Operator Procedures (Visual Basic)

Visual Studio 2015

An operator procedure is a series of Visual Basic statements that define the behavior of a standard operator (such as *****, **<>**, or **And**) on a class or structure you have defined. This is also called *operator overloading*.

When to Define Operator Procedures

When you have defined a class or structure, you can declare variables to be of the type of that class or structure. Sometimes such a variable needs to participate in an operation as part of an expression. To do this, it must be an operand of an operator.

Visual Basic defines operators only on its fundamental data types. You can define the behavior of an operator when one or both of the operands are of the type of your class or structure.

For more information, see Operator Statement.

Types of Operator Procedure

An operator procedure can be one of the following types:

- A definition of a unary operator where the argument is of the type of your class or structure.
- A definition of a binary operator where at least one of the arguments is of the type of your class or structure.
- A definition of a conversion operator where the argument is of the type of your class or structure.
- A definition of a conversion operator that returns the type of your class or structure.

Conversion operators are always unary, and you always use **CType** as the operator you are defining.

Declaration Syntax

The syntax for declaring an operator procedure is as follows:

```
Public Shared [Widening | Narrowing] Operator operatorsymbol (operand1 [, operand2]) As datatype

' Statements of the operator procedure.

End Operator
```

You use the **Widening** or **Narrowing** keyword only on a type conversion operator. The operator symbol is always CType

Function (Visual Basic) for a type conversion operator.

You declare two operands to define a binary operator, and you declare one operand to define a unary operator, including a type conversion operator. All operands must be declared **ByVal**.

You declare each operand the same way you declare parameters for Sub Procedures (Visual Basic).

Data Type

Because you are defining an operator on a class or structure you have defined, at least one of the operands must be of the data type of that class or structure. For a type conversion operator, either the operand or the return type must be of the data type of the class or structure.

For more details, see Operator Statement.

Calling Syntax

You invoke an operator procedure implicitly by using the operator symbol in an expression. You supply the operands the same way you do for predefined operators.

The syntax for an implicit call to an operator procedure is as follows:

```
Dim testStruct As structurename

Dim testNewStruct As structurename = testStruct operatorsymbol 10
```

Illustration of Declaration and Call

The following structure stores a signed 128-bit integer value as the constituent high-order and low-order parts. It defines the **+** operator to add two `veryLong` values and generate a resulting `veryLong` value.

VB

```
Public Structure veryLong
    Dim highOrder As Long
    Dim lowOrder As Long
    Public Shared Operator +(ByVal v As veryLong,
                             ByVal w As veryLong) As veryLong
        Dim sum As New veryLong
        sum = v
        Try
            sum.lowOrder += w.lowOrder
        Catch ex As System.OverflowException
            sum.lowOrder -= (Long.MaxValue - w.lowOrder + 1)
            sum.highOrder += 1
        End Try
        sum.highOrder += w.highOrder
        Return sum
    End Operator
End Structure
```

The following example shows a typical call to the **+** operator defined on `veryLong`.

VB

```
Dim v1, v2, v3 As veryLong
v1.highOrder = 1
v1.lowOrder = Long.MaxValue
v2.highOrder = 0
v2.lowOrder = 4
v3 = v1 + v2
```

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

See Also

Procedures in Visual Basic
Sub Procedures (Visual Basic)
Function Procedures (Visual Basic)
Property Procedures (Visual Basic)
Procedure Parameters and Arguments (Visual Basic)
Operator Statement
How to: Define an Operator (Visual Basic)
How to: Define a Conversion Operator (Visual Basic)
How to: Call an Operator Procedure (Visual Basic)
How to: Use a Class that Defines Operators (Visual Basic)

How to: Define an Operator (Visual Basic)

Visual Studio 2015

If you have defined a class or structure, you can define the behavior of a standard operator (such as *****, **<>**, or **And**) when one or both of the operands is of the type of your class or structure.

Define the standard operator as an operator procedure within the class or structure. All operator procedures must be **Public Shared**.

Defining an operator on a class or structure is also called *overloading* the operator.

Example

The following example defines the **+** operator for a structure called `height`. The structure uses heights measured in feet and inches. One *inch* is 2.54 centimeters, and one *foot* is 12 inches. To ensure normalized values (inches < 12.0), the constructor performs *modulo* 12 arithmetic. The **+** operator uses the constructor to generate normalized values.

VB

```
Public Shadows Structure height
    ' Need Shadows because System.Windows.Forms.Form also defines property Height.
    Private feet As Integer
    Private inches As Double
    Public Sub New(ByVal f As Integer, ByVal i As Double)
        Me.feet = f + (CInt(i) \ 12)
        Me.inches = i Mod 12.0
    End Sub
    Public Overloads Function ToString() As String
        Return Me.feet & "' " & Me.inches & """"
    End Function
    Public Shared Operator +(ByVal h1 As height,
                             ByVal h2 As height) As height
        Return New height(h1.feet + h2.feet, h1.inches + h2.inches)
    End Operator
End Structure
```

You can test the structure `height` with the following code.

VB

```
Public Sub consumeHeight()
    Dim p1 As New height(3, 10)
    Dim p2 As New height(4, 8)
    Dim p3 As height = p1 + p2
    Dim s As String = p1.ToString() & " + " & p2.ToString() &
          " = " & p3.ToString() & " (= 8' 6"" ?)"
    Dim p4 As New height(2, 14)
    s &= vbCrLf & "2' 14"" = " & p4.ToString() & " (= 3' 2"" ?)"
```

```
        Dim p5 As New height(4, 24)
        s &= vbCrLf & "4' 24"" = " & p5.ToString() & " (= 6' 0"" ?)"
        MsgBox(s)
    End Sub
```

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

See Also

- Operator Procedures (Visual Basic)
- How to: Define a Conversion Operator (Visual Basic)
- How to: Call an Operator Procedure (Visual Basic)
- How to: Use a Class that Defines Operators (Visual Basic)
- Operator Statement
- Structure Statement
- How to: Declare a Structure (Visual Basic)
- Mod Operator (Visual Basic)

How to: Define a Conversion Operator (Visual Basic)

Visual Studio 2015

If you have defined a class or structure, you can define a type conversion operator between the type of your class or structure and another data type (such as **Integer**, **Double**, or **String**).

Define the type conversion as a CType Function (Visual Basic) procedure within the class or structure. All conversion procedures must be **Public Shared**, and each one must specify either Widening (Visual Basic) or Narrowing (Visual Basic).

Defining an operator on a class or structure is also called *overloading* the operator.

Example

The following example defines conversion operators between a structure called `digit` and a **Byte**.

VB

```
Public Structure digit
Private dig As Byte
    Public Sub New(ByVal b As Byte)
        If (b < 0 OrElse b > 9) Then Throw New System.ArgumentException(
            "Argument outside range for Byte")
        Me.dig = b
    End Sub
    Public Shared Widening Operator CType(ByVal d As digit) As Byte
        Return d.dig
    End Operator
    Public Shared Narrowing Operator CType(ByVal b As Byte) As digit
        Return New digit(b)
    End Operator
End Structure
```

You can test the structure `digit` with the following code.

VB

```
Public Sub consumeDigit()
    Dim d1 As New digit(4)
    Dim d2 As New digit(7)
    Dim d3 As digit = CType(CByte(3), digit)
    Dim s As String = "Initial 4 generates " & CStr(CType(d1, Byte)) &
          vbCrLf & "Initial 7 generates " & CStr(CType(d2, Byte)) &
          vbCrLf & "Converted 3 generates " & CStr(CType(d3, Byte))
    Try
        Dim d4 As digit
```

```
        d4 = CType(CType(d1, Byte) + CType(d2, Byte), digit)
    Catch e4 As System.Exception
        s &= vbCrLf & "4 + 7 generates " & """" & e4.Message & """"
    End Try
    Try
        Dim d5 As digit = CType(CByte(10), digit)
    Catch e5 As System.Exception
        s &= vbCrLf & "Initial 10 generates " & """" & e5.Message & """"
    End Try
    MsgBox(s)
End Sub
```

See Also

- Operator Procedures (Visual Basic)
- How to: Define an Operator (Visual Basic)
- How to: Call an Operator Procedure (Visual Basic)
- How to: Use a Class that Defines Operators (Visual Basic)
- Operator Statement
- Structure Statement
- How to: Declare a Structure (Visual Basic)
- Implicit and Explicit Conversions (Visual Basic)
- Widening and Narrowing Conversions (Visual Basic)

How to: Call an Operator Procedure (Visual Basic)

Visual Studio 2015

You call an operator procedure by using the operator symbol in an expression. In the case of a conversion operator, you call the CType Function (Visual Basic) to convert a value from one data type to another.

You do not call operator procedures explicitly. You just use the operator, or the **CType** function, in an assignment statement or an expression, the same way you ordinarily use an operator. Visual Basic makes the call to the operator procedure.

Defining an operator on a class or structure is also called *overloading* the operator.

To call an operator procedure

1. Use the operator symbol in an expression in the ordinary way.
2. Be sure the data types of the operands are appropriate for the operator, and in the correct order.
3. The operator contributes to the value of the expression as expected.

To call a conversion operator procedure

1. Use **CType** inside an expression.
2. Be sure the data types of the operands are appropriate for the conversion, and in the correct order.
3. **CType** calls the conversion operator procedure and returns the converted value.

Example

The following example creates two TimeSpan structures, adds them together, and stores the result in a third TimeSpan structure. The TimeSpan structure defines operator procedures to overload several standard operators.

VB

```
Dim firstSpan As New TimeSpan(3, 30, 0)
Dim secondSpan As New TimeSpan(1, 30, 30)
Dim combinedSpan As TimeSpan = firstSpan + secondSpan
Dim s As String = firstSpan.ToString() &
          " + " & secondSpan.ToString() &
          " = " & combinedSpan.ToString()
MsgBox(s)
```

Because TimeSpan overloads the standard **+** operator, the previous example calls an operator procedure when it calculates

the value of `combinedSpan`.

For an example of calling a conversation operator procedure, see How to: Use a Class that Defines Operators (Visual Basic).

Compiling the Code

Be sure the class or structure you are using defines the operator you want to use.

See Also

Operator Procedures (Visual Basic)
How to: Define an Operator (Visual Basic)
How to: Define a Conversion Operator (Visual Basic)
Operator Statement
Widening (Visual Basic)
Narrowing (Visual Basic)
Structure Statement
How to: Declare a Structure (Visual Basic)
Implicit and Explicit Conversions (Visual Basic)
Widening and Narrowing Conversions (Visual Basic)

How to: Use a Class that Defines Operators (Visual Basic)

Visual Studio 2015

If you are using a class or structure that defines its own operators, you can access those operators from Visual Basic.

Defining an operator on a class or structure is also called *overloading* the operator.

Example

The following example accesses the SQL structure SqlString, which defines the conversion operators (CType Function (Visual Basic)) in both directions between a SQL string and a Visual Basic string. Use **CType(***SQL string expression*, **String)** to convert a SQL string to a Visual Basic string, and **CType(***Visual Basic string expression*, SqlString**)** to convert in the other direction.

VB

```
' Insert the following line at the beginning of your source file.
Imports System.Data.SqlTypes
```

VB

```
Public Sub setJobString(ByVal g As Integer)
    Dim title As String
    Dim jobTitle As System.Data.SqlTypes.SqlString
    Select Case g
        Case 1
            title = "President"
        Case 2
            title = "Vice President"
        Case 3
            title = "Director"
        Case 4
            title = "Manager"
        Case Else
            title = "Worker"
    End Select
    jobTitle = CType(title, SqlString)
    MsgBox("Group " & CStr(g) & " generates title """ &
          CType(jobTitle, String) & """")
End Sub
```

The SqlString structure defines a conversion operator (CType Function (Visual Basic)) from **String** to SqlString and another from SqlString to **String**. The statement that assigns `title` to `jobTitle` makes use of the first operator, and the MsgBox function call uses the second.

Compiling the Code

Be sure the class or structure you are using defines the operator you want to use. Do not assume that the class or structure has defined every operator available for overloading. For a list of available operators, see Operator Statement.

Include the appropriate **Imports** statement for the SQL string at the beginning of your source file (in this case System.Data.SqlTypes).

Your project must have references to System.Data and System.XML.

See Also

Operator Procedures (Visual Basic)
How to: Define an Operator (Visual Basic)
How to: Define a Conversion Operator (Visual Basic)
How to: Call an Operator Procedure (Visual Basic)
Widening (Visual Basic)
Narrowing (Visual Basic)
Structure Statement
How to: Declare a Structure (Visual Basic)
Implicit and Explicit Conversions (Visual Basic)
Widening and Narrowing Conversions (Visual Basic)

Procedure Overloading (Visual Basic)

Visual Studio 2015

Overloading a procedure means defining it in multiple versions, using the same name but different parameter lists. The purpose of overloading is to define several closely related versions of a procedure without having to differentiate them by name. You do this by varying the parameter list.

Overloading Rules

When you overload a procedure, the following rules apply:

- **Same Name**. Each overloaded version must use the same procedure name.
- **Different Signature**. Each overloaded version must differ from all other overloaded versions in at least one of the following respects:
  - Number of parameters
  - Order of the parameters
  - Data types of the parameters
  - Number of type parameters (for a generic procedure)
  - Return type (only for a conversion operator)

  Together with the procedure name, the preceding items are collectively called the *signature* of the procedure. When you call an overloaded procedure, the compiler uses the signature to check that the call correctly matches the definition.
- **Items Not Part of Signature**. You cannot overload a procedure without varying the signature. In particular, you cannot overload a procedure by varying only one or more of the following items:
  - Procedure modifier keywords, such as **Public**, **Shared**, and **Static**
  - Parameter or type parameter names
  - Type parameter constraints (for a generic procedure)
  - Parameter modifier keywords, such as **ByRef** and **Optional**
  - Whether it returns a value
  - The data type of the return value (except for a conversion operator)

  The items in the preceding list are not part of the signature. Although you cannot use them to differentiate between overloaded versions, you can vary them among overloaded versions that are properly differentiated by their signatures.

- **Late-Bound Arguments**. If you intend to pass a late bound object variable to an overloaded version, you must declare the appropriate parameter as Object.

Multiple Versions of a Procedure

Suppose you are writing a **Sub** procedure to post a transaction against a customer's balance, and you want to be able to refer to the customer either by name or by account number. To accommodate this, you can define two different **Sub** procedures, as in the following example:

VB

```
Sub postName(ByVal custName As String, ByVal amount As Single)
    ' Insert code to access customer record by customer name.
End Sub
Sub postAcct(ByVal custAcct As Integer, ByVal amount As Single)
    ' Insert code to access customer record by account number.
End Sub
```

Overloaded Versions

An alternative is to overload a single procedure name. You can use the Overloads (Visual Basic) keyword to define a version of the procedure for each parameter list, as follows:

VB

```
Overloads Sub post(ByVal custName As String, ByVal amount As Single)
    ' Insert code to access customer record by customer name.
End Sub
Overloads Sub post(ByVal custAcct As Integer, ByVal amount As Single)
    ' Insert code to access customer record by account number.
End Sub
```

Additional Overloads

If you also wanted to accept a transaction amount in either **Decimal** or **Single**, you could further overload `post` to allow for this variation. If you did this to each of the overloads in the preceding example, you would have four **Sub** procedures, all with the same name but with four different signatures.

Advantages of Overloading

The advantage of overloading a procedure is in the flexibility of the call. To use the `post` procedure declared in the preceding example, the calling code can obtain the customer identification as either a **String** or an **Integer**, and then call the same procedure in either case. The following example illustrates this:

VB

```
Imports MSVB = Microsoft.VisualBasic
```

VB

```
Dim customer As String
Dim accountNum As Integer
Dim amount As Single
customer = MSVB.Interaction.InputBox("Enter customer name or number")
amount = MSVB.Interaction.InputBox("Enter transaction amount")
Try
    accountNum = CInt(customer)
    Call post(accountNum, amount)
Catch
    Call post(customer, amount)
End Try
```

See Also

Procedures in Visual Basic
How to: Define Multiple Versions of a Procedure (Visual Basic)
How to: Call an Overloaded Procedure (Visual Basic)
How to: Overload a Procedure that Takes Optional Parameters (Visual Basic)
How to: Overload a Procedure that Takes an Indefinite Number of Parameters (Visual Basic)
Considerations in Overloading Procedures (Visual Basic)
Overload Resolution (Visual Basic)
Overloads (Visual Basic)
Generic Types in Visual Basic (Visual Basic)

How to: Define Multiple Versions of a Procedure (Visual Basic)

Visual Studio 2015

You can define a procedure in multiple versions by *overloading* it, using the same name but a different parameter list for each version. The purpose of overloading is to define several closely related versions of a procedure without having to differentiate them by name.

For more information, see Procedure Overloading (Visual Basic).

To define multiple versions of a procedure

1. Write a **Sub** or **Function** declaration statement for each version of the procedure you want to define. Use the same procedure name in every declaration.
2. Precede the **Sub** or **Function** keyword in each declaration with the Overloads (Visual Basic) keyword. You can optionally omit **Overloads** in the declarations, but if you include it in any of the declarations, you must include it in every declaration.
3. Following each declaration statement, write procedure code to handle the specific case where the calling code supplies arguments matching that version's parameter list. You do not have to test for which parameters the calling code has supplied. Visual Basic passes control to the matching version of your procedure.
4. Terminate each version of the procedure with the **End Sub** or **End Function** statement as appropriate.

Example

The following example defines a **Sub** procedure to post a transaction against a customer's balance. It uses the **Overloads** keyword to define two versions of the procedure, one that accepts the customer by name and the other by account number.

VB

```
Overloads Sub post(ByVal custName As String, ByVal amount As Single)
    ' Insert code to access customer record by customer name.
End Sub
Overloads Sub post(ByVal custAcct As Integer, ByVal amount As Single)
    ' Insert code to access customer record by account number.
End Sub
```

The calling code can obtain the customer identification as either a **String** or an **Integer**, and then use the same calling statement in either case.

For information on how to call these versions of the `post` procedure, see How to: Call an Overloaded Procedure (Visual Basic).

Compiling the Code

Make sure each of your overloaded versions has the same procedure name but a different parameter list.

See Also

Procedures in Visual Basic
Procedure Parameters and Arguments (Visual Basic)
Troubleshooting Procedures (Visual Basic)
How to: Overload a Procedure that Takes Optional Parameters (Visual Basic)
How to: Overload a Procedure that Takes an Indefinite Number of Parameters (Visual Basic)
Considerations in Overloading Procedures (Visual Basic)
Overload Resolution (Visual Basic)

How to: Call an Overloaded Procedure (Visual Basic)

Visual Studio 2015

The advantage of overloading a procedure is in the flexibility of the call. The calling code can obtain the information it needs to pass to the procedure and then call a single procedure name, no matter what arguments it is passing.

To call a procedure that has more than one version defined

1. In the calling code, determine which data to pass to the procedure.
2. Write the procedure call in the normal way, presenting the data in the argument list. Be sure the arguments match the parameter list in one of the versions defined for the procedure.
3. You do not have to determine which version of the procedure to call. Visual Basic passes control to the version matching your argument list.

   The following example calls the `post` procedure declared in How to: Define Multiple Versions of a Procedure (Visual Basic). It obtains the customer identification, determines whether it is a **String** or an **Integer**, and then in either case calls the same procedure.

   VB

   ```
   Imports MSVB = Microsoft.VisualBasic
   ```

   VB

   ```
   Dim customer As String
   Dim accountNum As Integer
   Dim amount As Single
   customer = MSVB.Interaction.InputBox("Enter customer name or number")
   amount = MSVB.Interaction.InputBox("Enter transaction amount")
   Try
       accountNum = CInt(customer)
       Call post(accountNum, amount)
   Catch
       Call post(customer, amount)
   End Try
   ```

See Also

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

Procedure Overloading (Visual Basic)
Troubleshooting Procedures (Visual Basic)
How to: Define Multiple Versions of a Procedure (Visual Basic)
How to: Overload a Procedure that Takes Optional Parameters (Visual Basic)
How to: Overload a Procedure that Takes an Indefinite Number of Parameters (Visual Basic)
Considerations in Overloading Procedures (Visual Basic)
Overload Resolution (Visual Basic)
Overloads (Visual Basic)

How to: Overload a Procedure that Takes Optional Parameters (Visual Basic)

Visual Studio 2015

If a procedure has one or more Optional (Visual Basic) parameters, you cannot define an overloaded version matching any of its implicit overloads. For more information, see "Implicit Overloads for Optional Parameters" in Considerations in Overloading Procedures (Visual Basic).

One Optional Parameter

To overload a procedure that takes one optional parameter

1. Write a **Sub** or **Function** declaration statement that includes the optional parameter in the parameter list. Do not use the **Optional** keyword in this overloaded version.
2. Precede the **Sub** or **Function** keyword with the Overloads (Visual Basic) keyword.
3. Write the procedure code that should execute when the calling code supplies the optional argument.
4. Terminate the procedure with the **End Sub** or **End Function** statement as appropriate.
5. Write a second declaration statement that is identical to the first declaration except that it does not include the optional parameter in the parameter list.
6. Write the procedure code that should execute when the calling code does not supply the optional argument. Terminate the procedure with the **End Sub** or **End Function** statement as appropriate.

   The following example shows a procedure defined with an optional parameter, an equivalent set of two overloaded procedures, and finally examples of both invalid and valid overloaded versions.

VB

```
Sub q(ByVal b As Byte, Optional ByVal j As Long = 6)
```

VB

```
' The preceding definition is equivalent to the following two overloads.
' Overloads Sub q(ByVal b As Byte)
' Overloads Sub q(ByVal b As Byte, ByVal j As Long)
```

VB

```
' Therefore, the following overload is not valid because the signature is already
in use.
' Overloads Sub q(ByVal c As Byte, ByVal k As Long)
```

```
' The following overload uses a different signature and is valid.
Overloads Sub q(ByVal b As Byte, ByVal j As Long, ByVal s As Single)
```

Multiple Optional Parameters

For a procedure with more than one optional parameter, you normally need more than two overloaded versions. For example, if there are two optional parameters, and the calling code can supply or omit each one independently of the other, you need four overloaded versions, one for each possible combination of supplied arguments.

As the number of optional parameters increases, the complexity of the overloading increases. Unless some combinations of supplied arguments are not acceptable, for *N* optional parameters you need *2 ^ N* overloaded versions. Depending on the nature of the procedure, you might find that the clarity of logic justifies the extra effort of defining all the overloaded versions.

To overload a procedure that takes more than one optional parameter

1. Determine which combinations of supplied optional arguments are acceptable to the logic of the procedure. An unacceptable combination might arise if one optional parameter depends on another. For example, if one parameter accepts a spouse's name and another accepts the spouse's age, a combination of arguments supplying the age but omitting the name is unacceptable.

2. For each acceptable combination of supplied optional arguments, write a **Sub** or **Function** declaration statement that defines the corresponding parameter list. Do not use the **Optional** keyword.

3. In each declaration, precede the **Sub** or **Function** keyword with the Overloads (Visual Basic) keyword.

4. Following each declaration, write the procedure code that should execute when the calling code supplies an argument list corresponding to that declaration's parameter list.

5. Terminate each procedure with the **End Sub** or **End Function** statement as appropriate.

See Also

Procedures in Visual Basic
Procedure Parameters and Arguments (Visual Basic)
Optional Parameters (Visual Basic)
Parameter Arrays (Visual Basic)
Procedure Overloading (Visual Basic)
Troubleshooting Procedures (Visual Basic)
How to: Define Multiple Versions of a Procedure (Visual Basic)
How to: Call an Overloaded Procedure (Visual Basic)
How to: Overload a Procedure that Takes an Indefinite Number of Parameters (Visual Basic)
Overload Resolution (Visual Basic)

How to: Overload a Procedure that Takes an Indefinite Number of Parameters (Visual Basic)

Visual Studio 2015

If a procedure has a ParamArray (Visual Basic) parameter, you cannot define an overloaded version taking a one-dimensional array for the parameter array. For more information, see "Implicit Overloads for a ParamArray Parameter" in Considerations in Overloading Procedures (Visual Basic).

To overload a procedure that takes a variable number of parameters

1. Ascertain that the procedure and calling code logic benefits from overloaded versions more than from a **ParamArray** parameter. See "Overloads and ParamArrays" in Considerations in Overloading Procedures (Visual Basic).
2. Determine which numbers of supplied values the procedure should accept in the variable part of the parameter list. This might include the case of no value, and it might include the case of a single one-dimensional array.
3. For each acceptable number of supplied values, write a **Sub** or **Function** declaration statement that defines the corresponding parameter list. Do not use either the **Optional** or the **ParamArray** keyword in this overloaded version.
4. In each declaration, precede the **Sub** or **Function** keyword with the Overloads (Visual Basic) keyword.
5. Following each declaration, write the procedure code that should execute when the calling code supplies values corresponding to that declaration's parameter list.
6. Terminate each procedure with the **End Sub** or **End Function** statement as appropriate.

Example

The following example shows a procedure defined with a ParamArray (Visual Basic) parameter, and then an equivalent set of overloaded procedures.

VB

```
Sub p(ByVal d As Date, ByVal ParamArray c() As Char)
```

VB

```
' The preceding definition is equivalent to the following overloads.
' Overloads Sub p(ByVal d As Date)
' Overloads Sub p(ByVal d As Date, ByVal c() As Char)
' Overloads Sub p(ByVal d As Date, ByVal c1 As Char)
```

```
' Overloads Sub p(ByVal d As Date, ByVal c1 As Char, ByVal c2 As Char)
' And so on, with an additional Char argument in each successive overload.
```

You cannot overload such a procedure with a parameter list that takes a one-dimensional array for the parameter array. However, you can use the signatures of the other implicit overloads. The following declarations illustrate this.

VB

```
' The following overload is not valid because it takes an array for the parameter array.
' Overloads Sub p(ByVal x As Date, ByVal y() As Char)
' The following overload takes a single value for the parameter array and is valid.
Overloads Sub p(ByVal z As Date, ByVal w As Char)
```

The code in the overloaded versions does not have to test whether the calling code supplied one or more values for the **ParamArray** parameter, or if so, how many. Visual Basic passes control to the version matching the calling argument list.

Compiling the Code

Because a procedure with a **ParamArray** parameter is equivalent to a set of overloaded versions, you cannot overload such a procedure with a parameter list corresponding to any of these implicit overloads. For more information, see Considerations in Overloading Procedures (Visual Basic).

.NET Framework Security

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 length of the array the calling code passed to it, and take appropriate steps if it is too large for your application.

See Also

- Procedures in Visual Basic
- Procedure Parameters and Arguments (Visual Basic)
- Optional Parameters (Visual Basic)
- Parameter Arrays (Visual Basic)
- Procedure Overloading (Visual Basic)
- Troubleshooting Procedures (Visual Basic)
- How to: Define Multiple Versions of a Procedure (Visual Basic)
- How to: Call an Overloaded Procedure (Visual Basic)
- How to: Overload a Procedure that Takes Optional Parameters (Visual Basic)
- Overload Resolution (Visual Basic)

Considerations in Overloading Procedures (Visual Basic)

Visual Studio 2015

When you overload a procedure, you must use a different *signature* for each overloaded version. This usually means each version must specify a different parameter list. For more information, see "Different Signature" in Procedure Overloading (Visual Basic).

You can overload a **Function** procedure with a **Sub** procedure, and vice versa, provided they have different signatures. Two overloads cannot differ only in that one has a return value and the other does not.

You can overload a property the same way you overload a procedure, and with the same restrictions. However, you cannot overload a procedure with a property, or vice versa.

Alternatives to Overloaded Versions

You sometimes have alternatives to overloaded versions, particularly when the presence of arguments is optional or their number is variable.

Keep in mind that optional arguments are not necessarily supported by all languages, and parameter arrays are limited to Visual Basic. If you are writing a procedure that is likely to be called from code written in any of several different languages, overloaded versions offer the greatest flexibility.

Overloads and Optional Arguments

When the calling code can optionally supply or omit one or more arguments, you can define multiple overloaded versions or use optional parameters.

When to Use Overloaded Versions

You can consider defining a series of overloaded versions in the following cases:

- The logic in the procedure code is significantly different depending on whether the calling code supplies an optional argument or not.
- The procedure code cannot reliably test whether the calling code has supplied an optional argument. This is the case, for example, if there is no possible candidate for a default value that the calling code could not be expected to supply.

When to Use Optional Parameters

You might prefer one or more optional parameters in the following cases:

- The only required action when the calling code does not supply an optional argument is to set the parameter to a default value. In such a case, the procedure code can be less complicated if you define a single version with one or more **Optional** parameters.

For more information, see Optional Parameters (Visual Basic).

Overloads and ParamArrays

When the calling code can pass a variable number of arguments, you can define multiple overloaded versions or use a parameter array.

When to Use Overloaded Versions

You can consider defining a series of overloaded versions in the following cases:

- You know that the calling code never passes more than a small number of values to the parameter array.
- The logic in the procedure code is significantly different depending on how many values the calling code passes.
- The calling code can pass values of different data types.

When to Use a Parameter Array

You are better served by a **ParamArray** parameter in the following cases:

- You are not able to predict how many values the calling code can pass to the parameter array, and it could be a large number.
- The procedure logic lends itself to iterating through all the values the calling code passes, performing essentially the same operations on every value.

For more information, see Parameter Arrays (Visual Basic).

Implicit Overloads for Optional Parameters

A procedure with an Optional (Visual Basic) parameter is equivalent to two overloaded procedures, one with the optional parameter and one without it. You cannot overload such a procedure with a parameter list corresponding to either of these. The following declarations illustrate this.

VB

```
Overloads Sub q(ByVal b As Byte, Optional ByVal j As Long = 6)
```

VB

```
' The preceding definition is equivalent to the following two overloads.
' Overloads Sub q(ByVal b As Byte)
' Overloads Sub q(ByVal b As Byte, ByVal j As Long)
```

VB

```
' Therefore, the following overload is not valid because the signature is already in
use.
' Overloads Sub q(ByVal c As Byte, ByVal k As Long)
' The following overload uses a different signature and is valid.
Overloads Sub q(ByVal b As Byte, ByVal j As Long, ByVal s As Single)
```

For a procedure with more than one optional parameter, there is a set of implicit overloads, arrived at by logic similar to that in the preceding example.

Implicit Overloads for a ParamArray Parameter

The compiler considers a procedure with a ParamArray (Visual Basic) parameter to have an infinite number of overloads, differing from each other in what the calling code passes to the parameter array, as follows:

- One overload for when the calling code does not supply an argument to the **ParamArray**
- One overload for when the calling code supplies a one-dimensional array of the **ParamArray** element type
- For every positive integer, one overload for when the calling code supplies that number of arguments, each of the **ParamArray** element type

The following declarations illustrate these implicit overloads.

VB

```
Overloads Sub p(ByVal d As Date, ByVal ParamArray c() As Char)
```

VB

```
' The preceding definition is equivalent to the following overloads.
' Overloads Sub p(ByVal d As Date)
' Overloads Sub p(ByVal d As Date, ByVal c() As Char)
' Overloads Sub p(ByVal d As Date, ByVal c1 As Char)
' Overloads Sub p(ByVal d As Date, ByVal c1 As Char, ByVal c2 As Char)
' And so on, with an additional Char argument in each successive overload.
```

You cannot overload such a procedure with a parameter list that takes a one-dimensional array for the parameter array. However, you can use the signatures of the other implicit overloads. The following declarations illustrate this.

VB

```
' The following overload is not valid because it takes an array for the parameter array.
' Overloads Sub p(ByVal x As Date, ByVal y() As Char)
' The following overload takes a single value for the parameter array and is valid.
Overloads Sub p(ByVal z As Date, ByVal w As Char)
```

Typeless Programming as an Alternative to Overloading

If you want to allow the calling code to pass different data types to a parameter, an alternative approach is typeless programming. You can set the type checking switch to **Off** with either the Option Strict Statement or the /optionstrict compiler option. Then you do not have to declare the parameter's data type. However, this approach has the following disadvantages compared to overloading:

- Typeless programming produces less efficient execution code.
- The procedure must test for every data type it anticipates being passed.
- The compiler cannot signal an error if the calling code passes a data type that the procedure does not support.

See Also

Procedures in Visual Basic
Procedure Parameters and Arguments (Visual Basic)
Troubleshooting Procedures (Visual Basic)
How to: Define Multiple Versions of a Procedure (Visual Basic)
How to: Call an Overloaded Procedure (Visual Basic)
How to: Overload a Procedure that Takes Optional Parameters (Visual Basic)
How to: Overload a Procedure that Takes an Indefinite Number of Parameters (Visual Basic)
Overload Resolution (Visual Basic)
Overloads (Visual Basic)

Overload Resolution (Visual Basic)

Visual Studio 2015

When the Visual Basic compiler encounters a call to a procedure that is defined in several overloaded versions, the compiler must decide which of the overloads to call. It does this by performing the following steps:

1. **Accessibility.** It eliminates any overload with an access level that prevents the calling code from calling it.

2. **Number of Parameters.** It eliminates any overload that defines a different number of parameters than are supplied in the call.

3. **Parameter Data Types.** The compiler gives instance methods preference over extension methods. If any instance method is found that requires only widening conversions to match the procedure call, all extension methods are dropped and the compiler continues with only the instance method candidates. If no such instance method is found, it continues with both instance and extension methods.

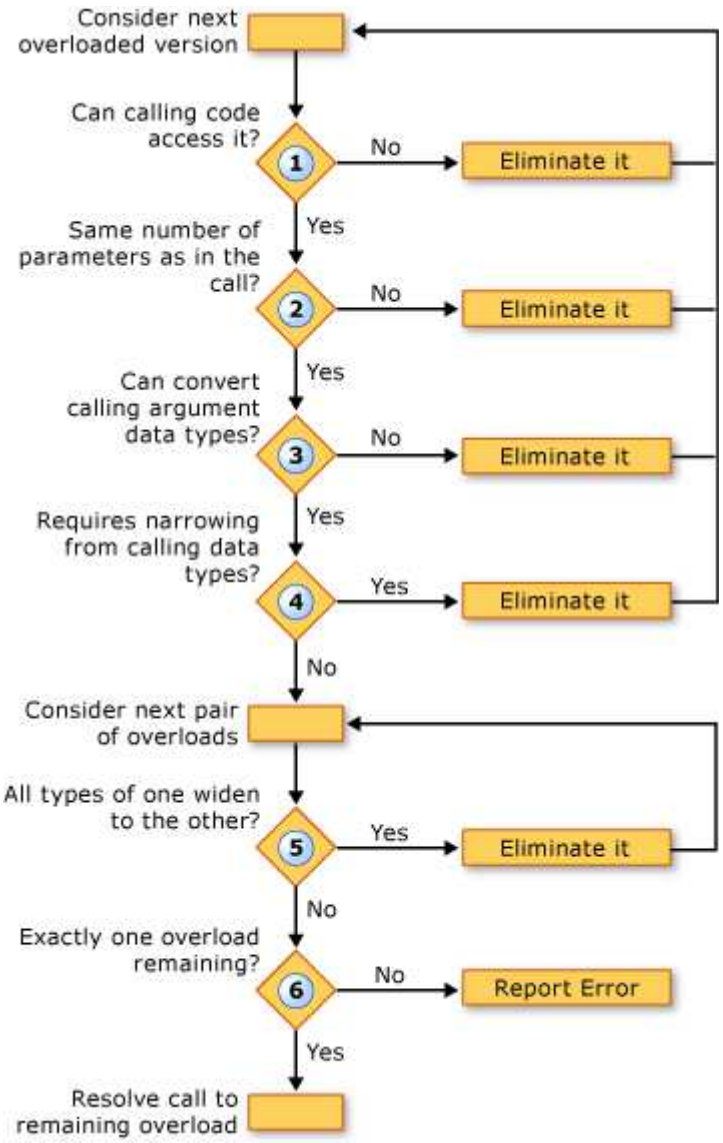
   In this step, it eliminates any overload for which the data types of the calling arguments cannot be converted to the parameter types defined in the overload.

4. **Narrowing Conversions.** It eliminates any overload that requires a narrowing conversion from the calling argument types to the defined parameter types. This is true whether the type checking switch (Option Strict Statement) is **On** or **Off**.

5. **Least Widening.** The compiler considers the remaining overloads in pairs. For each pair, it compares the data types of the defined parameters. If the types in one of the overloads all widen to the corresponding types in the other, the compiler eliminates the latter. That is, it retains the overload that requires the least amount of widening.

6. **Single Candidate.** It continues considering overloads in pairs until only one overload remains, and it resolves the call to that overload. If the compiler cannot reduce the overloads to a single candidate, it generates an error.

The following illustration shows the process that determines which of a set of overloaded versions to call.



Resolving among overloaded versions

The following example illustrates this 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.

Overloaded 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 as follows:

If the call supplies the last argument as	The compiler resolves the call to the overload declaring the last argument as
No value (argument omitted)	**Optional**
A single value	**Optional**
Two or more values in a comma-separated list	**ParamArray**
An array of any length (including an empty array)	**ParamArray**

See Also

Optional Parameters (Visual Basic)
Parameter Arrays (Visual Basic)
Procedure Overloading (Visual Basic)
Troubleshooting Procedures (Visual Basic)
How to: Define Multiple Versions of a Procedure (Visual Basic)
How to: Call an Overloaded Procedure (Visual Basic)
How to: Overload a Procedure that Takes Optional Parameters (Visual Basic)
How to: Overload a Procedure that Takes an Indefinite Number of Parameters (Visual Basic)
Considerations in Overloading Procedures (Visual Basic)
Overloads (Visual Basic)
Extension Methods (Visual Basic)