Visual Basic Language Reference

Visual Studio 2015

This section provides reference information for various aspects of the Visual Basic language.

In This Section

Typographic and Code Conventions (Visual Basic)
: 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 (Visual Basic)
: Lists all Visual Basic keywords and provides links to more information.

Attributes (Visual Basic)
: Documents the attributes available in Visual Basic.

Constants and Enumerations (Visual Basic)
: Documents the constants and enumerations available in Visual Basic.

Data Type Summary (Visual Basic)
: Documents the data types available in Visual Basic.

Directives (Visual Basic)
: Documents the compiler directives available in Visual Basic.

Functions (Visual Basic)
: Documents the run-time functions available in Visual Basic.

Modifiers (Visual Basic)
: Lists the Visual Basic run-time modifiers and provides links to more information.

Modules (Visual Basic)
: Documents the modules available in Visual Basic and their members.

Nothing (Visual Basic)
: Describes the default value of any data type.

Objects (Visual Basic)
: Documents the objects available in Visual Basic and their members.

Operators (Visual Basic)
: Documents the operators available in Visual Basic.

Properties (Visual Basic)

Documents the properties available in Visual Basic.

Queries (Visual Basic)
: Provides reference information about using Language-Integrated Query (LINQ) expressions in your code.

Statements (Visual Basic)
: Documents the declaration and executable statements available in Visual Basic.

Recommended XML Tags for Documentation Comments (Visual Basic)
: Describes the documentation comments for which IntelliSense is provided in the Visual Basic Code Editor.

XML Axis Properties (Visual Basic)
: Provides links to information about using XML axis properties to access XML directly in your code.

XML Literals (Visual Basic)
: Provides links to information about using XML literals to incorporate XML directly in your code.

Error Messages (Visual Basic)
: 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).

Modules (Visual Basic)

Visual Studio 2015

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

Operators (Visual Basic)

Visual Studio 2015

In This Section

Operator Precedence in Visual Basic

Operators Listed by Functionality (Visual Basic)

Data Types of Operator Results (Visual Basic)

DirectCast Operator (Visual Basic)

TryCast Operator (Visual Basic)

New Operator

Arithmetic Operators

Assignment Operators

Bit Shift Operators

Comparison Operators

Concatenation Operators

Logical/Bitwise Operators

Miscellaneous Operators

Related Sections

Visual Basic Language Reference

Visual Basic

Operators Listed by Functionality (Visual Basic)

Visual Studio 2015

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 in Visual Basic
Operator Precedence in Visual Basic

Miscellaneous Operators (Visual Basic)

Visual Studio 2015

The following are miscellaneous operators defined in Visual Basic.

AddressOf Operator (Visual Basic)

Await Operator (Visual Basic)

GetType Operator (Visual Basic)

Function Expression (Visual Basic)

If Operator (Visual Basic)

TypeOf Operator (Visual Basic)

See Also

Operators Listed by Functionality (Visual Basic)

Properties (Visual Basic)

Visual Studio 2015

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).

Visual Basic Properties

DateString	Returns or sets a **String** value representing the current date according to your system.
Now	Returns a **Date** value containing the current date and time according to your system.
ScriptEngine	Returns a **String** representing the runtime currently in use.
ScriptEngineBuildVersion	Returns an **Integer** containing the build version number of the runtime currently in use.
ScriptEngineMajorVersion	Returns an **Integer** containing the major version number of the runtime currently in use.
ScriptEngineMinorVersion	Returns an **Integer** containing the minor version number of the runtime currently in use.
TimeOfDay	Returns or sets a **Date** value containing the current time of day according to your system.
Timer	Returns a **Double** value representing the number of seconds elapsed since midnight.
TimeString	Returns or sets a **String** value representing the current time of day according to your system.
Today	Returns or sets a **Date** value containing the current date according to your system.

See Also

Visual Basic Language Reference
Visual Basic

Queries (Visual Basic)

Visual Studio 2015

Visual Basic enables you to create Language-Integrated Query (LINQ) expressions in your code.

In This Section

Aggregate Clause (Visual Basic)
: Describes the **Aggregate** clause, which applies one or more aggregate functions to a collection.

Distinct Clause (Visual Basic)
: Describes the **Distinct** clause, which restricts the values of the current range variable to eliminate duplicate values in query results.

From Clause (Visual Basic)
: Describes the **From** clause, which specifies a collection and a range variable for a query.

Group By Clause (Visual Basic)
: 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 (Visual Basic)
: Describes the **Group Join** clause, which combines two collections into a single hierarchical collection.

Join Clause (Visual Basic)
: Describes the **Join** clause, which combines two collections into a single collection.

Let Clause (Visual Basic)
: Describes the **Let** clause, which computes a value and assigns it to a new variable in the query.

Order By Clause (Visual Basic)
: Describes the **Order By** clause, which specifies the sort order for columns in a query.

Select Clause (Visual Basic)
: Describes the **Select** clause, which declares a set of range variables for a query.

Skip Clause (Visual Basic)
: Describes the **Skip** clause, which bypasses a specified number of elements in a collection and then returns the remaining elements.

Skip While Clause (Visual Basic)
: 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 (Visual Basic)
: Describes the **Take** clause, which returns a specified number of contiguous elements from the start of a collection.

Take While Clause (Visual Basic)
: 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 (Visual Basic)
: Describes the **Where** clause, which specifies a filtering condition for a query.

See Also

LINQ in Visual Basic
Introduction to LINQ in Visual Basic

Statements (Visual Basic)

Visual Studio 2015

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 (Visual Basic)

Declaration Contexts and Default Access Levels (Visual Basic)

Attribute List (Visual Basic)

Parameter List (Visual Basic)

Type List (Visual Basic)

Related Sections

Visual Basic Language Reference

Visual Basic

A-E Statements

Visual Studio 2015

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

F-P Statements

Visual Studio 2015

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

Q-Z Statements

Visual Studio 2015

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

Try...Catch...Finally Statement (Visual Basic)

Visual Studio 2015

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
tryStatements	Optional. Statement(s) where an error can occur. Can be a compound statement.
Catch	Optional. Multiple **Catch** blocks permitted. If an exception occurs when processing the **Try** block, each **Catch** statement is examined in textual order to determine whether it handles the exception, with *exception* representing the exception that has been thrown.
exception	Optional. Any variable name. The initial value of *exception* is the value of the thrown error. Used with **Catch** to specify the error caught. If omitted, the **Catch** statement catches any exception.
type	Optional. Specifies the type of class filter. If the value of *exception* is of the type specified by *type* or of a derived type, the identifier becomes bound to the exception object.
When	Optional. A **Catch** statement with a **When** clause catches exceptions only when *expression* evaluates to **True**. A **When** clause is applied only after checking the type of the exception, and *expression* may refer to the identifier representing the exception.

expression	Optional. Must be implicitly convertible to **Boolean**. Any expression that describes a generic filter. Typically used to filter by error number. Used with **When** keyword to specify circumstances under which the error is caught.
catchStatements	Optional. Statement(s) to handle errors that occur in the associated **Try** block. Can be a compound statement.
Exit Try	Optional. Keyword that breaks out of the **Try...Catch...Finally** structure. Execution resumes with the code immediately following the **End Try** statement. The **Finally** statement will still be executed. Not allowed in **Finally** blocks.
Finally	Optional. A **Finally** block is always executed when execution leaves any part of the **Try...Catch** statement.
finallyStatements	Optional. Statement(s) that are executed after all other error processing has occurred.
End Try	Terminates the **Try...Catch...Finally** 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 (Visual Basic).

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.

VB

```
Private Sub TextFileExample(ByVal filePath As String)

    ' Verify that the file exists.
    If System.IO.File.Exists(filePath) = False Then
        Console.Write("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.Write(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 (C# and Visual Basic).

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 (Visual Basic).

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 (C# and Visual Basic) 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.

VB

```
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.

Example

The following example illustrates the structure of the **Try...Catch...Finally** statement.

VB

```
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
```

Example

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.

VB

```
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
```

Example

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.

VB

```
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
```

Example

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.

VB

```
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
```

Example

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**.

VB

```
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
```

Example

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.

VB

```
    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 (Visual Basic)
- On Error Statement (Visual Basic)
- Best Practices for Using Code Snippets
- Exception Handling (Task Parallel Library)
- Throw Statement (Visual Basic)

Clauses (Visual Basic)

Visual Studio 2015

The topics in this section document Visual Basic run-time clauses.

In This Section

Alias Clause (Visual Basic)

As Clause (Visual Basic)

Handles Clause (Visual Basic)

Implements Clause (Visual Basic)

In Clause (Visual Basic)

Into Clause (Visual Basic)

Of Clause (Visual Basic)

Related Sections

Visual Basic Language Reference

Visual Basic

Alias Clause (Visual Basic)

Visual Studio 2015

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.

VB

```
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 (Visual Basic)

Implements Clause (Visual Basic)

Visual Studio 2015

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 (Visual Basic) 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 (Visual Basic) to be reimplemented.
- You can reimplement the member with a different name.

The **Implements** keyword can be used in these contexts:

Event Statement

Function Statement (Visual Basic)

Property Statement

Sub Statement (Visual Basic)

See Also

Implements Statement
Interface Statement (Visual Basic)
Class Statement (Visual Basic)
Structure Statement

Attribute List (Visual Basic)

Visual Studio 2015

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

attributemodifier
Required for attributes applied at the beginning of a source file. Can be Assembly (Visual Basic) or Module.

attributename
Required. Name of the attribute.

attributearguments
Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.

attributeinitializer
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 (C# and Visual Basic). For information on attribute names, see Declared Element Names (Visual Basic).

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.

VB

```
<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 (Visual Basic)
Module <keyword> (Visual Basic)
Attributes (C# and Visual Basic)
How to: Break and Combine Statements in Code (Visual Basic)

Parameter List (Visual Basic)

Visual Studio 2015

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 (Visual Basic) 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 (Visual Basic).

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 (Visual Basic).

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 (Visual Basic).

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 (Visual Basic).

- **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 (Visual Basic).

- **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 (Visual Basic).

Example

The following example shows a **Function** procedure that defines two parameters.

VB

```
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 (Visual Basic)
Sub Statement (Visual Basic)
Declare Statement
Structure Statement
Option Strict Statement
Attributes (C# and Visual Basic)
How to: Break and Combine Statements in Code (Visual Basic)

Access Levels in Visual Basic

Visual Studio 2015

The *access level* of a declared element is the extent of the ability to access it, that is, what code has permission to read it or write to it. This is determined not only by how you declare the element itself, but also by the access level of the element's container. Code that cannot access a containing element cannot access any of its contained elements, even those declared as **Public**. For example, a **Public** variable in a **Private** structure can be accessed from inside the class that contains the structure, but not from outside that class.

Public

The Public (Visual Basic) keyword in the declaration statement specifies that the elements 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. The following code shows a sample **Public** declaration.

```
Public Class classForEverybody
```

You can use **Public** only at module, interface, or namespace level. This means you can declare a public element at the level of a source file or namespace, or inside an interface, module, class, or structure, but not in a procedure.

Protected

The Protected (Visual Basic) keyword in the declaration statement specifies that the elements can be accessed only from within the same class, or from a class derived from this class. The following code shows a sample **Protected** declaration.

```
Protected Class classForMyHeirs
```

You can use **Protected** only at class level, and only when you declare a member of a class. This means you can declare a protected element in a class, but not at the level of a source file or namespace, or inside an interface, module, structure, or procedure.

Friend

The Friend (Visual Basic) keyword in the declaration statement specifies that the elements can be accessed from within the same assembly, but not from outside the assembly. The following code shows a sample **Friend** declaration.

```
Friend stringForThisProject As String
```

You can use **Friend** only at module, interface, or namespace level. This means you can declare a friend element at the level of a source file or namespace, or inside an interface, module, class, or structure, but not in a procedure.

Protected Friend

The **Protected** and **Friend** keywords together in the declaration statement specify that the elements can be accessed either from derived classes or from within the same assembly, or both. The following code shows a sample **Protected Friend** declaration.

```
Protected Friend stringForProjectAndHeirs As String
```

You can use **Protected Friend** only at class level, and only when you declare a member of a class. This means you can declare a protected friend element in a class, but not at the level of a source file or namespace, or inside an interface, module, structure, or procedure.

Private

The Private (Visual Basic) keyword in the declaration statement specifies that the elements can be accessed only from within the same module, class, or structure. The following code shows a sample **Private** declaration.

```
Private numberForMeOnly As Integer
```

You can use **Private** only at module level. This means you can declare a private element inside a module, class, or structure, but not at the level of a source file or namespace, inside an interface, or in a procedure.

At the module level, the **Dim** statement without any access level keywords is equivalent to a **Private** declaration. However, you might want to use the **Private** keyword to make your code easier to read and interpret.

Access Modifiers

The keywords that specify access level are called *access modifiers*. The following table compares the access modifiers.

Access modifier	Access level granted	Elements you can declare with this access level	Declaration context within which you can use this modifier

Public	Unrestricted: Any code that can see a public element can access it	Interfaces Modules Classes Structures Structure members Procedures Properties Member variables Constants Enumerations Events External declarations Delegates	Source file Namespace Interface Module Class Structure
Protected	Derivational: Code in the class that declares a protected element, or a class derived from it, can access the element	Interfaces Classes Structures Procedures Properties Member variables Constants Enumerations Events External declarations Delegates	Class
Friend	Assembly: Code in the assembly that declares a friend element can access it	Interfaces Modules Classes Structures	Source file Namespace Interface Module

		Structure members Procedures Properties Member variables Constants Enumerations Events External declarations Delegates	Class Structure
Protected Friend	Union of **Protected** and **Friend**: Code in the same class or the same assembly as a protected friend element, or within any class derived from the element's class, can access it	Interfaces Classes Structures Procedures Properties Member variables Constants Enumerations Events External declarations Delegates	Class
Private	Declaration context: Code in the type that declares a private element, including code within contained types, can access the element	Interfaces Classes Structures Structure members Procedures Properties Member variables Constants	Module Class Structure

		Enumerations Events External declarations Delegates	

See Also

Dim Statement (Visual Basic)
Static (Visual Basic)
Declared Element Names (Visual Basic)
References to Declared Elements (Visual Basic)
Declared Element Characteristics (Visual Basic)
Lifetime in Visual Basic
Scope in Visual Basic
How to: Control the Availability of a Variable (Visual Basic)
Variables in Visual Basic
Variable Declaration in Visual Basic

Declaration Contexts and Default Access Levels (Visual Basic)

Visual Studio 2015

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 (Visual Basic))	Not allowed	**Private** (**Public** in **Structure**, not allowed in **Interface**)	**Public**
Constant (Const Statement (Visual Basic))	Not allowed	**Private** (**Public** in **Structure**, not allowed in **Interface**)	**Public**
Enumeration (Enum Statement (Visual Basic))	**Friend**	**Public**	Not allowed
Class (Class Statement (Visual Basic))	**Friend**	**Public**	Not allowed
Structure (Structure Statement)	**Friend**	**Public**	Not allowed
Module (Module Statement)	**Friend**	Not allowed	Not allowed
Interface (Interface Statement (Visual Basic))	**Friend**	**Public**	Not allowed

Procedure (Function Statement (Visual Basic), Sub Statement (Visual Basic))	Not allowed	**Public**	Not allowed
External reference (Declare Statement)	Not allowed	**Public** (not allowed in **Interface**)	Not allowed
Operator (Operator Statement)	Not allowed	**Public** (not allowed in **Interface** or **Module**)	Not allowed
Property (Property Statement)	Not allowed	**Public**	Not allowed
Default property (Default (Visual Basic))	Not allowed	**Public** (not allowed in **Module**)	Not allowed
Event (Event Statement)	Not allowed	**Public**	Not allowed
Delegate (Delegate Statement)	**Friend**	**Public**	Not allowed

For more information, see Access Levels in Visual Basic.

See Also

Friend (Visual Basic)
Private (Visual Basic)
Public (Visual Basic)

How to: Create XML Documentation in Visual Basic

Visual Studio 2015

This example shows how to add XML documentation comments to your code.

> **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 Visual Studio IDE.

To create XML documentation for a type or member

1. In the **Code Editor**, position your cursor on the line above the type or member for which you want to create documentation.
2. Type `'''` (three single-quotation marks).

   An XML skeleton for the type or member is added in the **Code Editor**.
3. Add descriptive information between the appropriate tags.

   > **Note**
   >
   > If you add additional lines within the XML documentation block, each line must begin with `'''`.
4. Add additional code that uses the type or member with the new XML documentation comments.

   IntelliSense displays the text from the <summary> tag for the type or member.
5. Compile the code to generate an XML file containing the documentation comments. For more information, see /doc.

See Also

Documenting Your Code with XML (Visual Basic)
Recommended XML Tags for Documentation Comments (Visual Basic)
/doc

Recommended XML Tags for Documentation Comments (Visual Basic)

Visual Studio 2015

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 (Visual Basic)
/doc
How to: Create XML Documentation in Visual Basic