Visual Basic Coding Conventions

In this article

Microsoft develops samples and documentation that follow the guidelines in this topic. If you follow the same coding conventions, you may gain the following benefits:
Your code will have a consistent look, so that readers can better focus on content, not layout.
Readers understand your code more quickly because they can make assumptions based on previous experience.
You can copy, change, and maintain the code more easily.
You help ensure that your code demonstrates "best practices" for Visual Basic.

Naming Conventions

For information about naming guidelines, see Naming Guidelines topic.
Do not use "My" or "my" as part of a variable name. This practice creates confusion with the My objects.
You do not have to change the names of objects in auto-generated code to make them fit the guidelines.

Layout Conventions

Insert tabs as spaces, and use smart indenting with four-space indents.
Use **Pretty listing (reformatting) of code** to reformat your code in the code editor. For more information, see Options, Text Editor, Basic (Visual Basic).
Use only one statement per line. Don't use the Visual Basic line separator character (:).
Avoid using the explicit line continuation character "_" in favor of implicit line continuation wherever the language allows it.
Use only one declaration per line.
If **Pretty listing (reformatting) of code** doesn't format continuation lines automatically, manually indent continuation lines one tab stop. However, always left-align items in a list.
Copy

```
a As Integer,
b As Integer
```

Add at least one blank line between method and property definitions.

Commenting Conventions

Put comments on a separate line instead of at the end of a line of code.
Start comment text with an uppercase letter, and end comment text with a period.
Insert one space between the comment delimiter (') and the comment text.
VBCopy

```
' Here is a comment.
```

Do not surround comments with formatted blocks of asterisks.

Program Structure

When you use the Main method, use the default construct for new console applications, and use My for command-line arguments.
VBCopy

```
Sub Main()
  For Each argument As String In My.Application.CommandLineArgs
    ' Add code here to use the string variable.
  Next
End Sub
```

Language Guidelines

String Data Type

To concatenate strings, use an ampersand (&).
VBCopy

```
MsgBox("hello" & vbCrLf & "goodbye")
```

To append strings in loops, use the StringBuilder object.
VBCopy

```
Dim longString As New System.Text.StringBuilder
For count As Integer = 1 To 1000
  longString.Append(count)
Next
```

Relaxed Delegates in Event Handlers

Do not explicitly qualify the arguments (Object and EventArgs) to event handlers. If you are not using the event arguments that are passed to an event (for example, sender as Object, e as EventArgs), use relaxed delegates, and leave out the event arguments in your code:
VBCopy
```
Public Sub Form1_Load() Handles Form1.Load
End Sub
```

Unsigned Data Type

Use Integer rather than unsigned types, except where they are necessary.

Arrays

Use the short syntax when you initialize arrays on the declaration line. For example, use the following syntax.
VBCopy
```
Dim letters1 As String() = {"a", "b", "c"}
```

Do not use the following syntax.
VBCopy
```
Dim letters2() As String = New String() {"a", "b", "c"}
```

Put the array designator on the type, not on the variable. For example, use the following syntax:
VBCopy
```
Dim letters4 As String() = {"a", "b", "c"}
```

Do not use the following syntax:
VBCopy
```
Dim letters3() As String = {"a", "b", "c"}
```

Use the { } syntax when you declare and initialize arrays of basic data types. For example, use the following syntax:
VBCopy
```
Dim letters5 As String() = {"a", "b", "c"}
```

Do not use the following syntax:
VBCopy
```
Dim letters6(2) As String
letters6(0) = "a"
letters6(1) = "b"
letters6(2) = "c"
```

Use the With Keyword

When you make a series of calls to one object, consider using the With keyword:

VBCopy
```
With orderLog
  .Log = "Application"
  .Source = "Application Name"
  .MachineName = "Computer Name"
End With
```

Use the Try...Catch and Using Statements when you use Exception Handling

Do not use On Error Goto.

Use the IsNot Keyword

Use the IsNot keyword instead of Not...Is Nothing.

New Keyword

Use short instantiation. For example, use the following syntax:

VBCopy
```
Dim employees As New List(Of String)
```

The preceding line is equivalent to this:

VBCopy
```
Dim employees2 As List(Of String) = New List(Of String)
```

Use object initializers for new objects instead of the parameterless constructor:

VBCopy
```
Dim orderLog As New EventLog With {
    .Log = "Application",
    .Source = "Application Name",
    .MachineName = "Computer Name"}
```

Event Handling

Use Handles rather than AddHandler:

VBCopy
```
Private Sub ToolStripMenuItem1_Click() Handles ToolStripMenuItem1.Click
End Sub
```

Use AddressOf, and do not instantiate the delegate explicitly:

VBCopy
```
Dim closeItem As New ToolStripMenuItem(
    "Close", Nothing, AddressOf ToolStripMenuItem1_Click)
Me.MainMenuStrip.Items.Add(closeItem)
```

When you define an event, use the short syntax, and let the compiler define the delegate:
VBCopy
```
Public Event SampleEvent As EventHandler(Of SampleEventArgs)
' or
Public Event SampleEvent(ByVal source As Object,
                         ByVal e As SampleEventArgs)
```

Do not verify whether an event is Nothing (null) before you call the RaiseEvent method. RaiseEvent checks for Nothing before it raises the event.

Using Shared Members

Call Shared members by using the class name, not from an instance variable.

Use XML Literals

XML literals simplify the most common tasks that you encounter when you work with XML (for example, load, query, and transform). When you develop with XML, follow these guidelines:

- Use XML literals to create XML documents and fragments instead of calling XML APIs directly.
- Import XML namespaces at the file or project level to take advantage of the performance optimizations for XML literals.
- Use the XML axis properties to access elements and attributes in an XML document.
- Use embedded expressions to include values and to create XML from existing values instead of using API calls such as the Add method:
- VBCopy
- Private Function GetHtmlDocument(

```
    ByVal items As IEnumerable(Of XElement)) As String

  Dim htmlDoc = <html>
                  <body>
                    <table border="0" cellspacing="2">
                      <%=
                        From item In items
                        Select <tr>
                                 <td style="width:480">
                                   <%= item.<title>.Value %>
                                 </td>
                                 <td><%= item.<pubDate>.Value %></td>
                               </tr>
                      %>
                    </table>
                  </body>
                </html>

  Return htmlDoc.ToString()
```

```
        End Function
```

LINQ Queries

Use meaningful names for query variables:

VBCopy

```
Dim seattleCustomers = From cust In customers
                       Where cust.City = "Seattle"
```

Provide names for elements in a query to make sure that property names of anonymous types are correctly capitalized using Pascal casing:

VBCopy

```
Dim customerOrders = From customer In customers
                     Join order In orders
                       On customer.CustomerID Equals order.CustomerID
                     Select Customer = customer, Order = order
```

Rename properties when the property names in the result would be ambiguous. For example, if your query returns a customer name and an order ID, rename them instead of leaving them as Name and ID in the result:

VBCopy

```
Dim customerOrders2 = From cust In customers
                      Join ord In orders
                        On cust.CustomerID Equals ord.CustomerID
                      Select CustomerName = cust.Name,
                             OrderID = ord.ID
```

Use type inference in the declaration of query variables and range variables:

VBCopy

```
Dim customerList = From cust In customers
```

Align query clauses under the From statement:

VBCopy

```
Dim newyorkCustomers = From cust In customers
                       Where cust.City = "New York"
                       Select cust.LastName, cust.CompanyName
```

Use Where clauses before other query clauses so that later query clauses operate on the filtered set of data:

VBCopy

```
Dim newyorkCustomers2 = From cust In customers
                        Where cust.City = "New York"
                        Order By cust.LastName
```

Use the Join clause to explicitly define a join operation instead of using the Where clause to implicitly define a join operation:

VBCopy

```
Dim customerList2 = From cust In customers
                    Join order In orders
                      On cust.CustomerID Equals order.CustomerID
                    Select cust, order
```