(Visual Basic\)](#).

Example

The following example illustrates how to use the `-platform` compiler option.

```
vbc -platform:x86 myFile.vb
```

See also

- [/target \(Visual Basic\)](#)
- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)

-quiet

10/7/2019 • 2 minutes to read • [Edit Online](#)

Prevents the compiler from displaying code for syntax-related errors and warnings.

Syntax

```
-quiet
```

Remarks

By default, `-quiet` is not in effect. When the compiler reports a syntax-related error or warning, it also outputs the line from source code. For applications that parse compiler output, it may be more convenient for the compiler to output only the text of the diagnostic.

In the following example, `Module1` outputs an error that includes source code when compiled without `-quiet`.

```
Module Module1
    Sub Main()
        x()
    End Sub
End Module
```

Output:

```
C:\projects\vb2.vb(3) : error BC30451: 'x' is not declared. It may be inaccessible due to its protection
level.

x()
~
```

Compiled with `-quiet`, the compiler outputs only the following:

```
E:\test\t2.vb(3) : error BC30451: Name 'x' is not declared.
```

NOTE

The `-quiet` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code compiles `T2.vb` and does not display code for syntax-related compiler diagnostics:

```
vbc -quiet t2.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)

-recurse

10/7/2019 • 2 minutes to read • [Edit Online](#)

Compiles source-code files in all child directories of either the specified directory or the project directory.

Syntax

```
-recurse:[dir\]file
```

Arguments

`dir`

Optional. The directory in which you want the search to begin. If not specified, the search begins in the project directory.

`file`

Required. The file(s) to search for. Wildcard characters are allowed.

Remarks

You can use wildcards in a file name to compile all matching files in the project directory without using `-recurse`. If no output file name is specified, the compiler bases the output file name on the first input file processed. This is generally the first file in the list of files compiled when viewed alphabetically. For this reason, it is best to specify an output file using the `-out` option.

NOTE

The `-recurse` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following command compiles all Visual Basic files in the current directory.

```
vbc *.vb
```

The following command compiles all Visual Basic files in the `Test\ABC` directory and any directories below it, and then generates `Test.ABC.dll`.

```
vbc -target:library -out:Test.ABC.dll -recurse:Test\ABC\*.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [-out \(Visual Basic\)](#)
- [Sample Compilation Command Lines](#)

-reference (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Causes the compiler to make type information in the specified assemblies available to the project you are currently compiling.

Syntax

```
-reference:fileList
```

or

```
-r:fileList
```

Arguments

TERM	DEFINITION
<code>fileList</code>	Required. Comma-delimited list of assembly file names. If the file name contains a space, enclose the name in quotation marks.

Remarks

The file(s) you import must contain assembly metadata. Only public types are visible outside the assembly. The [/addmodule](#) option imports metadata from a module.

If you reference an assembly (Assembly A) which itself references another assembly (Assembly B), you need to reference Assembly B if:

- A type from Assembly A inherits from a type or implements an interface from Assembly B.
- A field, property, event, or method that has a return type or parameter type from Assembly B is invoked.

Use [-libpath](#) to specify the directory in which one or more of your assembly references is located.

For the compiler to recognize a type in an assembly (not a module), it must be forced to resolve the type. One example of how you can do this is to define an instance of the type. Other ways are available to resolve type names in an assembly for the compiler. For example, if you inherit from a type in an assembly, the type name then becomes known to the compiler.

The Vbc.rsp response file, which references commonly used .NET Framework assemblies, is used by default. Use `-noconfig` if you do not want the compiler to use Vbc.rsp.

The short form of `-reference` is `/r`.

Example

The following command compiles source file `Input.vb` and reference assemblies from `Metad1.dll` and `Metad2.dll` to produce `Out.exe`.

```
vbc -reference:metad1.dll,metad2.dll -out:out.exe input.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [-noconfig](#)
- [-target \(Visual Basic\)](#)
- [Public](#)
- [Sample Compilation Command Lines](#)

-refonly (Visual Basic)

4/2/2019 • 2 minutes to read • [Edit Online](#)

The **-refonly** option indicates that the primary output of the compilation should be a reference assembly instead of an implementation assembly. The `-refonly` parameter silently disables outputting PDBs, as reference assemblies cannot be executed.

Every compiler option is available in two forms: **-option** and **/option**. The documentation only shows the -option form.

Syntax

```
-refonly
```

Remarks

Visual Basic supports the `-refout` switch starting with version 15.3.

Reference assemblies are metadata-only assemblies that contain metadata but no implementation code. They include type and member information for everything except anonymous types. The reason for using `throw null` bodies (as opposed to no bodies) is so that PEVerify could run and pass (thus validating the completeness of the metadata).

Reference assemblies include an assembly-level [ReferenceAssembly](#) attribute. This attribute may be specified in source (then the compiler won't need to synthesize it). Because of this attribute, runtimes will refuse to load reference assemblies for execution (but they can still be loaded in a reflection-only context). Tools that reflect on assemblies need to ensure they load reference assemblies as reflection-only; otherwise, the runtime throws a [BadImageFormatException](#).

The `-refonly` and `-refout` options are mutually exclusive.

See also

- [-refout](#)
- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)

-refout (Visual Basic)

10/18/2019 • 2 minutes to read • [Edit Online](#)

The **-refout** option specifies a file path where the reference assembly should be output.

Every compiler option is available in two forms: **-option** and **/option**. The documentation only shows the -option form.

Syntax

```
-refout:filepath
```

Arguments

`filepath`

The path and filename of the reference assembly. It should generally be in a sub-folder of the primary assembly. The recommended convention (used by MSBuild) is to place the reference assembly in a "ref/" sub-folder relative to the primary assembly. All folders in `filepath` must exist; the compiler does not create them.

Remarks

Visual Basic supports the `-refout` switch starting with version 15.3.

Reference assemblies are metadata-only assemblies that contain metadata but no implementation code. They include type and member information for everything except anonymous types. Their method bodies are replaced with a single `throw null` statement. The reason for using `throw null` method bodies (as opposed to no bodies) is so that PEVerify can run and pass (thus validating the completeness of the metadata).

Reference assemblies include an assembly-level [ReferenceAssembly](#) attribute. This attribute may be specified in source (then the compiler won't need to synthesize it). Because of this attribute, runtimes refuse to load reference assemblies for execution (but they can still be loaded in a reflection-only context). Tools that reflect on assemblies need to ensure they load reference assemblies as reflection-only; otherwise, the runtime throws a [BadImageFormatException](#).

The `-refout` and `-refonly` options are mutually exclusive.

See also

- [-refonly](#)
- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)

-removeintchecks

10/7/2019 • 2 minutes to read • [Edit Online](#)

Turns overflow-error checking for integer operations on or off.

Syntax

```
-removeintchecks[+ | -]
```

Arguments

TERM	DEFINITION
+ -	<p>Optional. The <code>-removeintchecks-</code> option causes the compiler to check all integer calculations for overflow errors. The default is <code>-removeintchecks-</code>.</p> <p>Specifying <code>-removeintchecks</code> or <code>-removeintchecks+</code> prevents error checking and can make integer calculations faster. However, without error checking, and if data type capacities are overflowed, incorrect results may be stored without raising an error.</p>

TO SET -REMOVEINTCHECKS IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Click the **Advanced** button.
4. Modify the value of the **Remove integer overflow checks** box.

Example

The following code compiles `Test.vb` and turns off integer overflow-error checking.

```
vbc -removeintchecks+ test.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)

-resource (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Embeds a managed resource in an assembly.

Syntax

```
-resource:filename[,identifier[,public|private]]
```

or

```
-res:filename[,identifier[,public|private]]
```

Arguments

TERM	DEFINITION
<code>filename</code>	Required. The name of the resource file to embed in the output file. By default, <code>filename</code> is public in the assembly. Enclose the file name in quotation marks (" ") if it contains a space.
<code>identifier</code>	Optional. The logical name for the resource; the name used to load it. The default is the name of the file. Optionally, you can specify whether the resource is public or private in the assembly manifest, as with the following: <code>-res:filename.res, myname.res, public</code>

Remarks

Use `-linkresource` to link a resource to an assembly without placing the resource file in the output file.

If `filename` is a .NET Framework resource file created, for example, by the [Resgen.exe \(Resource File Generator\)](#) or in the development environment, it can be accessed with members in the [System.Resources](#) namespace (see [ResourceManager](#) for more information). To access all other resources at run time, use one of the following methods: [GetManifestResourceInfo](#), [GetManifestResourceNames](#), or [GetManifestResourceStream](#).

The short form of `-resource` is `-res`.

For information about how to set `-resource` in the Visual Studio IDE, see [Managing Application Resources \(.NET\)](#).

Example

The following code compiles `In.vb` and attaches resource file `Rf.resource`.

```
vbc -res:rf.resource in.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [-win32resource](#)
- [-linkresource \(Visual Basic\)](#)
- [-target \(Visual Basic\)](#)
- [Sample Compilation Command Lines](#)

-rootnamespace

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies a namespace for all type declarations.

Syntax

```
-rootnamespace:namespace
```

Arguments

TERM	DEFINITION
<code>namespace</code>	The name of the namespace in which to enclose all type declarations for the current project.

Remarks

If you use the Visual Studio executable file (Devenv.exe) to compile a project created in the Visual Studio integrated development environment, use `-rootnamespace` to specify the value of the [RootNamespace](#) property. See [Devenv Command Line Switches](#) for more information.

Use the common language runtime MSIL Disassembler (`Ildasm.exe`) to view the namespace names in your output file.

TO SET -ROOTNAMESPACE IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Application** tab.
3. Modify the value in the **Root Namespace** box.

Example

The following code compiles `In.vb` and encloses all type declarations in the namespace `mynamespace`.

```
vbc -rootnamespace:mynamespace in.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [Ildasm.exe \(IL Disassembler\)](#)
- [Sample Compilation Command Lines](#)

-sdkpath

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies the location of msclib.dll and Microsoft.VisualBasic.dll.

Syntax

```
-sdkpath: path
```

Arguments

path

The directory containing the versions of msclib.dll and Microsoft.VisualBasic.dll to use for compilation. This path is not verified until it is loaded. Enclose the directory name in quotation marks (" ") if it contains a space.

Remarks

This option tells the Visual Basic compiler to load the msclib.dll and Microsoft.VisualBasic.dll files from a non-default location. The `-sdkpath` option was designed to be used with `-netcf`. The .NET Compact Framework uses different versions of these support libraries to avoid the use of types and language features not found on the devices.

NOTE

The `-sdkpath` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. The `-sdkpath` option is set when a Visual Basic device project is loaded.

You can specify that the compiler should compile without a reference to the Visual Basic Runtime Library by using the `-vbruntime` compiler option. For more information, see [-vbruntime](#).

Example

The following code compiles `Myfile.vb` with the .NET Compact Framework, using the versions of Msclib.dll and Microsoft.VisualBasic.dll found in the default installation directory of the .NET Compact Framework on the C drive. Typically, you would use the most recent version of the .NET Compact Framework.

```
vbc -netcf -sdkpath:"c:\Program Files\Microsoft Visual Studio .NET 2003\CompactFrameworkSDK\v1.0.5000\Windows CE" myfile.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)
- [-netcf](#)
- [-vbruntime](#)

-target (Visual Basic)

10/18/2019 • 4 minutes to read • [Edit Online](#)

Specifies the format of compiler output.

Syntax

```
-target:{exe | library | module | winexe | appcontainerexe | winmdobj}
```

Remarks

The following table summarizes the effect of the `-target` option.

OPTION	BEHAVIOR
<code>-target:exe</code>	<p>Causes the compiler to create an executable console application.</p> <p>This is the default option when no <code>-target</code> option is specified. The executable file is created with an .exe extension.</p> <p>Unless otherwise specified with the <code>/out</code> option, the output file name takes the name of the input file that contains the <code>Sub Main</code> procedure.</p> <p>Only one <code>Sub Main</code> procedure is required in the source-code files that are compiled into an .exe file. Use the <code>-main</code> compiler option to specify which class contains the <code>Sub Main</code> procedure.</p>
<code>-target:library</code>	<p>Causes the compiler to create a dynamic-link library (DLL).</p> <p>The dynamic-link library file is created with a .dll extension.</p> <p>Unless otherwise specified with the <code>-out</code> option, the output file name takes the name of the first input file.</p> <p>When building a DLL, a <code>Sub Main</code> procedure is not required.</p>

OPTION	BEHAVIOR
<code>-target:module</code>	<p>Causes the compiler to generate a module that can be added to an assembly.</p> <p>The output file is created with an extension of .netmodule.</p> <p>The .NET common language runtime cannot load a file that does not have an assembly. However, you can incorporate such a file into the assembly manifest of an assembly by using <code>-reference</code>.</p> <p>When code in one module references internal types in another module, both modules must be incorporated into an assembly manifest by using <code>-reference</code>.</p> <p>The <code>-addmodule</code> option imports metadata from a module.</p>
<code>-target:winexe</code>	<p>Causes the compiler to create an executable Windows-based application.</p> <p>The executable file is created with an .exe extension. A Windows-based application is one that provides a user interface from either the .NET Framework class library or with the Windows APIs.</p> <p>Unless otherwise specified with the <code>-out</code> option, the output file name takes the name of the input file that contains the <code>Sub Main</code> procedure.</p> <p>Only one <code>Sub Main</code> procedure is required in the source-code files that are compiled into an .exe file. In cases where your code has more than one class that has a <code>Sub Main</code> procedure, use the <code>-main</code> compiler option to specify which class contains the <code>Sub Main</code> procedure</p>
<code>-target:appcontainerexe</code>	<p>Causes the compiler to create an executable Windows-based application that must be run in an app container. This setting is designed to be used for Windows 8.x Store applications.</p> <p>The appcontainerexe setting sets a bit in the Characteristics field of the Portable Executable file. This bit indicates that the app must be run in an app container. When this bit is set, an error occurs if the <code>CreateProcess</code> method tries to launch the application outside of an app container. Aside from this bit setting, -target:appcontainerexe is equivalent to -target:winexe.</p> <p>The executable file is created with an .exe extension.</p> <p>Unless you specify otherwise by using the <code>-out</code> option, the output file name takes the name of the input file that contains the <code>Sub Main</code> procedure.</p> <p>Only one <code>Sub Main</code> procedure is required in the source-code files that are compiled into an .exe file. If your code contains more than one class that has a <code>Sub Main</code> procedure, use the <code>-main</code> compiler option to specify which class contains the <code>Sub Main</code> procedure</p>

OPTION	BEHAVIOR
<code>-target:winmdobj</code>	<p>Causes the compiler to create an intermediate file that you can convert to a Windows Runtime binary (.winmd) file. The .winmd file can be consumed by JavaScript and C++ programs, in addition to managed language programs.</p> <p>The intermediate file is created with a .winmdobj extension.</p> <p>Unless you specify otherwise by using the <code>-out</code> option, the output file name takes the name of the first input file. A <code>Sub Main</code> procedure isn't required.</p> <p>The .winmdobj file is designed to be used as input for the WinMDExp export tool to produce a Windows metadata (WinMD) file. The WinMD file has a .winmd extension and contains both the code from the original library and the WinMD definitions that JavaScript, C++, and the Windows Runtime use.</p>

Unless you specify `-target:module`, `-target` causes a .NET Framework assembly manifest to be added to an output file.

Each instance of Vbc.exe produces, at most, one output file. If you specify a compiler option such as `-out` or `-target` more than one time, the last one the compiler processes is put into effect. Information about all files in a compilation is added to the manifest. All output files except those created with `-target:module` contain assembly metadata in the manifest. Use [Ildasm.exe \(IL Disassembler\)](#) to view the metadata in an output file.

The short form of `-target` is `-t`.

To set `-target` in the Visual Studio IDE

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Application** tab.
3. Modify the value in the **Application Type** box.

Example

The following code compiles `in.vb`, creating `in.dll`:

```
vbc -target:library in.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [-main](#)
- [-out \(Visual Basic\)](#)
- [-reference \(Visual Basic\)](#)
- [-addmodule](#)
- [-moduleassemblyname](#)
- [Assemblies in .NET](#)
- [Sample Compilation Command Lines](#)

-subsystemversion (Visual Basic)

5/30/2019 • 2 minutes to read • [Edit Online](#)

Specifies the minimum version of the subsystem on which the generated executable file can run, thereby determining the versions of Windows on which the executable file can run. Most commonly, this option ensures that the executable file can leverage particular security features that aren't available with older versions of Windows.

NOTE

To specify the subsystem itself, use the [-target](#) compiler option.

Syntax

```
-subsystemversion:major.minor
```

Parameters

`major.minor`

The minimum required version of the subsystem, as expressed in a dot notation for major and minor versions. For example, you can specify that an application can't run on an operating system that's older than Windows 7 if you set the value of this option to 6.01, as the table later in this topic describes. You must specify the values for `major` and `minor` as integers.

Leading zeroes in the `minor` version don't change the version, but trailing zeroes do. For example, 6.1 and 6.01 refer to the same version, but 6.10 refers to a different version. We recommend expressing the minor version as two digits to avoid confusion.

Remarks

The following table lists common subsystem versions of Windows.

WINDOWS VERSION	SUBSYSTEM VERSION
Windows 2000	5.00
Windows XP	5.01
Windows Server 2003	5.02
Windows Vista	6.00
Windows 7	6.01
Windows Server 2008	6.01
Windows 8	6.02

Default values

The default value of the **-subsystemversion** compiler option depends on the conditions in the following list:

- The default value is 6.02 if any compiler option in the following list is set:
 - [-target:appcontainerexe](#)
 - [-targetwinmdobj](#)
 - [-platform:arm](#)
- The default value is 6.00 if you're using MSBuild, you're targeting .NET Framework 4.5, and you haven't set any of the compiler options that were specified earlier in this list.
- The default value is 4.00 if none of the previous conditions is true.

Setting this option

To set the **-subsystemversion** compiler option in Visual Studio, you must open the .vbproj file and specify a value for the `SubsystemVersion` property in the MSBuild XML. You can't set this option in the Visual Studio IDE. For more information, see "Default values" earlier in this topic or [Common MSBuild Project Properties](#).

See also

- [Visual Basic Command-Line Compiler](#)
- [MSBuild Properties](#)

-utf8output (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Displays compiler output using UTF-8 encoding.

Syntax

```
-utf8output[+ | -]
```

Arguments

+ | -

Optional. The default for this option is `-utf8output-`, which means compiler output does not use UTF-8 encoding. Specifying `-utf8output` is the same as specifying `-utf8output+`.

Remarks

In some international configurations, compiler output cannot be displayed correctly in the console. In such situations, use `-utf8output` and redirect compiler output to a file.

NOTE

The `-utf8output` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code compiles `In.vb` and directs the compiler to display output using UTF-8 encoding.

```
vbc -utf8output in.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)

-vbruntime

10/7/2019 • 2 minutes to read • [Edit Online](#)

Specifies that the compiler should compile without a reference to the Visual Basic Runtime Library, or with a reference to a specific runtime library.

Syntax

```
-vbruntime:{ - | + | * | path }
```

Arguments

- Compile without a reference to the Visual Basic Runtime Library.

+ Compile with a reference to the default Visual Basic Runtime Library.

* Compile without a reference to the Visual Basic Runtime Library, and embed core functionality from the Visual Basic Runtime Library into the assembly.

path

Compile with a reference to the specified library (DLL).

Remarks

The `-vbruntime` compiler option enables you to specify that the compiler should compile without a reference to the Visual Basic Runtime Library. If you compile without a reference to the Visual Basic Runtime Library, errors or warnings are logged on code or language constructs that generate a call to a Visual Basic runtime helper. (A *Visual Basic runtime helper* is a function defined in Microsoft.VisualBasic.dll that is called at runtime to execute a specific language semantic.)

The `-vbruntime+` option produces the same behavior that occurs if no `-vbruntime` switch is specified. You can use the `-vbruntime+` option to override previous `-vbruntime` switches.

Most objects of the `My` type are unavailable when you use the `-vbruntime-` or `-vbruntime:path` options.

Embedding Visual Basic Runtime core functionality

The `-vbruntime*` option enables you to compile without a reference to a runtime library. Instead, core functionality from the Visual Basic Runtime Library is embedded in the user assembly. You can use this option if your application runs on platforms that do not contain the Visual Basic runtime.

The following runtime members are embedded:

- [Conversions](#) class
- [AscW\(Char\)](#) method
- [AscW\(String\)](#) method

- [ChrW\(Int32\) method](#)
- [vbBack constant](#)
- [vbCr constant](#)
- [vbCrLf constant](#)
- [vbFormFeed constant](#)
- [vbLf constant](#)
- [vbNewLine constant](#)
- [vbNullChar constant](#)
- [vbNullString constant](#)
- [vbTab constant](#)
- [vbVerticalTab constant](#)
- Some objects of the `My` type

If you compile using the `-vbruntime*` option and your code references a member from the Visual Basic Runtime Library that is not embedded with the core functionality, the compiler returns an error that indicates that the member is not available.

Referencing a specified library

You can use the `path` argument to compile with a reference to a custom runtime library instead of the default Visual Basic Runtime Library.

If the value for the `path` argument is a fully qualified path to a DLL, the compiler will use that file as the runtime library. If the value for the `path` argument is not a fully qualified path to a DLL, the Visual Basic compiler will search for the identified DLL in the current folder first. It will then search in the path that you have specified by using the `-sdkpath` compiler option. If the `-sdkpath` compiler option is not used, the compiler will search for the identified DLL in the .NET Framework folder (`%systemroot%\Microsoft.NET\Framework\versionNumber`).

Example

The following example shows how to use the `-vbruntime` option to compile with a reference to a custom library.

```
vbc -vbruntime:C:\VBLibraries\CustomVBLibrary.dll
```

See also

- [Visual Basic Core – New compilation mode in Visual Studio 2010 SP1](#)
- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)
- [-sdkpath](#)

-verbose

10/7/2019 • 2 minutes to read • [Edit Online](#)

Causes the compiler to produce verbose status and error messages.

Syntax

```
-verbose[+ | -]
```

Arguments

+ | -

Optional. Specifying `-verbose` is the same as specifying `-verbose+`, which causes the compiler to emit verbose messages. The default for this option is `-verbose-`.

Remarks

The `-verbose` option displays information about the total number of errors issued by the compiler, reports which assemblies are being loaded by a module, and displays which files are currently being compiled.

NOTE

The `-verbose` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code compiles `In.vb` and directs the compiler to display verbose status information.

```
vbc -verbose in.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)

-warnaserror (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Causes the compiler to treat the first occurrence of a warning as an error.

Syntax

```
-warnaserror[+ | -][:numberList]
```

Arguments

TERM	DEFINITION
+ -	Optional. By default, <code>-warnaserror-</code> is in effect; warnings do not prevent the compiler from producing an output file. The <code>-warnaserror</code> option, which is the same as <code>-warnaserror+</code> , causes warnings to be treated as errors.
numberList	Optional. Comma-delimited list of the warning ID numbers to which the <code>-warnaserror</code> option applies. If no warning ID is specified, the <code>-warnaserror</code> option applies to all warnings.

Remarks

The `-warnaserror` option treats all warnings as errors. Any messages that would ordinarily be reported as warnings are instead reported as errors. The compiler reports subsequent occurrences of the same warning as warnings.

By default, `-warnaserror-` is in effect, which causes the warnings to be informational only. The `-warnaserror` option, which is the same as `-warnaserror+`, causes warnings to be treated as errors.

If you want only a few specific warnings to be treated as errors, you may specify a comma-separated list of warning numbers to treat as errors.

NOTE

The `-warnaserror` option does not control how warnings are displayed. Use the `-nowarn` option to disable warnings.

TO SET -WARNASERROR TO TREAT ALL WARNINGS AS ERRORS IN THE VISUAL STUDIO IDE

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Make sure the **Disable all warnings** check box is unchecked.
4. Check the **Treat all warnings as errors** check box.

TO SET -WARNASERROR TO TREAT SPECIFIC WARNINGS AS ERRORS IN THE VISUAL STUDIO IDE

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Make sure the **Disable all warnings** check box is unchecked.
4. Make sure the **Treat all warnings as errors** check box is unchecked.
5. Select **Error** from the **Notification** column adjacent to the warning that should be treated as an error.

Example

The following code compiles `In.vb` and directs the compiler to display an error for the first occurrence of every warning it finds.

```
vbc -warnaserror in.vb
```

Example

The following code compiles `T2.vb` and treats only the warning for unused local variables (42024) as an error.

```
vbc -warnaserror:42024 t2.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)
- [Configuring Warnings in Visual Basic](#)

-win32icon

10/7/2019 • 2 minutes to read • [Edit Online](#)

Inserts an .ico file in the output file. This .ico file represents the output file in **File Explorer**.

Syntax

```
-win32icon:filename
```

Arguments

TERM	DEFINITION
filename	The .ico file to add to your output file. Enclose the file name in quotation marks (" ") if it contains a space.

Remarks

You can create an .ico file with the Microsoft Windows Resource Compiler (RC). The resource compiler is invoked when you compile a Visual C++ program; an .ico file is created from the .rc file. The `-win32icon` and `-win32resource` options are mutually exclusive.

See [-linkresource \(Visual Basic\)](#) to reference a .NET Framework resource file, or [-resource \(Visual Basic\)](#) to attach a .NET Framework resource file. See [-win32resource](#) to import a .res file.

TO SET -WIN32ICON IN THE VISUAL STUDIO IDE

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Application** tab.
3. Modify the value in the **Icon** box.

Example

The following code compiles `In.vb` and attaches an .ico file, `Rf.ico`.

```
vbc -win32icon:rf.ico in.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)

-win32manifest (Visual Basic)

10/7/2019 • 2 minutes to read • [Edit Online](#)

Identifies a user-defined Win32 application manifest file to be embedded into a project's portable executable (PE) file.

Syntax

```
-win32manifest: fileName
```

Arguments

TERM	DEFINITION
fileName	The path of the custom manifest file.

Remarks

By default, the Visual Basic compiler embeds an application manifest that specifies a requested execution level of asInvoker. It creates the manifest in the same folder in which the executable file is built, typically the bin\Debug or bin\Release folder when you use Visual Studio. If you want to supply a custom manifest, for example to specify a requested execution level of highestAvailable or requireAdministrator, use this option to specify the name of the file.

NOTE

This option and the [-win32resource](#) option are mutually exclusive. If you try to use both options in the same command line, you will get a build error.

An application that has no application manifest that specifies a requested execution level will be subject to file/registry virtualization under the User Account Control feature in Windows Vista. For more information about virtualization, see [ClickOnce Deployment on Windows Vista](#).

Your application will be subject to virtualization if either of the following conditions is true:

1. You use the `-nowin32manifest` option and you do not provide a manifest in a later build step or as part of a Windows Resource (.res) file by using the `-win32resource` option.
2. You provide a custom manifest that does not specify a requested execution level.

Visual Studio creates a default .manifest file and stores it in the debug and release directories alongside the executable file. You can view or edit the default app.manifest file by clicking **View UAC Settings** on the **Application** tab in the Project Designer. For more information, see [Application Page, Project Designer \(Visual Basic\)](#).

You can provide the application manifest as a custom post-build step or as part of a Win32 resource file by using the `-nowin32manifest` option. Use that same option if you want your application to be subject to file or registry virtualization on Windows Vista. This will prevent the compiler from creating and embedding a default manifest in the PE file.

Example

The following example shows the default manifest that the Visual Basic compiler inserts into a PE.

NOTE

The compiler inserts a standard application name `MyApplication.app` into the manifest XML. This is a workaround to enable applications to run on Windows Server 2003 Service Pack 3.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <requestedExecutionLevel level="asInvoker"/>
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

See also

- [Visual Basic Command-Line Compiler](#)
- [-nowin32manifest \(Visual Basic\)](#)

-win32resource

10/7/2019 • 2 minutes to read • [Edit Online](#)

Inserts a Win32 resource file in the output file.

Syntax

```
-win32resource:filename
```

Arguments

`filename`

The name of the resource file to add to your output file. Enclose the file name in quotation marks (" ") if it contains a space.

Remarks

You can create a Win32 resource file with the Microsoft Windows Resource Compiler (RC).

A Win32 resource can contain version or bitmap (icon) information that helps identify your application in **File Explorer**. If you do not specify `-win32resource`, the compiler generates version information based on the assembly version. The `-win32resource` and `-win32icon` options are mutually exclusive.

See [-linkresource \(Visual Basic\)](#) to reference a .NET Framework resource file, or [-resource \(Visual Basic\)](#) to attach a .NET Framework resource file.

NOTE

The `-win32resource` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code compiles `In.vb` and attaches a Win32 resource file, `Rf.res`:

```
vbc -win32resource:rf.res in.vb
```

See also

- [Visual Basic Command-Line Compiler](#)
- [Sample Compilation Command Lines](#)

Visual Basic compiler options listed by category

9/23/2019 • 3 minutes to read • [Edit Online](#)

The Visual Basic command-line compiler is provided as an alternative to compiling programs from within the Visual Studio integrated development environment (IDE). The following is a list of the Visual Basic command-line compiler options sorted by functional category.

Every compiler option is available in two forms: **-option** and **/option**. The documentation only shows the -option form.

Compiler output

OPTION	PURPOSE
<code>-nologo</code>	Suppresses compiler banner information.
<code>-utf8output</code>	Displays compiler output using UTF-8 encoding.
<code>-verbose</code>	Outputs extra information during compilation.
<code>-modulename:<string></code>	Specify the name of the source module
<code>-preferreduilang</code>	Specify a language for compiler output.

Optimization

OPTION	PURPOSE
<code>-filealign</code>	Specifies where to align the sections of the output file.
<code>-optimize</code>	Enables/disables optimizations.

Output files

OPTION	PURPOSE
<code>-doc</code>	Process documentation comments to an XML file.
<code>-deterministic</code>	Causes the compiler to output an assembly whose binary content is identical across compilations if inputs are identical.
<code>-netcf</code>	Sets the compiler to target the .NET Compact Framework.
<code>-out</code>	Specifies an output file.
<code>-refonly</code>	Outputs only a reference assembly.
<code>-refout</code>	Specifies the output path of a reference assembly.

OPTION	PURPOSE
-target	Specifies the format of the output.

.NET assemblies

OPTION	PURPOSE
-addmodule	Causes the compiler to make all type information from the specified file(s) available to the project you are currently compiling.
-delaysign	Specifies whether the assembly will be fully or partially signed.
-imports	Imports a namespace from a specified assembly.
-keycontainer	Specifies a key container name for a key pair to give an assembly a strong name.
-keyfile	Specifies a file containing a key or key pair to give an assembly a strong name.
-libpath	Specifies the location of assemblies referenced by the -reference option.
-reference	Imports metadata from an assembly.
-moduleassemblyname	Specifies the name of the assembly that a module will be a part of.
-analyzer	Run the analyzers from this assembly (Short form: -a)
-additionalfile	Names additional files that don't directly affect code generation but may be used by analyzers for producing errors or warnings.

Debugging/error checking

OPTION	PURPOSE
-bugreport	Creates a file that contains information that makes it easy to report a bug.
-debug	Produces debugging information.
-nowarn	Suppresses the compiler's ability to generate warnings.
-quiet	Prevents the compiler from displaying code for syntax-related errors and warnings.
-removeintchecks	Disables integer overflow checking.
-warnaserror	Promotes warnings to errors.

OPTION	PURPOSE
<code>-ruleset:<file></code>	Specify a ruleset file that disables specific diagnostics.

Help

OPTION	PURPOSE
<code>-?</code>	Displays the compiler options. This command is the same as specifying the <code>-help</code> option. No compilation occurs.
<code>-help</code>	Displays the compiler options. This command is the same as specifying the <code>-?</code> option. No compilation occurs.

Language

OPTION	PURPOSE
<code>-langversion</code>	Specify language version: 9 9.0 10 10.0 11 11.0.
<code>-optionexplicit</code>	Enforces explicit declaration of variables.
<code>-optionstrict</code>	Enforces strict type semantics.
<code>-optioncompare</code>	Specifies whether string comparisons should be binary or use locale-specific text semantics.
<code>-optioninfer</code>	Enables the use of local type inference in variable declarations.

Preprocessor

OPTION	PURPOSE
<code>-define</code>	Defines symbols for conditional compilation.

Resources

OPTION	PURPOSE
<code>-linkresource</code>	Creates a link to a managed resource.
<code>-resource</code>	Embeds a managed resource in an assembly.
<code>-win32icon</code>	Inserts an .ico file into the output file.
<code>-win32resource</code>	Inserts a Win32 resource into the output file.

Miscellaneous

OPTION	PURPOSE
@ (Specify Response File)	Specifies a response file.
-baseaddress	Specifies the base address of a DLL.
-codepage	Specifies the code page to use for all source code files in the compilation.
-errorreport	Specifies how the Visual Basic compiler should report internal compiler errors.
-highentropyva	Tells the Windows kernel whether a particular executable supports high entropy Address Space Layout Randomization (ASLR).
-main	Specifies the class that contains the <code>Sub Main</code> procedure to use at startup.
-noconfig	Do not compile with Vbc.rsp
-nostdlib	Causes the compiler not to reference the standard libraries.
-nowin32manifest	Instructs the compiler not to embed any application manifest into the executable file.
-platform	Specifies the processor platform the compiler targets for the output file.
-recurse	Searches subdirectories for source files to compile.
-rootnamespace	Specifies a namespace for all type declarations.
-sdkpath	Specifies the location of Mscorlib.dll and Microsoft.VisualBasic.dll.
-vbruntime	Specifies that the compiler should compile without a reference to the Visual Basic Runtime Library, or with a reference to a specific runtime library.
-win32manifest	Identifies a user-defined Win32 application manifest file to be embedded into a project's portable executable (PE) file.
<code>-parallel[+ -]</code>	Specifies whether to use concurrent build (+).
<code>-checksumalgorithm:<alg></code>	<p>Specify the algorithm for calculating the source file checksum stored in PDB. Supported values are: SHA1 (default) or SHA256.</p> <p>Due to collision problems with SHA1, Microsoft recommends SHA256 or better.</p>

See also

- [Visual Basic Compiler Options Listed Alphabetically](#)
- [Manage project and solution properties](#)

.NET Framework Reference Information (Visual Basic)

5/14/2019 • 2 minutes to read • [Edit Online](#)

This topic provides links to information about how to work with the .NET Framework class library.

Related Sections

[Getting Started](#)

Provides a comprehensive overview of the .NET Framework and links to additional resources.

[Class Library Overview](#)

Introduces the classes, interfaces, and value types that help expedite and optimize the development process and provide access to system functionality.

[Development Guide](#)

Provides a guide to all key technology areas and tasks for application development, including creating, configuring, debugging, securing, and deploying your application. This topic also provides information about dynamic programming, interoperability, extensibility, memory management, and threading.

[Tools](#)

Describes the tools that you can use to develop, configure, and deploy applications by using .NET Framework technologies.

[.NET API Browser](#)

Provides syntax, code examples, and related information for each class in the .NET Framework namespaces.

Visual Basic Language Specification

4/2/2019 • 2 minutes to read • [Edit Online](#)

The Visual Basic Language Specification is the authoritative source for answers to all questions about Visual Basic grammar and syntax. It contains detailed information about the language, including many points not covered in the Visual Basic reference documentation.

The specification is available on the [Microsoft Download Center](#).

This site contains the [VB 11 specification](#). It's built from the Markdown files contained in [the dotnet/vblang GitHub repository](#).

Issues on the specification should be created in the [dotnet/vblang](#) repository. Or, if you're interested in fixing any errors you find, you may submit a [Pull Request](#) to the same repository.

See also

- [Visual Basic Language Reference](#)

NEXT

2 minutes to read

Visual Basic Language Walkthroughs

9/13/2019 • 2 minutes to read • [Edit Online](#)

Walkthroughs give step-by-step instructions for common scenarios, which makes them a good place to start learning about the product or a particular feature area.

[Writing an Async Program](#)

Shows how to create an asynchronous solution by using `Async` and `Await`.

[Declaring and Raising Events](#)

Illustrates how events are declared and raised in Visual Basic.

[Handling Events](#)

Shows how to handle events using either the standard `WithEvents` keyword or the new `AddHandler` / `RemoveHandler` keywords.

[Creating and Implementing Interfaces](#)

Shows how interfaces are declared and implemented in Visual Basic.

[Defining Classes](#)

Describes how to declare a class and its fields, properties, methods, and events.

[Writing Queries in Visual Basic](#)

Demonstrates how you can use Visual Basic language features to write Language-Integrated Query (LINQ) query expressions.

[Implementing IEnumerable\(Of T\) in Visual Basic](#)

Demonstrates how to create a class that implements the `IEnumerable(of String)` interface and a class that implements the `IEnumerator(of String)` interface to read a text file one line at a time.

[Calling Windows APIs](#)

Explains how to use `Declare` statements and call Windows APIs. Includes information about using attributes to control marshaling for the API call and how to expose an API call as a method of a class.

[Creating COM Objects with Visual Basic](#)

Demonstrates how to create COM objects in Visual Basic, both with and without the COM class template.

[Implementing Inheritance with COM Objects](#)

Demonstrates how to use Visual Basic 6.0 to create a COM object containing a class, and then use it as a base class in Visual Basic.

[Determining Where My.Application.Log Writes Information](#)

Describes the default `My.Application.Log` settings and how to determine the settings for your application.

[Changing Where My.Application.Log Writes Information](#)

Shows how to override the default `My.Application.Log` and `My.Log` settings for logging event information and cause the `Log` object to write to other log listeners.

[Filtering My.Application.Log Output](#)

Demonstrates how to change the default log filtering for the `My.Application.Log` object.

[Creating Custom Log Listeners](#)

Demonstrates how to create a custom log listener and configure it to listen to the output of the `My.Application.Log` object.

[Embedding Types from Managed Assemblies](#)

Describes how to create an assembly and a client program that embeds types from it.

[Validating That Passwords Are Complex \(Visual Basic\)](#)

Demonstrates how to check for strong-password characteristics and update a string parameter with information about which checks a password fails.

[Encrypting and Decrypting Strings in Visual Basic](#)

Shows how to use the [DESCryptoServiceProvider](#) class to encrypt and decrypt strings.

[Manipulating Files and Folders in Visual Basic](#)

Demonstrates how to use Visual Basic functions to determine information about a file, search for a string in a file, and write to a file.

[Manipulating Files Using .NET Framework Methods](#)

Demonstrates how to use .NET Framework methods to determine information about a file, search for a string in a file, and write to a file.

[Persisting an Object in Visual Basic](#)

Demonstrates how to create a simple object and persist its data to a file.

[Walkthrough: Test-First Support with the Generate From Usage Feature](#)

Demonstrates how to do test-first development, in which you first write unit tests and then write the source code to make the tests succeed.