Procedures in Visual Basic

Visual Studio 2015

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 (Visual Basic), the appropriate Exit Statement (Visual Basic) statement for the procedure, or the procedure's End <keyword> Statement (Visual Basic) 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 (Visual Basic) 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 (Visual Basic) return a value to the calling code. They can perform other actions before returning.
- Property Procedures (Visual Basic) return and assign values of properties on objects or modules.
- Operator Procedures (Visual Basic) 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 (Visual Basic)

Visual Studio 2015

You enclose a procedure between a starting declaration statement (**Sub** or **Function**) and an ending declaration statement (**End Sub** or **End Function**). All the procedure's code lies between these statements.

A procedure cannot contain another procedure, so its starting and ending statements must be outside any other procedure.

If you have code that performs the same task in different places, you can write the task once as a procedure and then call it from different places in your code.

To create a procedure that does not return a value

1. Outside any other procedure, use a **Sub** statement, followed by an **End Sub** statement.
2. In the **Sub** statement, follow the **Sub** keyword with the name of the procedure, then the parameter list in parentheses.
3. Place the procedure's code statements between the **Sub** and **End Sub** statements.

To create a procedure that returns a value

1. Outside any other procedure, use a **Function** statement, followed by an **End Function** statement.
2. In the **Function** statement, follow the **Function** keyword with the name of the procedure, then the parameter list in parentheses, and then an **As** clause specifying the data type of the return value.
3. Place the procedure's code statements between the **Function** and **End Function** statements.
4. Use a **Return** statement to return the value to the calling code.

To connect your new procedure with the old, repetitive blocks of code

1. Make sure you define the new procedure in a place where the old code has access to it.
2. In your old, repetitive code block, replace the statements that perform the repetitive task with a single statement that calls the **Sub** or **Function** procedure.
3. If your procedure is a **Function** that returns a value, ensure that your calling statement performs an action with the returned value, such as storing it in a variable, or else the value will be lost.

Example

The following **Function** procedure calculates the longest side, or hypotenuse, of a right triangle, given the values for the

other two sides.

VB

```
Function hypotenuse(ByVal side1 As Single, ByVal side2 As Single) As Single
    Return Math.Sqrt((side1 ^ 2) + (side2 ^ 2))
End Function
```

See Also

Procedures in Visual Basic
Sub Procedures (Visual Basic)
Function Procedures (Visual Basic)
Property Procedures (Visual Basic)
Operator Procedures (Visual Basic)
Procedure Parameters and Arguments (Visual Basic)
Recursive Procedures (Visual Basic)
Procedure Overloading (Visual Basic)
Objects and Classes in Visual Basic
Object-Oriented Programming (C# and Visual Basic)

Sub Procedures (Visual Basic)

Visual Studio 2015

A **Sub** procedure is a series of Visual Basic statements enclosed by the **Sub** and **End Sub** statements. The **Sub** procedure performs a task and then returns control to the calling code, but it does not return a value to the calling code.

Each time the procedure is called, its statements are executed, starting with the first executable statement after the **Sub** statement and ending with the first **End Sub**, **Exit Sub**, or **Return** statement encountered.

You can define a **Sub** procedure in modules, classes, and structures. By default, it is **Public**, which means you can call it from anywhere in your application that has access to the module, class, or structure in which you defined it. The term, *method*, describes a **Sub** or **Function** procedure that is accessed from outside its defining module, class, or structure. For more information, see Procedures in Visual Basic.

A **Sub** procedure can take arguments, such as constants, variables, or expressions, which are passed to it by the calling code.

Declaration Syntax

The syntax for declaring a **Sub** procedure is as follows:

```
[modifiers] Sub subname[(parameterlist)]

' Statements of the Sub procedure.

End Sub
```

The *modifiers* can specify access level and information about overloading, overriding, sharing, and shadowing. For more information, see Sub Statement (Visual Basic).

Parameter Declaration

You declare each procedure parameter similarly to how you declare a variable, specifying the parameter name and data type. You can also specify the passing mechanism, and whether the parameter is optional or a parameter array.

The syntax for each parameter in the parameter list is as follows:

```
[Optional] [ByVal | ByRef] [ParamArray] parametername As datatype
```

If the parameter is optional, you must also supply a default value as part of its declaration. The syntax for specifying a default value is as follows:

```
Optional [ByVal | ByRef] parametername As datatype = defaultvalue
```

Parameters as Local Variables

When control passes to the procedure, each parameter is treated as a local variable. This means that its lifetime is the

same as that of the procedure, and its scope is the whole procedure.

Calling Syntax

You invoke a **Sub** procedure explicitly with a stand-alone calling statement. You cannot call it by using its name in an expression. You must provide values for all arguments that are not optional, and you must enclose the argument list in parentheses. If no arguments are supplied, you can optionally omit the parentheses. The use of the **Call** keyword is optional but not recommended.

The syntax for a call to a **Sub** procedure is as follows:

`[Call]` *subname*`[(`*argumentlist*`)]`

You can call a **Sub** method from outside the class that defines it. First, you have to use the **New** keyword to create an instance of the class, or call a method that returns an instance of the class. For more information, see New Operator (Visual Basic). Then, you can use the following syntax to call the **Sub** method on the instance object:

Object.methodname`[(`*argumentlist*`)]`

Illustration of Declaration and Call

The following **Sub** procedure tells the computer operator which task the application is about to perform, and also displays a time stamp. Instead of duplicating this code at the start of every task, the application just calls `tellOperator` from various locations. Each call passes a string in the `task` argument that identifies the task being started.

VB

```
Sub tellOperator(ByVal task As String)
    Dim stamp As Date
    stamp = TimeOfDay()
    MsgBox("Starting " & task & " at " & CStr(stamp))
End Sub
```

The following example shows a typical call to `tellOperator`.

VB

```
tellOperator("file update")
```

See Also

Procedures in Visual Basic
Function Procedures (Visual Basic)

Property Procedures (Visual Basic)
Operator Procedures (Visual Basic)
Procedure Parameters and Arguments (Visual Basic)
Sub Statement (Visual Basic)
How to: Call a Procedure that Does Not Return a Value (Visual Basic)
How to: Call an Event Handler in Visual Basic

How to: Call a Procedure that Does Not Return a Value (Visual Basic)

Visual Studio 2015

A **Sub** procedure does not return a value to the calling code. You call it explicitly with a stand-alone calling statement. You cannot call it by simply using its name within an expression.

To call a Sub procedure

1. Specify the name of the **Sub** procedure.

2. Follow the procedure name with parentheses to enclose the argument list. If there are no arguments, you can optionally omit the parentheses. However, using the parentheses makes your code easier to read.

3. Place the arguments in the argument list within the parentheses, separated by commas. Be sure you supply the arguments in the same order that the **Sub** procedure defines the corresponding parameters.

   The following example calls the Visual Basic AppActivate function to activate an application window. AppActivate takes the window title as its sole argument. It does not return a value to the calling code. If a Notepad process is not running, the example throws an ArgumentException. The **Shell** procedure assumes the applications are in the paths specified.

   VB

```
Dim notepadID As Integer
' Activate a running Notepad process.
AppActivate("Untitled - Notepad")
' AppActivate can also use the return value of the Shell function.
' Shell runs a new instance of Notepad.
notepadID = Shell("C:\WINNT\NOTEPAD.EXE", AppWinStyle.NormalFocus)
' Activate the new instance of Notepad.
AppActivate(notepadID)
```

See Also

Shell
ArgumentException
Procedures in Visual Basic
Sub Procedures (Visual Basic)
Procedure Parameters and Arguments (Visual Basic)
Sub Statement (Visual Basic)
How to: Create a Procedure (Visual Basic)
How to: Call a Procedure That Returns a Value (Visual Basic)
How to: Call an Event Handler in Visual Basic

How to: Call an Event Handler in Visual Basic

Visual Studio 2015

An *event* is an action or occurrence — such as a mouse click or a credit limit exceeded — that is recognized by some program component, and for which you can write code to respond. An *event handler* is the code you write to respond to an event.

An event handler in Visual Basic is a **Sub** procedure. However, you do not normally call it the same way as other **Sub** procedures. Instead, you identify the procedure as a handler for the event. You can do this either with a Handles Clause (Visual Basic) clause and a WithEvents (Visual Basic) variable, or with an AddHandler Statement. Using a **Handles** clause is the default way to declare an event handler in Visual Basic. This is the way the event handlers are written by the designers when you program in the integrated development environment (IDE). The **AddHandler** statement is suitable for raising events dynamically at run time.

When the event occurs, Visual Basic automatically calls the event handler procedure. Any code that has access to the event can cause it to occur by executing a RaiseEvent Statement.

You can associate more than one event handler with the same event. In some cases you can dissociate a handler from an event. For more information, see Events (Visual Basic).

To call an event handler using Handles and WithEvents

1. Make sure the event is declared with an Event Statement.
2. Declare an object variable at module or class level, using the WithEvents (Visual Basic) keyword. The **As** clause for this variable must specify the class that raises the event.
3. In the declaration of the event-handling **Sub** procedure, add a Handles Clause (Visual Basic) clause that specifies the **WithEvents** variable and the event name.
4. When the event occurs, Visual Basic automatically calls the **Sub** procedure. Your code can use a **RaiseEvent** statement to make the event occur.

   The following example defines an event and a **WithEvents** variable that refers to the class that raises the event. The event-handling **Sub** procedure uses a **Handles** clause to specify the class and event it handles.

   VB

```
Public Class raisesEvent
    Public Event somethingHappened()
    Dim WithEvents happenObj As New raisesEvent
    Public Sub processHappen() Handles happenObj.somethingHappened
        ' Insert code to handle somethingHappened event.
    End Sub
End Class
```

To call an event handler using AddHandler

1. Make sure the event is declared with an **Event** statement.
2. Execute an AddHandler Statement to dynamically connect the event-handling **Sub** procedure with the event.
3. When the event occurs, Visual Basic automatically calls the **Sub** procedure. Your code can use a **RaiseEvent** statement to make the event occur.

   The following example defines a **Sub** procedure to handle the Closing event of a form. It then uses the AddHandler Statement to associate the `catchClose` procedure as an event handler for Closing.

   VB

```
' Place these procedures inside a Form class definition.
Private Sub catchClose(ByVal sender As Object, ByVal e As
System.ComponentModel.CancelEventArgs)
    ' Insert code to deal with impending closure of this form.
End Sub
Public Sub formOpened()
    AddHandler Me.Closing, AddressOf catchClose
End Sub
```

   You can dissociate an event handler from an event by executing the RemoveHandler Statement.

See Also

Procedures in Visual Basic
Sub Procedures (Visual Basic)
Sub Statement (Visual Basic)
AddressOf Operator (Visual Basic)
How to: Create a Procedure (Visual Basic)
How to: Call a Procedure that Does Not Return a Value (Visual Basic)

Function Procedures (Visual Basic)

Visual Studio 2015

A **Function** procedure is a series of Visual Basic statements enclosed by the **Function** and **End Function** statements. The **Function** procedure performs a task and then returns control to the calling code. When it returns control, it also returns a value to the calling code.

Each time the procedure is called, its statements run, starting with the first executable statement after the **Function** statement and ending with the first **End Function**, **Exit Function**, or **Return** statement encountered.

You can define a **Function** procedure in a module, class, or structure. It is **Public** by default, which means you can call it from anywhere in your application that has access to the module, class, or structure in which you defined it.

A **Function** procedure can take arguments, such as constants, variables, or expressions, which are passed to it by the calling code.

Declaration Syntax

The syntax for declaring a **Function** procedure is as follows:

VB

```
[Modifiers] Function FunctionName [(ParameterList)] As ReturnType
    [Statements]
End Function
```

The *modifiers* can specify access level and information regarding overloading, overriding, sharing, and shadowing. For more information, see Function Statement (Visual Basic).

You declare each parameter the same way you do for Sub Procedures (Visual Basic).

Data Type

Every **Function** procedure has a data type, just as every variable does. This data type is specified by the **As** clause in the **Function** statement, and it determines the data type of the value the function returns to the calling code. The following sample declarations illustrate this.

VB

```
Function yesterday() As Date
End Function

Function findSqrt(ByVal radicand As Single) As Single
End Function
```

For more information, see "Parts" in Function Statement (Visual Basic).

Returning Values

The value a **Function** procedure sends back to the calling code is called its return value. The procedure returns this value in one of two ways:

- It uses the **Return** statement to specify the return value, and returns control immediately to the calling program. The following example illustrates this.

  VB

  ```
  Function FunctionName [(ParameterList)] As ReturnType
      ' The following statement immediately transfers control back
      ' to the calling code and returns the value of Expression.
      Return Expression
  End Function
  ```

- It assigns a value to its own function name in one or more statements of the procedure. Control does not return to the calling program until an **Exit Function** or **End Function** statement is executed. The following example illustrates this.

  VB

  ```
  Function FunctionName [(ParameterList)] As ReturnType
      ‘ The following statement does not transfer control back to the calling code.
      FunctionName = Expression
      ' When control returns to the calling code, Expression is the return value.
  End Function
  ```

The advantage of assigning the return value to the function name is that control does not return from the procedure until it encounters an **Exit Function** or **End Function** statement. This allows you to assign a preliminary value and adjust it later if necessary.

For more information about returning values, see Function Statement (Visual Basic). For information about returning arrays, see Arrays in Visual Basic.

Calling Syntax

You invoke a **Function** procedure by including its name and arguments either on the right side of an assignment statement or in an expression. You must provide values for all arguments that are not optional, and you must enclose the argument list in parentheses. If no arguments are supplied, you can optionally omit the parentheses.

The syntax for a call to a **Function** procedure is as follows:

lvalue = *functionname*[(*argumentlist*)]

```
If ((functionname[(argumentlist)] / 3) <= expression) Then
```

When you call a **Function** procedure, you do not have to use its return value. If you do not, all the actions of the function are performed, but the return value is ignored. MsgBox is often called in this manner.

Illustration of Declaration and Call

The following **Function** procedure calculates the longest side, or hypotenuse, of a right triangle, given the values for the other two sides.

VB

```
Function hypotenuse(ByVal side1 As Single, ByVal side2 As Single) As Single
    Return Math.Sqrt((side1 ^ 2) + (side2 ^ 2))
End Function
```

The following example shows a typical call to `hypotenuse`.

VB

```
Dim testLength, testHypotenuse As Single
testHypotenuse = hypotenuse(testLength, 10.7)
```

See Also

Procedures in Visual Basic
Sub Procedures (Visual Basic)
Property Procedures (Visual Basic)
Operator Procedures (Visual Basic)
Procedure Parameters and Arguments (Visual Basic)
Function Statement (Visual Basic)
How to: Create a Procedure that Returns a Value (Visual Basic)
How to: Return a Value from a Procedure (Visual Basic)
How to: Call a Procedure That Returns a Value (Visual Basic)

How to: Create a Procedure that Returns a Value (Visual Basic)

Visual Studio 2015

You use a **Function** procedure to return a value to the calling code.

To create a procedure that returns a value

1. Outside any other procedure, use a **Function** statement, followed by an **End Function** statement.
2. In the **Function** statement, follow the **Function** keyword with the name of the procedure, and then the parameter list in parentheses.
3. Follow the parentheses with an **As** clause to specify the data type of the returned value.
4. Place the procedure's code statements between the **Function** and **End Function** statements.
5. Use a **Return** statement to return the value to the calling code.

   The following **Function** procedure calculates the longest side, or hypotenuse, of a right triangle, given the values for the other two sides.

   VB

   ```
   Function hypotenuse(ByVal side1 As Single, ByVal side2 As Single) As Single
       Return Math.Sqrt((side1 ^ 2) + (side2 ^ 2))
   End Function
   ```

   The following example shows a typical call to `hypotenuse`.

   VB

   ```
   Dim testLength, testHypotenuse As Single
   testHypotenuse = hypotenuse(testLength, 10.7)
   ```

See Also

Procedures in Visual Basic
Sub Procedures (Visual Basic)
Property Procedures (Visual Basic)
Operator Procedures (Visual Basic)
Procedure Parameters and Arguments (Visual Basic)
Function Statement (Visual Basic)
How to: Return a Value from a Procedure (Visual Basic)

How to: Call a Procedure That Returns a Value (Visual Basic)

How to: Call a Procedure That Returns a Value (Visual Basic)

Visual Studio 2015

A **Function** procedure returns a value to the calling code. You call it by including its name and arguments either on the right side of an assignment statement or in an expression.

To call a Function procedure within an expression

1. Use the **Function** procedure name the same way you would use a variable. You can use a **Function** procedure call anywhere you can use a variable or constant in an expression.

2. Follow the procedure name with parentheses to enclose the argument list. If there are no arguments, you can optionally omit the parentheses. However, using the parentheses makes your code easier to read.

3. Place the arguments in the argument list within the parentheses, separated by commas. Be sure you supply the arguments in the same order that the **Function** procedure defines the corresponding parameters.

   Alternatively, you can pass one or more arguments by name. For more information, see Passing Arguments by Position and by Name (Visual Basic).

4. The value returned from the procedure participates in the expression just as the value of a variable or constant would.

To call a Function procedure in an assignment statement

1. Use the **Function** procedure name following the equal (**=**) sign in the assignment statement.

2. Follow the procedure name with parentheses to enclose the argument list. If there are no arguments, you can optionally omit the parentheses. However, using the parentheses makes your code easier to read.

3. Place the arguments in the argument list within the parentheses, separated by commas. Be sure you supply the arguments in the same order that the **Function** procedure defines the corresponding parameters, unless you are passing them by name.

4. The value returned from the procedure is stored in the variable or property on the left side of the assignment statement.

Example

The following example calls the Visual Basic Environ to retrieve the value of an operating system environment variable. The first line calls **Environ** within an expression, and the second line calls it in an assignment statement. **Environ** takes the variable name as its sole argument. It returns the variable's value to the calling code.

VB

```
MsgBox("Value of PATH is " & Environ("PATH"))
Dim currentPath As String = Environ("PATH")
```

See Also

Function Procedures (Visual Basic)
Procedure Parameters and Arguments (Visual Basic)
Function Statement (Visual Basic)
How to: Create a Procedure that Returns a Value (Visual Basic)
How to: Return a Value from a Procedure (Visual Basic)
How to: Call a Procedure that Does Not Return a Value (Visual Basic)

Troubleshooting Procedures (Visual Basic)

Visual Studio 2015

This page lists some common problems that can occur when working with procedures.

Returning an Array Type from a Function Procedure

If a **Function** procedure returns an array data type, you cannot use the **Function** name to store values in the elements of the array. If you attempt to do this, the compiler interprets it as a call to the **Function**. The following example generates compiler errors.

```
Function allOnes(ByVal n As Integer) As Integer()

For i As Integer = 1 To n - 1

' The following statement generates a COMPILER ERROR.

allOnes(i) = 1

Next i

' The following statement generates a COMPILER ERROR.

Return allOnes()

End Function
```

The statement `allOnes(i) = 1` generates a compiler error because it appears to call `allOnes` with an argument of the wrong data type (a singleton **Integer** instead of an **Integer** array). The statement `Return allOnes()` generates a compiler error because it appears to call `allOnes` with no argument.

Correct Approach: To be able to modify the elements of an array that is to be returned, define an internal array as a local variable. The following example compiles without error.

VB

```
Function allOnes(ByVal n As Integer) As Integer()
    Dim i As Integer, iArray(n) As Integer
    For i = 0 To n - 1
        iArray(i) = 1
    Next i
    Return iArray
End Function
```

Argument Not Being Modified by Procedure Call

If you intend to allow a procedure to change a programming element underlying an argument in the calling code, you must pass it by reference. But a procedure can access the elements of a reference type argument even if you pass it by value.

- **Underlying Variable**. To allow the procedure to replace the value of the underlying variable element itself, the procedure must declare the parameter ByRef (Visual Basic). Also, the calling code must not enclose the argument in parentheses, because that would override the **ByRef** passing mechanism.
- **Reference Type Elements**. If you declare a parameter ByVal (Visual Basic), the procedure cannot modify the underlying variable element itself. However, if the argument is a reference type, the procedure can modify the members of the object to which it points, even though it cannot replace the variable's value. For example, if the argument is an array variable, the procedure cannot assign a new array to it, but it can change one or more of its elements. The changed elements are reflected in the underlying array variable in the calling code.

The following example defines two procedures that take an array variable by value and operate on its elements. Procedure `increase` simply adds one to each element. Procedure `replace` assigns a new array to the parameter `a()` and then adds one to each element. However, the reassignment does not affect the underlying array variable in the calling code because `a()` is declared **ByVal**.

VB

```
Public Sub increase(ByVal a() As Long)
    For j As Integer = 0 To UBound(a)
        a(j) = a(j) + 1
    Next j
End Sub
```

VB

```
Public Sub replace(ByVal a() As Long)
    Dim k() As Long = {100, 200, 300}
    a = k
    For j As Integer = 0 To UBound(a)
        a(j) = a(j) + 1
    Next j
End Sub
```

The following example makes calls to `increase` and `replace`.

VB

```
Dim n() As Long = {10, 20, 30, 40}
Call increase(n)
MsgBox("After increase(n): " & CStr(n(0)) & ", " &
    CStr(n(1)) & ", " & CStr(n(2)) & ", " & CStr(n(3)))
Call replace(n)
MsgBox("After replace(n): " & CStr(n(0)) & ", " &
```

```
        CStr(n(1)) & ", " & CStr(n(2)) & ", " & CStr(n(3)))
```

The first **MsgBox** call displays "After increase(n): 11, 21, 31, 41". Because `n` is a reference type, `increase` can change its members, even though it is passed **ByVal**.

The second **MsgBox** call displays "After replace(n): 11, 21, 31, 41". Because `n` is passed **ByVal**, `replace` cannot modify the variable `n` by assigning a new array to it. When `replace` creates the new array instance `k` and assigns it to the local variable `a`, it loses the reference to `n` passed in by the calling code. When it increments the members of `a`, only the local array `k` is affected.

Correct Approach: To be able to modify an underlying variable element itself, pass it by reference. The following example shows the change in the declaration of `replace` that allows it to replace one array with another in the calling code.

VB

```
Public Sub replace(ByRef a() As Long)
```

Unable to Define an Overload

If you want to define an overloaded version of a procedure, you must use the same name but a different signature. If the compiler cannot differentiate your declaration from an overload with the same signature, it generates an error.

The *signature* of a procedure is determined by the procedure name and the parameter list. Each overload must have the same name as all the other overloads but must differ from all of them in at least one of the other components of the signature. For more information, see Procedure Overloading (Visual Basic).

The following items, even though they pertain to the parameter list, are not components of a procedure's signature:

- Procedure modifier keywords, such as **Public**, **Shared**, and **Static**
- Parameter names
- Parameter modifier keywords, such as **ByRef** and **Optional**
- The data type of the return value (except for a conversion operator)

You cannot overload a procedure by varying only one or more of the preceding items.

Correct Approach: To be able to define a procedure overload, you must vary the signature. Because you must use the same name, you must vary the number, order, or data types of the parameters. In a generic procedure, you can vary the number of type parameters. In a conversion operator (CType Function (Visual Basic)), you can vary the return type.

Overload Resolution with Optional and ParamArray Arguments

If you are overloading a procedure with one or more Optional (Visual Basic) parameters or a ParamArray (Visual Basic) parameter, you must avoid duplicating any of the *implicit overloads*. For information, see Considerations in Overloading Procedures (Visual Basic).

Calling a Wrong Version of an Overloaded Procedure

If a procedure has several overloaded versions, you should be familiar with all their parameter lists and understand how Visual Basic resolves calls among the overloads. Otherwise you could call an overload other than the intended one.

When you have determined which overload you want to call, be careful to observe the following rules:

- Supply the correct number of arguments, and in the correct order.
- Ideally, your arguments should have the exact same data types as the corresponding parameters. In any case, the data type of each argument must widen to that of its corresponding parameter. This is true even with the Option Strict Statement set to **Off**. If an overload requires any narrowing conversion from your argument list, that overload is not eligible to be called.
- If you supply arguments that require widening, make their data types as close as possible to the corresponding parameter data types. If two or more overloads accept your argument data types, the compiler resolves your call to the overload that calls for the least amount of widening.

You can reduce the chance of data type mismatches by using the CType Function (Visual Basic) conversion keyword when preparing your arguments.

Overload Resolution Failure

When you call an overloaded procedure, the compiler attempts to eliminate all but one of the overloads. If it succeeds, it resolves the call to that overload. If it eliminates all the overloads, or if it cannot reduce the eligible overloads to a single candidate, it generates an error.

The following example illustrates the overload resolution process.

VB

```
Overloads Sub z(ByVal x As Byte, ByVal y As Double)
End Sub
Overloads Sub z(ByVal x As Short, ByVal y As Single)
End Sub
Overloads Sub z(ByVal x As Integer, ByVal y As Single)
End Sub
```

VB

```
Dim r, s As Short
Call z(r, s)
Dim p As Byte, q As Short
' The following statement causes an overload resolution error.
```

```
Call z(p, q)
```

In the first call, the compiler eliminates the first overload because the type of the first argument (**Short**) narrows to the type of the corresponding parameter (**Byte**). It then eliminates the third overload because each argument type in the second overload (**Short** and **Single**) widens to the corresponding type in the third overload (**Integer** and **Single**). The second overload requires less widening, so the compiler uses it for the call.

In the second call, the compiler cannot eliminate any of the overloads on the basis of narrowing. It eliminates the third overload for the same reason as in the first call, because it can call the second overload with less widening of the argument types. However, the compiler cannot resolve between the first and second overloads. Each has one defined parameter type that widens to the corresponding type in the other (**Byte** to **Short**, but **Single** to **Double**). The compiler therefore generates an overload resolution error.

Correct Approach: To be able to call an overloaded procedure without ambiguity, use CType Function (Visual Basic) to match the argument data types to the parameter types. The following example shows a call to `z` that forces resolution to the second overload.

VB

```
Call z(CType(p, Short), CType(q, Single))
```

Overload Resolution with Optional and ParamArray Arguments

If two overloads of a procedure have identical signatures except that the last parameter is declared Optional (Visual Basic) in one and ParamArray (Visual Basic) in the other, the compiler resolves a call to that procedure according to the closest match. For more information, see Overload Resolution (Visual Basic).

See Also

Procedures in Visual Basic
Sub Procedures (Visual Basic)
Function Procedures (Visual Basic)
Property Procedures (Visual Basic)
Operator Procedures (Visual Basic)
Procedure Parameters and Arguments (Visual Basic)
Procedure Overloading (Visual Basic)
Considerations in Overloading Procedures (Visual Basic)
Overload Resolution (Visual Basic)

Recursive Procedures (Visual Basic)

Visual Studio 2015

A *recursive* procedure is one that calls itself. In general, this is not the most effective way to write Visual Basic code.

The following procedure uses recursion to calculate the factorial of its original argument.

VB

```
Function factorial(ByVal n As Integer) As Integer
    If n <= 1 Then
        Return 1
    Else
        Return factorial(n - 1) * n
    End If
End Function
```

Considerations with Recursive Procedures

Limiting Conditions. You must design a recursive procedure to test for at least one condition that can terminate the recursion, and you must also handle the case where no such condition is satisfied within a reasonable number of recursive calls. Without at least one condition that can be met without fail, your procedure runs a high risk of executing in an infinite loop.

Memory Usage. Your application has a limited amount of space for local variables. Each time a procedure calls itself, it uses more of that space for additional copies of its local variables. If this process continues indefinitely, it eventually causes a StackOverflowException error.

Efficiency. You can almost always substitute a loop for recursion. A loop does not have the overhead of passing arguments, initializing additional storage, and returning values. Your performance can be much better without recursive calls.

Mutual Recursion. You might observe very poor performance, or even an infinite loop, if two procedures call each other. Such a design presents the same problems as a single recursive procedure, but can be harder to detect and debug.

Calling with Parentheses. When a **Function** procedure calls itself recursively, you must follow the procedure name with parentheses, even if there is no argument list. Otherwise, the function name is taken as representing the return value of the function.

Testing. If you write a recursive procedure, you should test it very carefully to make sure it always meets some limiting condition. You should also ensure that you cannot run out of memory due to having too many recursive calls.

See Also

StackOverflowException

Procedures in Visual Basic
Sub Procedures (Visual Basic)
Function Procedures (Visual Basic)
Property Procedures (Visual Basic)
Operator Procedures (Visual Basic)
Procedure Parameters and Arguments (Visual Basic)
Procedure Overloading (Visual Basic)
Troubleshooting Procedures (Visual Basic)
Loop Structures (Visual Basic)
Troubleshooting Exceptions: System.StackOverflowException

Partial Methods (Visual Basic)

Visual Studio 2015

Partial methods enable developers to insert custom logic into code. Typically, the code is part of a designer-generated class. Partial methods are defined in a partial class that is created by a code generator, and they are commonly used to provide notification that something has been changed. They enable the developer to specify custom behavior in response to the change.

The designer of the code generator defines only the method signature and one or more calls to the method. Developers can then provide implementations for the method if they want to customize the behavior of the generated code. When no implementation is provided, calls to the method are removed by the compiler, resulting in no additional performance overhead.

Declaration

The generated code marks the definition of a partial method by placing the keyword **Partial** at the start of the signature line.

VB

```
Partial Private Sub QuantityChanged()
End Sub
```

The definition must meet the following conditions:

- The method must be a **Sub**, not a **Function**.
- The body of the method must be left empty.
- The access modifier must be **Private**.

Implementation

The implementation consists primarily of filling in the body of the partial method. The implementation is typically in a separate partial class from the definition, and is written by a developer who wants to extend the generated code.

VB

```
Private Sub QuantityChanged()
'    Code for executing the desired action.
End Sub
```

The previous example duplicates the signature in the declaration exactly, but variations are possible. In particular, other

modifiers can be added, such as **Overloads** or **Overrides**. Only one **Overrides** modifier is permitted. For more information about method modifiers, see Sub Statement (Visual Basic).

Use

You call a partial method as you would call any other **Sub** procedure. If the method has been implemented, the arguments are evaluated and the body of the method is executed. However, remember that implementing a partial method is optional. If the method is not implemented, a call to it has no effect, and expressions passed as arguments to the method are not evaluated.

Example

In a file named Product.Designer.vb, define a `Product` class that has a `Quantity` property.

VB

```
Partial Class Product

    Private _Quantity As Integer

    Property Quantity() As Integer
        Get
            Return _Quantity
        End Get
        Set(ByVal value As Integer)
            _Quantity = value
            QuantityChanged()
        End Set
    End Property

    ' Provide a signature for the partial method.
    Partial Private Sub QuantityChanged()
    End Sub
End Class
```

In a file named Product.vb, provide an implementation for `QuantityChanged`.

VB

```
Partial Class Product

    Private Sub QuantityChanged()
        MsgBox("Quantity was changed to " & Me.Quantity)
    End Sub

End Class
```

Finally, in the Main method of a project, declare a `Product` instance and provide an initial value for its `Quantity`

property.

VB

```
Module Module1

    Sub Main()
        Dim product1 As New Product With {.Quantity = 100}
    End Sub

End Module
```

A message box should appear that displays this message:

```
Quantity was changed to 100
```

See Also

Sub Statement (Visual Basic)
Sub Procedures (Visual Basic)
Optional Parameters (Visual Basic)
Partial (Visual Basic)
Code Generation in LINQ to SQL
Adding Business Logic By Using Partial Methods

Procedure Parameters and Arguments (Visual Basic)

Visual Studio 2015

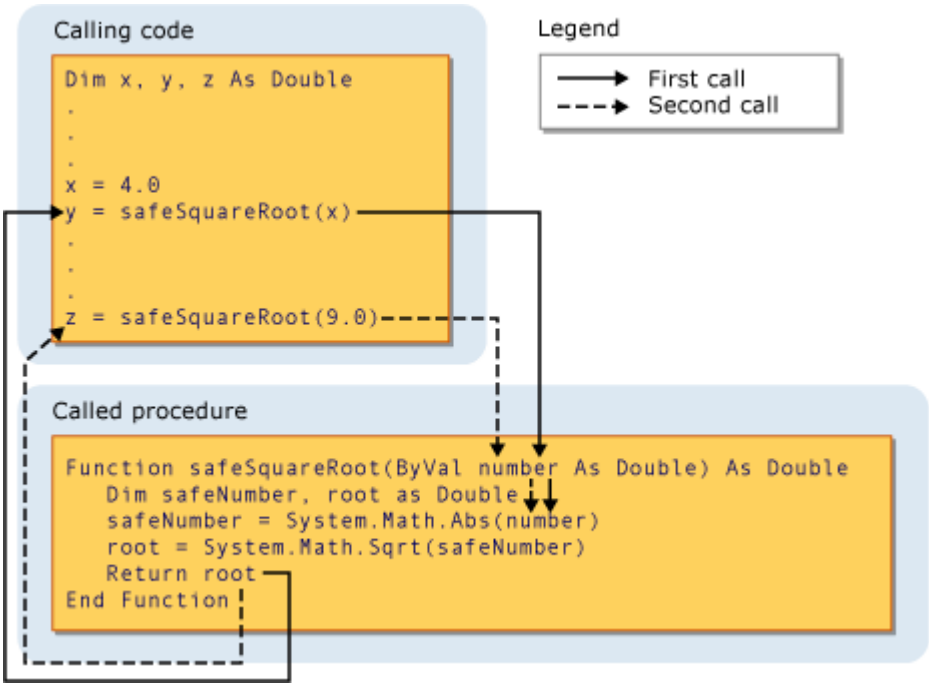
In most cases, a procedure needs some information about the circumstances in which it has been called. A procedure that performs repeated or shared tasks uses different information for each call. This information consists of variables, constants, and expressions that you pass to the procedure when you call it.

A *parameter* represents a value that the procedure expects you to supply when you call it. The procedure's declaration defines its parameters.

You can define a procedure with no parameters, one parameter, or more than one. The part of the procedure definition that specifies the parameters is called the *parameter list*.

An *argument* represents the value you supply to a procedure parameter when you call the procedure. The calling code supplies the arguments when it calls the procedure. The part of the procedure call that specifies the arguments is called the *argument list*.

The following illustration shows code calling the procedure `safeSquareRoot` from two different places. The first call passes the value of the variable `x` (4.0) to the parameter `number`, and the return value in `root` (2.0) is assigned to the variable `y`. The second call passes the literal value 9.0 to `number`, and assigns the return value (3.0) to variable `z`.



Passing an argument to a parameter

For more information, see Differences Between Parameters and Arguments (Visual Basic).

Parameter Data Type

You define a data type for a parameter by using the **As** clause in its declaration. For example, the following function accepts a string and an integer.

VB

```
Function appointment(ByVal day As String, ByVal hour As Integer) As String
    ' Insert code to return any appointment for the given day and time.
    Return "appointment"
End Function
```

If the type checking switch (Option Strict Statement) is **Off,** the **As** clause is optional, except that if any one parameter uses it, all parameters must use it. If type checking is **On**, the **As** clause is required for all procedure parameters.

If the calling code expects to supply an argument with a data type different from that of its corresponding parameter, such as **Byte** to a **String** parameter, it must do one of the following:

- Supply only arguments with data types that widen to the parameter data type;
- Set **Option Strict Off** to allow implicit narrowing conversions; or
- Use a conversion keyword to explicitly convert the data type.

Type Parameters

A *generic procedure* also defines one or more *type parameters* in addition to its normal parameters. A generic procedure allows the calling code to pass different data types each time it calls the procedure, so it can tailor the data types to the requirements of each individual call. See Generic Procedures in Visual Basic.

See Also

Procedures in Visual Basic
Sub Procedures (Visual Basic)
Function Procedures (Visual Basic)
Property Procedures (Visual Basic)
Operator Procedures (Visual Basic)
How to: Define a Parameter for a Procedure (Visual Basic)
How to: Pass Arguments to a Procedure (Visual Basic)
Passing Arguments by Value and by Reference (Visual Basic)
Procedure Overloading (Visual Basic)
Type Conversions in Visual Basic

Differences Between Parameters and Arguments (Visual Basic)

Visual Studio 2015

In most cases, a procedure must have some information about the circumstances in which it has been called. A procedure that performs repeated or shared tasks uses different information for each call. This information consists of variables, constants, and expressions that you pass to the procedure when you call it.

To communicate this information to the procedure, the procedure defines a *parameter*, and the calling code passes an *argument* to that parameter. You can think of the parameter as a parking space and the argument as an automobile. Just as different automobiles can park in a parking space at different times, the calling code can pass a different argument to the same parameter every time that it calls the procedure.

Parameters

A *parameter* represents a value that the procedure expects you to pass when you call it. The procedure's declaration defines its parameters.

When you define a **Function** or **Sub** procedure, you specify a *parameter list* in parentheses immediately following the procedure name. For each parameter, you specify a name, a data type, and a passing mechanism (ByVal (Visual Basic) or ByRef (Visual Basic)). You can also indicate that a parameter is optional. This means that the calling code does not have to pass a value for it.

The name of each parameter serves as a *local variable* in the procedure. You use the parameter name the same way you use any other variable.

Arguments

An *argument* represents the value that you pass to a procedure parameter when you call the procedure. The calling code supplies the arguments when it calls the procedure.

When you call a **Function** or **Sub** procedure, you include an *argument list* in parentheses immediately following the procedure name. Each argument corresponds to the parameter in the same position in the list.

In contrast to parameter definition, arguments do not have names. Each argument is an expression, which can contain zero or more variables, constants, and literals. The data type of the evaluated expression should typically match the data type defined for the corresponding parameter, and in any case it must be convertible to the parameter type.

See Also

Procedures in Visual Basic
Sub Procedures (Visual Basic)

Function Procedures (Visual Basic)
Property Procedures (Visual Basic)
Operator Procedures (Visual Basic)
How to: Define a Parameter for a Procedure (Visual Basic)
How to: Pass Arguments to a Procedure (Visual Basic)
Passing Arguments by Value and by Reference (Visual Basic)
Recursive Procedures (Visual Basic)
Procedure Overloading (Visual Basic)

How to: Define a Parameter for a Procedure (Visual Basic)

Visual Studio 2015

A *parameter* allows the calling code to pass a value to the procedure when it calls it. You declare each parameter for a procedure the same way you declare a variable, specifying its name and data type. You also specify the passing mechanism, and whether the parameter is optional.

For more information, see Procedure Parameters and Arguments (Visual Basic).

To define a procedure parameter

1. In the procedure declaration, add the parameter name to the procedure's parameter list, separating it from other parameters by commas.

2. Decide the data type of the parameter.

3. Follow the parameter name with an **As** clause to specify the data type.

4. Decide the passing mechanism you want for the parameter. Normally you pass a parameter by value, unless you want the procedure to be able to change its value in the calling code.

5. Precede the parameter name with ByVal (Visual Basic) or ByRef (Visual Basic) to specify the passing mechanism. For more information, see Differences Between Passing an Argument By Value and By Reference (Visual Basic).

6. If the parameter is optional, precede the passing mechanism with Optional (Visual Basic) and follow the parameter data type with an equal sign (=) and a default value.

   The following example defines the outline of a **Sub** procedure with three parameters. The first two are required and the third is optional. The parameter declarations are separated in the parameter list by commas.

   VB

   ```
   Sub updateCustomer(ByRef c As customer, ByVal region As String,
     Optional ByVal level As Integer = 0)
     ' Insert code to update a customer object.
   End Sub
   ```

   The first parameter accepts a `customer` object, and `updateCustomer` can directly update the variable passed to `c` because the argument is passed ByRef (Visual Basic). The procedure cannot change the values of the last two arguments because they are passed ByVal (Visual Basic).

   If the calling code does not supply a value for the `level` parameter, Visual Basic sets it to the default value of 0.

   If the type checking switch (Option Strict Statement) is **Off**, the **As** clause is optional when you define a parameter. However, if any one parameter uses an **As** clause, all of them must use it. If the type checking switch is **On**, the **As**

clause is required for every parameter definition.

Specifying data types for all your programming elements is known as *strong typing*. When you set **Option Strict On**, Visual Basic enforces strong typing. This is strongly recommended, for the following reasons:

- It enables IntelliSense support for your variables and parameters. This allows you to see their properties and other members as you type in your code.
- It allows the compiler to perform type checking. This helps catch 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. 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. Your compiled code might have to convert back and forth between **Object** and other data types, which reduces performance.

See Also

Procedures in Visual Basic
Sub Procedures (Visual Basic)
Function Procedures (Visual Basic)
How to: Pass Arguments to a Procedure (Visual Basic)
Passing Arguments by Value and by Reference (Visual Basic)
Recursive Procedures (Visual Basic)
Procedure Overloading (Visual Basic)
Objects and Classes in Visual Basic
Object-Oriented Programming (C# and Visual Basic)

How to: Pass Arguments to a Procedure (Visual Basic)

Visual Studio 2015

When you call a procedure, you follow the procedure name with an argument list in parentheses. You supply an argument corresponding to every required parameter the procedure defines, and you can optionally supply arguments to the **Optional** parameters. If you do not supply an **Optional** parameter in the call, you must include a comma to mark its place in the argument list if you are supplying any subsequent arguments.

If you intend to pass an argument of a data type different from that of its corresponding parameter, such as **Byte** to **String**, you can set the type-checking switch (Option Strict Statement) to **Off**. If **Option Strict** is **On**, you must use either widening conversions or explicit conversion keywords. For more information, see Widening and Narrowing Conversions (Visual Basic) and Type Conversion Functions (Visual Basic).

For more information, see Procedure Parameters and Arguments (Visual Basic).

To pass one or more arguments to a procedure

1. In the calling statement, follow the procedure name with parentheses.

2. Inside the parentheses, put an argument list. Include an argument for each required parameter the procedure defines, and separate the arguments with commas.

3. Make sure each argument is a valid expression that evaluates to a data type convertible to the type the procedure defines for the corresponding parameter.

4. If a parameter is defined as Optional (Visual Basic), you can either include it in the argument list or omit it. If you omit it, the procedure uses the default value defined for that parameter.

5. If you omit an argument for an **Optional** parameter and there is another parameter after it in the parameter list, you can mark the place of the omitted argument by an extra comma in the argument list.

   The following example calls the Visual Basic MsgBox function.

   VB

   ```
   Dim mbResult As MsgBoxResult
   Dim displayString As String = "Show this string to the user"
   mbResult = MsgBox(displayString, , "Put this in the title bar")
   ```

   The preceding example supplies the required first argument, which is the message string to be displayed. It omits an argument for the optional second parameter, which specifies the buttons to be displayed on the message box. Because the call does not supply a value, **MsgBox** uses the default value, **MsgBoxStyle.OKOnly**, which displays only an **OK** button.

   The second comma in the argument list marks the place of the omitted second argument, and the last string is passed

to the optional third parameter of **MsgBox**, which is the text to be displayed in the title bar.

See Also

Sub Procedures (Visual Basic)
Function Procedures (Visual Basic)
Property Procedures (Visual Basic)
Operator Procedures (Visual Basic)
How to: Define a Parameter for a Procedure (Visual Basic)
Passing Arguments by Value and by Reference (Visual Basic)
Recursive Procedures (Visual Basic)
Procedure Overloading (Visual Basic)
Objects and Classes in Visual Basic
Object-Oriented Programming (C# and Visual Basic)

Passing Arguments by Value and by Reference (Visual Basic)

Visual Studio 2015

In Visual Basic, you can pass an argument to a procedure *by value* or *by reference*. This is known as the *passing mechanism*, and it determines whether the procedure can modify the programming element underlying the argument in the calling code. The procedure declaration determines the passing mechanism for each parameter by specifying the ByVal (Visual Basic) or ByRef (Visual Basic) keyword.

Distinctions

When passing an argument to a procedure, be aware of several different distinctions that interact with each other:

- Whether the underlying programming element is modifiable or nonmodifiable
- Whether the argument itself is modifiable or nonmodifiable
- Whether the argument is being passed by value or by reference
- Whether the argument data type is a value type or a reference type

For more information, see Differences Between Modifiable and Nonmodifiable Arguments (Visual Basic) and Differences Between Passing an Argument By Value and By Reference (Visual Basic).

Choice of Passing Mechanism

You should choose the passing mechanism carefully for each argument.

- **Protection**. In choosing between the two passing mechanisms, the most important criterion is the exposure of calling variables to change. The advantage of passing an argument **ByRef** is that the procedure can return a value to the calling code through that argument. The advantage of passing an argument **ByVal** is that it protects a variable from being changed by the procedure.
- **Performance**. Although the passing mechanism can affect the performance of your code, the difference is usually insignificant. One exception to this is a value type passed **ByVal**. In this case, Visual Basic copies the entire data contents of the argument. Therefore, for a large value type such as a structure, it can be more efficient to pass it **ByRef**.

  For reference types, only the pointer to the data is copied (four bytes on 32-bit platforms, eight bytes on 64-bit platforms). Therefore, you can pass arguments of type **String** or **Object** by value without harming performance.

Determination of the Passing Mechanism

The procedure declaration specifies the passing mechanism for each parameter. The calling code can't override a **ByVal** mechanism.

If a parameter is declared with **ByRef**, the calling code can force the mechanism to **ByVal** by enclosing the argument name in parentheses in the call. For more information, see How to: Force an Argument to Be Passed by Value (Visual Basic).

The default in Visual Basic is to pass arguments by value.

When to Pass an Argument by Value

- If the calling code element underlying the argument is a nonmodifiable element, declare the corresponding parameter ByVal (Visual Basic). No code can change the value of a nonmodifiable element.
- If the underlying element is modifiable, but you do not want the procedure to be able to change its value, declare the parameter **ByVal**. Only the calling code can change the value of a modifiable element passed by value.

When to Pass an Argument by Reference

- If the procedure has a genuine need to change the underlying element in the calling code, declare the corresponding parameter ByRef (Visual Basic).
- If the correct execution of the code depends on the procedure changing the underlying element in the calling code, declare the parameter **ByRef**. If you pass it by value, or if the calling code overrides the **ByRef** passing mechanism by enclosing the argument in parentheses, the procedure call might produce unexpected results.

Example

Description

The following example illustrates when to pass arguments by value and when to pass them by reference. Procedure `Calculate` has both a **ByVal** and a **ByRef** parameter. Given an interest rate, `rate`, and a sum of money, `debt`, the task of the procedure is to calculate a new value for `debt` that is the result of applying the interest rate to the original value of `debt`. Because `debt` is a **ByRef** parameter, the new total is reflected in the value of the argument in the calling code that corresponds to `debt`. Parameter `rate` is a **ByVal** parameter because `Calculate` should not change its value.

Code

VB

```
Module Module1

    Sub Main()
        ' Two interest rates are declared, one a constant and one a
        ' variable.
        Const highRate As Double = 12.5
        Dim lowRate = highRate * 0.6

        Dim initialDebt = 4999.99
        ' Make a copy of the original value of the debt.
        Dim debtWithInterest = initialDebt

        ' Calculate the total debt with the high interest rate applied.
        ' Argument highRate is a constant, which is appropriate for a
        ' ByVal parameter. Argument debtWithInterest must be a variable
        ' because the procedure will change its value to the calculated
        ' total with interest applied.
        Calculate(highRate, debtWithInterest)
        ' Format the result to represent currency, and display it.
        Dim debtString = Format(debtWithInterest, "C")
        Console.WriteLine("What I owe with high interest: " & debtString)

        ' Repeat the process with lowRate. Argument lowRate is not a
        ' constant, but the ByVal parameter protects it from accidental
        ' or intentional change by the procedure.

        ' Set debtWithInterest back to the original value.
        debtWithInterest = initialDebt
        Calculate(lowRate, debtWithInterest)
        debtString = Format(debtWithInterest, "C")
        Console.WriteLine("What I owe with low interest:  " & debtString)
    End Sub

    ' Parameter rate is a ByVal parameter because the procedure should
    ' not change the value of the corresponding argument in the
    ' calling code.

    ' The calculated value of the debt parameter, however, should be
    ' reflected in the value of the corresponding argument in the
    ' calling code. Therefore, it must be declared ByRef.
    Sub Calculate(ByVal rate As Double, ByRef debt As Double)
        debt = debt + (debt * rate / 100)
    End Sub

End Module
```

See Also

Procedures in Visual Basic

Procedure Parameters and Arguments (Visual Basic)
How to: Pass Arguments to a Procedure (Visual Basic)
How to: Change the Value of a Procedure Argument (Visual Basic)
How to: Protect a Procedure Argument Against Value Changes (Visual Basic)
How to: Force an Argument to Be Passed by Value (Visual Basic)
Passing Arguments by Position and by Name (Visual Basic)
Value Types and Reference Types

Differences Between Modifiable and Nonmodifiable Arguments (Visual Basic)

Visual Studio 2015

When you call a procedure, you typically pass one or more arguments to it. Each argument corresponds to an underlying programming element. Both the underlying elements and the arguments themselves can be either modifiable or nonmodifiable.

Modifiable and Nonmodifiable Elements

A programming element can be either a *modifiable element*, which can have its value changed, or a *nonmodifiable element*, which has a fixed value once it has been created.

The following table lists modifiable and nonmodifiable programming elements.

Modifiable elements	Nonmodifiable elements
Local variables (declared inside procedures), including object variables, except for read-only	Read-only variables, fields, and properties
Fields (member variables of modules, classes, and structures), except for read-only	Constants and literals
Properties, except for read-only	Enumeration members
Array elements	Expressions (even if their elements are modifiable)

Modifiable and Nonmodifiable Arguments

A *modifiable argument* is one with a modifiable underlying element. The calling code can store a new value at any time, and if you pass the argument ByRef (Visual Basic), the code in the procedure can also modify the underlying element in the calling code.

A *nonmodifiable argument* either has a nonmodifiable underlying element or is passed ByVal (Visual Basic). The procedure cannot modify the underlying element in the calling code, even if it is a modifiable element. If it is a nonmodifiable element, the calling code itself cannot modify it.

The called procedure might modify its local copy of a nonmodifiable argument, but that modification does not affect the underlying element in the calling code.

See Also

Procedures in Visual Basic
Procedure Parameters and Arguments (Visual Basic)
How to: Pass Arguments to a Procedure (Visual Basic)
Passing Arguments by Value and by Reference (Visual Basic)
Differences Between Passing an Argument By Value and By Reference (Visual Basic)
How to: Change the Value of a Procedure Argument (Visual Basic)
How to: Protect a Procedure Argument Against Value Changes (Visual Basic)
How to: Force an Argument to Be Passed by Value (Visual Basic)
Passing Arguments by Position and by Name (Visual Basic)
Value Types and Reference Types

Differences Between Passing an Argument By Value and By Reference (Visual Basic)

Visual Studio 2015

When you pass one or more arguments to a procedure, each argument corresponds to an underlying programming element in the calling code. You can pass either the value of this underlying element, or a reference to it. This is known as the *passing mechanism*.

Passing by Value

You pass an argument *by value* by specifying the ByVal (Visual Basic) keyword for the corresponding parameter in the procedure definition. When you use this passing mechanism, Visual Basic copies the value of the underlying programming element into a local variable in the procedure. The procedure code does not have any access to the underlying element in the calling code.

Passing by Reference

You pass an argument *by reference* by specifying the ByRef (Visual Basic) keyword for the corresponding parameter in the procedure definition. When you use this passing mechanism, Visual Basic gives the procedure a direct reference to the underlying programming element in the calling code.

Passing Mechanism and Element Type

The choice of passing mechanism is not the same as the classification of the underlying element type. Passing by value or by reference refers to what Visual Basic supplies to the procedure code. A value type or reference type refers to how a programming element is stored in memory.

However, the passing mechanism and element type are interrelated. The value of a reference type is a pointer to the data elsewhere in memory. This means that when you pass a reference type by value, the procedure code has a pointer to the underlying element's data, even though it cannot access the underlying element itself. For example, if the element is an array variable, the procedure code does not have access to the variable itself, but it can access the array members.

Ability to Modify

When you pass a nonmodifiable element as an argument, the procedure can never modify it in the calling code, whether it is passed **ByVal** or **ByRef**.

For a modifiable element, the following table summarizes the interaction between the element type and the passing mechanism.

Element type	Passed ByVal	Passed ByRef
Value type (contains only a value)	The procedure cannot change the variable or any of its members.	The procedure can change the variable and its members.
Reference type (contains a pointer to a class or structure instance)	The procedure cannot change the variable but can change members of the instance to which it points.	The procedure can change the variable and members of the instance to which it points.

See Also

Procedures in Visual Basic
Procedure Parameters and Arguments (Visual Basic)
How to: Pass Arguments to a Procedure (Visual Basic)
Passing Arguments by Value and by Reference (Visual Basic)
Differences Between Modifiable and Nonmodifiable Arguments (Visual Basic)
How to: Change the Value of a Procedure Argument (Visual Basic)
How to: Protect a Procedure Argument Against Value Changes (Visual Basic)
How to: Force an Argument to Be Passed by Value (Visual Basic)
Passing Arguments by Position and by Name (Visual Basic)
Value Types and Reference Types

How to: Change the Value of a Procedure Argument (Visual Basic)

Visual Studio 2015

When you call a procedure, each argument you supply corresponds to one of the parameters defined in the procedure. In some cases, the procedure code can change the value underlying an argument in the calling code. In other cases, the procedure can change only its local copy of an argument.

When you call the procedure, Visual Basic makes a local copy of every argument that is passed ByVal (Visual Basic). For each argument passed ByRef (Visual Basic), Visual Basic gives the procedure code a direct reference to the programming element underlying the argument in the calling code.

If the underlying element in the calling code is a modifiable element and the argument is passed **ByRef**, the procedure code can use the direct reference to change the element's value in the calling code.

Changing the Underlying Value

To change the underlying value of a procedure argument in the calling code

1. In the procedure declaration, specify ByRef (Visual Basic) for the parameter corresponding to the argument.
2. In the calling code, pass a modifiable programming element as the argument.
3. In the calling code, do not enclose the argument in parentheses in the argument list.
4. In the procedure code, use the parameter name to assign a value to the underlying element in the calling code.

See the example further down for a demonstration.

Changing Local Copies

If the underlying element in the calling code is a nonmodifiable element, or if the argument is passed **ByVal**, the procedure cannot change its value in the calling code. However, the procedure can change its local copy of such an argument.

To change the copy of a procedure argument in the procedure code

1. In the procedure declaration, specify ByVal (Visual Basic) for the parameter corresponding to the argument.

   -or-

   In the calling code, enclose the argument in parentheses in the argument list. This forces Visual Basic to pass the argument by value, even if the corresponding parameter specifies **ByRef**.

2. In the procedure code, use the parameter name to assign a value to the local copy of the argument. The underlying value in the calling code is not changed.

Example

The following example shows two procedures that take an array variable and operate on its elements. The `increase` procedure simply adds one to each element. The `replace` procedure assigns a new array to the parameter `a()` and then adds one to each element.

VB

```
Public Sub increase(ByVal a() As Long)
    For j As Integer = 0 To UBound(a)
        a(j) = a(j) + 1
    Next j
End Sub
```

VB

```
Public Sub replace(ByRef a() As Long)
    Dim k() As Long = {100, 200, 300}
    a = k
    For j As Integer = 0 To UBound(a)
        a(j) = a(j) + 1
    Next j
End Sub
```

VB

```
Dim n() As Long = {10, 20, 30, 40}
Call increase(n)
MsgBox("After increase(n): " & CStr(n(0)) & ", " &
    CStr(n(1)) & ", " & CStr(n(2)) & ", " & CStr(n(3)))
Call replace(n)
MsgBox("After replace(n): " & CStr(n(0)) & ", " &
    CStr(n(1)) & ", " & CStr(n(2)) & ", " & CStr(n(3)))
```

The first **MsgBox** call displays "After increase(n): 11, 21, 31, 41". Because the array `n` is a reference type, `replace` can change its members, even though the passing mechanism is **ByVal**.

The second **MsgBox** call displays "After replace(n): 101, 201, 301". Because `n` is passed **ByRef**, `replace` can modify the variable `n` in the calling code and assign a new array to it. Because `n` is a reference type, `replace` can also change its members.

You can prevent the procedure from modifying the variable itself in the calling code. See How to: Protect a Procedure Argument Against Value Changes (Visual Basic).

Compiling the Code

When you pass a variable by reference, you must use the **ByRef** keyword to specify this mechanism.

The default in Visual Basic is to pass arguments by value. However, it is good programming practice to include either the ByVal (Visual Basic) or ByRef (Visual Basic) keyword with every declared parameter. This makes your code easier to read.

.NET Framework Security

There is always a potential risk in allowing a procedure to change the value underlying an argument in the calling code. Make sure you expect this value to be changed, and be prepared to check it for validity before using it.

See Also

- Procedures in Visual Basic
- Procedure Parameters and Arguments (Visual Basic)
- How to: Pass Arguments to a Procedure (Visual Basic)
- Passing Arguments by Value and by Reference (Visual Basic)
- Differences Between Modifiable and Nonmodifiable Arguments (Visual Basic)
- Differences Between Passing an Argument By Value and By Reference (Visual Basic)
- How to: Protect a Procedure Argument Against Value Changes (Visual Basic)
- How to: Force an Argument to Be Passed by Value (Visual Basic)
- Passing Arguments by Position and by Name (Visual Basic)
- Value Types and Reference Types

How to: Protect a Procedure Argument Against Value Changes (Visual Basic)

Visual Studio 2015

If a procedure declares a parameter as ByRef (Visual Basic), Visual Basic gives the procedure code a direct reference to the programming element underlying the argument in the calling code. This permits the procedure to change the value underlying the argument in the calling code. In some cases the calling code might want to protect against such a change.

You can always protect an argument from change by declaring the corresponding parameter ByVal (Visual Basic) in the procedure. If you want to be able to change a given argument in some cases but not others, you can declare it **ByRef** and let the calling code determine the passing mechanism in each call. It does this by enclosing the corresponding argument in parentheses to pass it by value, or not enclosing it in parentheses to pass it by reference. For more information, see How to: Force an Argument to Be Passed by Value (Visual Basic).

Example

The following example shows two procedures that take an array variable and operate on its elements. The `increase` procedure simply adds one to each element. The `replace` procedure assigns a new array to the parameter `a()` and then adds one to each element. However, the reassignment does not affect the underlying array variable in the calling code.

VB

```
Public Sub increase(ByVal a() As Long)
    For j As Integer = 0 To UBound(a)
        a(j) = a(j) + 1
    Next j
End Sub
```

VB

```
Public Sub replace(ByVal a() As Long)
    Dim k() As Long = {100, 200, 300}
    a = k
    For j As Integer = 0 To UBound(a)
        a(j) = a(j) + 1
    Next j
End Sub
```

VB

```
Dim n() As Long = {10, 20, 30, 40}
Call increase(n)
MsgBox("After increase(n): " & CStr(n(0)) & ", " &
    CStr(n(1)) & ", " & CStr(n(2)) & ", " & CStr(n(3)))
Call replace(n)
```

```
MsgBox("After replace(n): " & CStr(n(0)) & ", " &
    CStr(n(1)) & ", " & CStr(n(2)) & ", " & CStr(n(3)))
```

The first **MsgBox** call displays "After increase(n): 11, 21, 31, 41". Because the array `n` is a reference type, `replace` can change its members, even though the passing mechanism is **ByVal**.

The second **MsgBox** call displays "After replace(n): 11, 21, 31, 41". Because `n` is passed **ByVal**, `replace` cannot modify the variable `n` in the calling code by assigning a new array to it. When `replace` creates the new array instance `k` and assigns it to the local variable `a`, it loses the reference to `n` passed in by the calling code. When it changes the members of `a`, only the local array `k` is affected. Therefore, `replace` does not increment the values of array `n` in the calling code.

Compiling the Code

The default in Visual Basic is to pass arguments by value. However, it is good programming practice to include either the ByVal (Visual Basic) or ByRef (Visual Basic) keyword with every declared parameter. This makes your code easier to read.

See Also

- Procedures in Visual Basic
- Procedure Parameters and Arguments (Visual Basic)
- How to: Pass Arguments to a Procedure (Visual Basic)
- Passing Arguments by Value and by Reference (Visual Basic)
- Differences Between Modifiable and Nonmodifiable Arguments (Visual Basic)
- Differences Between Passing an Argument By Value and By Reference (Visual Basic)
- How to: Change the Value of a Procedure Argument (Visual Basic)
- How to: Force an Argument to Be Passed by Value (Visual Basic)
- Passing Arguments by Position and by Name (Visual Basic)
- Value Types and Reference Types

How to: Force an Argument to Be Passed by Value (Visual Basic)

Visual Studio 2015

The procedure declaration determines the passing mechanism. If a parameter is declared ByRef (Visual Basic), Visual Basic expects to pass the corresponding argument by reference. This allows the procedure to change the value of the programming element underlying the argument in the calling code. If you wish to protect the underlying element against such change, you can override the **ByRef** passing mechanism in the procedure call by enclosing the argument name in parentheses. These parentheses are in addition to the parentheses enclosing the argument list in the call.

The calling code cannot override a ByVal (Visual Basic) mechanism.

To force an argument to be passed by value

- If the corresponding parameter is declared **ByVal** in the procedure, you do not need to take any additional steps. Visual Basic already expects to pass the argument by value.
- If the corresponding parameter is declared **ByRef** in the procedure, enclose the argument in parentheses in the procedure call.

Example

The following example overrides a **ByRef** parameter declaration. In the call that forces **ByVal**, note the two levels of parentheses.

VB

```
Sub setNewString(ByRef inString As String)
    inString = "This is a new value for the inString argument."
    MsgBox(inString)
End Sub
```

VB

```
Dim str As String = "Cannot be replaced if passed ByVal"

' The following call passes str ByVal even though it is declared ByRef.
Call setNewString((str))
' The parentheses around str protect it from change.
MsgBox(str)

' The following call allows str to be passed ByRef as declared.
Call setNewString(str)
' Variable str is not protected from change.
MsgBox(str)
```

When `str` is enclosed in extra parentheses within the argument list, the `setNewString` procedure cannot change its value in the calling code, and **MsgBox** displays "Cannot be replaced if passed ByVal". When `str` is not enclosed in extra parentheses, the procedure can change it, and **MsgBox** displays "This is a new value for the inString argument."

Compiling the Code

When you pass a variable by reference, you must use the **ByRef** keyword to specify this mechanism.

The default in Visual Basic is to pass arguments by value. However, it is good programming practice to include either the ByVal (Visual Basic) or ByRef (Visual Basic) keyword with every declared parameter. This makes your code easier to read.

Robust Programming

If a procedure declares a parameter ByRef (Visual Basic), the correct execution of the code might depend on being able to change the underlying element in the calling code. If the calling code overrides this calling mechanism by enclosing the argument in parentheses, or if it passes a nonmodifiable argument, the procedure cannot change the underlying element. This might produce unexpected results in the calling code.

.NET Framework Security

There is always a potential risk in allowing a procedure to change the value underlying an argument in the calling code. Make sure you expect this value to be changed, and be prepared to check it for validity before using it.

See Also

- Procedures in Visual Basic
- Procedure Parameters and Arguments (Visual Basic)
- How to: Pass Arguments to a Procedure (Visual Basic)
- Passing Arguments by Value and by Reference (Visual Basic)
- Differences Between Modifiable and Nonmodifiable Arguments (Visual Basic)
- Differences Between Passing an Argument By Value and By Reference (Visual Basic)
- How to: Change the Value of a Procedure Argument (Visual Basic)
- How to: Protect a Procedure Argument Against Value Changes (Visual Basic)
- Passing Arguments by Position and by Name (Visual Basic)
- Value Types and Reference Types

Passing Arguments by Position and by Name (Visual Basic)

Visual Studio 2015

When you call a **Sub** or **Function** procedure, you can pass arguments *by position* — in the order in which they appear in the procedure's definition — or you can pass them *by name*, without regard to position.

When you pass an argument by name, you specify the argument's declared name followed by a colon and an equal sign (**:=**), followed by the argument value. You can supply named arguments in any order.

For example, the following **Sub** procedure takes three arguments:

VB

```
Sub studentInfo(ByVal name As String,
       Optional ByVal age As Short = 0,
       Optional ByVal birth As Date = #1/1/2000#)

  Debug.WriteLine("Name = " & name &
                  "; age = " & CStr(age) &
                  "; birth date = " & CStr(birth))
End Sub
```

When you call this procedure, you can supply the arguments by position, by name, or by using a mixture of both.

Passing Arguments by Position

You can call the procedure `studentInfo` with its arguments passed by position and delimited by commas, as shown in the following example:

VB

```
Call studentInfo("Mary", 19, #9/21/1981#)
```

If you omit an optional argument in a positional argument list, you must hold its place with a comma. The following example calls `studentInfo` without the `age` argument:

VB

```
Call studentInfo("Mary", , #9/21/1981#)
```

Passing Arguments by Name

Alternatively, you can call `studentInfo` with the arguments passed by name, also delimited by commas, as shown in the following example:

VB

```
Call studentInfo(age:=19, birth:=#9/21/1981#, name:="Mary")
```

Mixing Arguments by Position and by Name

You can supply arguments both by position and by name in a single procedure call, as shown in the following example:

VB

```
Call studentInfo("Mary", birth:=#9/21/1981#)
```

In the preceding example, no extra comma is necessary to hold the place of the omitted `age` argument, since `birth` is passed by name.

When you supply arguments by a mixture of position and name, the positional arguments must all come first. Once you supply an argument by name, the remaining arguments must all be by name.

Supplying Optional Arguments by Name

Passing arguments by name is especially useful when you call a procedure that has more than one optional argument. If you supply arguments by name, you do not have to use consecutive commas to denote missing positional arguments. Passing arguments by name also makes it easier to keep track of which arguments you are passing and which ones you are omitting.

Restrictions on Supplying Arguments by Name

You cannot pass arguments by name to avoid entering required arguments. You can omit only the optional arguments.

You cannot pass a parameter array by name. This is because when you call the procedure, you supply an indefinite number of comma-separated arguments for the parameter array, and the compiler cannot associate more than one argument with a single name.

See Also

Procedures in Visual Basic
Procedure Parameters and Arguments (Visual Basic)

How to: Pass Arguments to a Procedure (Visual Basic)
Passing Arguments by Value and by Reference (Visual Basic)
Optional Parameters (Visual Basic)
Parameter Arrays (Visual Basic)
Optional (Visual Basic)
ParamArray (Visual Basic)

Optional Parameters (Visual Basic)

Visual Studio 2015

You can specify that a procedure parameter is optional and no argument has to be supplied for it when the procedure is called. *Optional parameters* are indicated by the **Optional** keyword in the procedure definition. The following rules apply:

- Every optional parameter in the procedure definition must specify a default value.
- The default value for an optional parameter must be a constant expression.
- Every parameter following an optional parameter in the procedure definition must also be optional.

The following syntax shows a procedure declaration with an optional parameter:

```
Sub sub name(ByVal parameter1 As datatype1, Optional ByVal parameter2 As datatype2 =
defaultvalue)
```

Calling Procedures with Optional Parameters

When you call a procedure with an optional parameter, you can choose whether to supply the argument. If you do not, the procedure uses the default value declared for that parameter.

When you omit one or more optional arguments in the argument list, you use successive commas to mark their positions. The following example call supplies the first and fourth arguments but not the second or third:

```
sub name(argument 1, , , argument 4)
```

The following example makes several calls to the **MsgBox** function. **MsgBox** has one required parameter and two optional parameters.

The first call to **MsgBox** supplies all three arguments in the order that **MsgBox** defines them. The second call supplies only the required argument. The third and fourth calls supply the first and third arguments. The third call does this by position, and the fourth call does it by name.

VB

```
MsgBox("Important message", MsgBoxStyle.Critical, "MsgBox Example")
MsgBox("Just display this message.")
MsgBox("Test message", , "Title bar text")
MsgBox(Title:="Title bar text", Prompt:="Test message")
```

Determining Whether an Optional Argument Is Present

A procedure cannot detect at run time whether a given argument has been omitted or the calling code has explicitly supplied the default value. If you need to make this distinction, you can set an unlikely value as the default. The following procedure defines the optional parameter `office`, and tests for its default value, `QJZ`, to see if it has been omitted in the call:

VB

```
Sub notify(ByVal company As String, Optional ByVal office As String = "QJZ")
    If office = "QJZ" Then
        Debug.WriteLine("office not supplied -- using Headquarters")
        office = "Headquarters"
    End If
    ' Insert code to notify headquarters or specified office.
End Sub
```

If the optional parameter is a reference type such as a **String**, you can use **Nothing** as the default value, provided this is not an expected value for the argument.

Optional Parameters and Overloading

Another way to define a procedure with optional parameters is to use overloading. If you have one optional parameter,

you can define two overloaded versions of the procedure, one accepting the parameter and one without it. This approach becomes more complicated as the number of optional parameters increases. However, its advantage is that you can be absolutely sure whether the calling program supplied each optional argument.

See Also

Procedures in Visual Basic
Procedure Parameters and Arguments (Visual Basic)
Passing Arguments by Value and by Reference (Visual Basic)
Passing Arguments by Position and by Name (Visual Basic)
Parameter Arrays (Visual Basic)
Procedure Overloading (Visual Basic)
Optional (Visual Basic)
ParamArray (Visual Basic)

Parameter Arrays (Visual Basic)

Visual Studio 2015

Usually, you cannot call a procedure with more arguments than the procedure declaration specifies. When you need an indefinite number of arguments, you can declare a *parameter array*, which allows a procedure to accept an array of values for a parameter. You do not have to know the number of elements in the parameter array when you define the procedure. The array size is determined individually by each call to the procedure.

Declaring a ParamArray

You use the ParamArray (Visual Basic) keyword to denote a parameter array in the parameter list. The following rules apply:

- A procedure can define only one parameter array, and it must be the last parameter in the procedure definition.
- The parameter array must be passed by value. It is good programming practice to explicitly include the ByVal (Visual Basic) keyword in the procedure definition.
- The parameter array is automatically optional. Its default value is an empty one-dimensional array of the parameter array's element type.
- All parameters preceding the parameter array must be required. The parameter array must be the only optional parameter.

Calling a ParamArray

When you call a procedure that defines a parameter array, you can supply the argument in any one of the following ways:

- Nothing — that is, you can omit the ParamArray (Visual Basic) argument. In this case, an empty array is passed to the procedure. You can also pass the Nothing (Visual Basic) keyword, with the same effect.
- A list of an arbitrary number of arguments, separated by commas. The data type of each argument must be implicitly convertible to the **ParamArray** element type.
- An array with the same element type as the parameter array's element type.

In all cases, the code within the procedure treats the parameter array as a one-dimensional array with elements of the same data type as the **ParamArray** data type.

Security Note

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, you should test for the size of the array that the calling code passed to it. Take appropriate steps if it is too large for your application. For more information, see Arrays in Visual Basic.

Example

The following example defines and calls the function `calcSum`. The **ParamArray** modifier for the parameter `args` enables the function to accept a variable number of arguments.

VB

```
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
```

The following example defines a procedure with a parameter array, and outputs the values of all the array elements passed to the parameter array.

VB

```
Sub studentScores(ByVal name As String, ByVal ParamArray scores() As String)
    Debug.WriteLine("Scores for " & name & ":" & vbCrLf)
    ' Use UBound to determine largest subscript of the array.
    For i As Integer = 0 To UBound(scores, 1)
        Debug.WriteLine("Score " & i & ": " & scores(i))
    Next i
End Sub
```

VB

```
Call studentScores("Anne", "10", "26", "32", "15", "22", "24", "16")
Call studentScores("Mary", "High", "Low", "Average", "High")
Dim JohnScores() As String = {"35", "Absent", "21", "30"}
Call studentScores("John", JohnScores)
```

See Also

UBound
Procedures in Visual Basic
Procedure Parameters and Arguments (Visual Basic)
Passing Arguments by Value and by Reference (Visual Basic)
Passing Arguments by Position and by Name (Visual Basic)
Optional Parameters (Visual Basic)
Procedure Overloading (Visual Basic)
Arrays in Visual Basic
Optional (Visual Basic)