VB.NET: What Happened to Control Arrays!!!

(Part I)

VB.NET no longer supports VB6 Control Arrays
Here's what you can do about it. Part II is another point of view.

2009 Note … This was written for the 2003 Version of VB.NET. The use of the "Region" as described isn't used in current versions of VB.NET.

Frank Bugeja, About Visual Basic reader from Australia ...

<**G'day Mate!**>

... writes,

I am a senior computing studies teacher teaching software design and development. I have been using Visual Basic 6.

Despite the fact that online experts of VB.NET and authors of VB.NET textbooks are saying that the omission of control arrays in VB.NET is a positive step, the omission of control arrays from VB.NET is, for educational purposes in teaching about arrays, a decidedly backward step.

Why?

1. No longer is it possible to simply copy a control, such as a textbox, and then paste it (once or several times) to create a control array.
2. The VB.NET code for creating a structure similar to a control array has been, in all the books on VB.NET that I have bought and online, much longer and much more complex. It lacks the simplicity of coding a control array that is found in VB6.

For example, to clear a control array of 6 labels of text in VB6, it is as simple as this:

```
For a = 1 to 6
       Label(a).Caption = ""
Next
```

Could you please show me how I can create a "control array" structure in VB.NET, and how it would be coded (using labels as the controls).

Well, mate ... Pull your swag up closer to the bush telly and we'll have a go at it!

In the style of a Clint Eastwood spaghetti western, I'm going to give you the Good, the Bad, and the really, really UGLY about this. And I do mean Ugly. We're talking "kiss your maiden aunt through an inch of pancake makeup" ugly here. In reverse order ...

The Ugly

You're absolutely right!

Well ... You're mainly right. If you reference the VB6 compatibility library, there are objects in there that act pretty much like control arrays. To see what I mean, simply use the VB.NET upgrade wizard with a program that contains a control array. The code is back to uh-uh-ugly again, but it works. The bad news is that Microsoft will not guarantee that the compatibility components will continue to be supported and you're not supposed to use them.

The VB.NET code to create and use "control arrays" is much longer and much more complex. But you don't have to buy a book to see that. You just have to check out Microsoft's white paper about it: Creating Control Arrays in Visual Basic .NET. Microsoft heard this complaint quite a bit so they wrote this official response. The code in the white paper is for both C# and VB.NET and also uses buttons rather than labels as you requested. So here's my (hopefully) simplified version of the white paper using labels.

According to Microsoft, to do something even close to what you can do in VB6 requires the creation a "simple component that duplicates control array functionality". I'll let you be the judge ... but it doesn't look like a duplicate to me. And it's nothing close to being as simple.

First the code.

You need both a new class and a hosting form to illustrate this. The end result can be seen at the top of this article. The class actually creates and destroys new labels. The complete class code is as follows:

```
Public Class LabelArray
    Inherits System.Collections.CollectionBase
    Private ReadOnly HostForm As _
    System.Windows.Forms.Form
    Public Function AddNewLabel() _
    As System.Windows.Forms.Label
        ' Create a new instance of the Label class.
        Dim aLabel As New System.Windows.Forms.Label
        ' Add the Label to the collection's
        ' internal list.
        Me.List.Add(aLabel)
        ' Add the Label to the Controls collection
        ' of the Form referenced by the HostForm field.
        HostForm.Controls.Add(aLabel)
        ' Set intial properties for the Label object.
        aLabel.Top = Count * 25
        aLabel.Width = 50
```

```
            aLabel.Left = 140
            aLabel.Tag = Me.Count
            aLabel.Text = "Label " & Me.Count.ToString
            Return aLabel
        End Function
        Public Sub New( _
            ByVal host As System.Windows.Forms.Form)
            HostForm = host
            Me.AddNewLabel()
        End Sub
        Default Public ReadOnly Property _
            Item(ByVal Index As Integer) As _
            System.Windows.Forms.Label
            Get
                Return CType(Me.List.Item(Index), _
                System.Windows.Forms.Label)
            End Get
        End Property


        Public Sub Remove()
            ' Check to be sure there is a Label to remove.
            If Me.Count > 0 Then
                ' Remove the last Label added to the array
                ' from the host form controls collection.
                ' Note the use of the default property in
                ' accessing the array.
                HostForm.Controls.Remove(Me(Me.Count - 1))
                Me.List.RemoveAt(Me.Count - 1)
            End If
        End Sub
    End Class
```

To illustrate how this class code would be used, you could create a Form that calls it. You would have to use the code shown below in the form:

```
    Public Class Form1
        Inherits System.Windows.Forms.Form
    #Region " Windows Form Designer generated code "
    ' Also you must add the statement:
    '   MyControlArray = New LabelArray(Me)
    ' after the InitializeComponent() call in the
    ' hidden Region code.
        ' Declare a new ButtonArray object.
        Dim MyControlArray As LabelArray
        Private Sub btnLabelAdd_Click( _
            ByVal sender As System.Object, _
            ByVal e As System.EventArgs) _
            Handles btnLabelAdd.Click
            ' Call the AddNewLabel method
            ' of MyControlArray.
            MyControlArray.AddNewLabel()
            ' Change the BackColor property
            ' of the Button 0.
            MyControlArray(0).BackColor = _
```

```
            System.Drawing.Color.Red
        End Sub



        Private Sub btnLabelRemove_Click( _
            ByVal sender As System.Object, _
            ByVal e As System.EventArgs) _
            Handles btnLabelRemove.Click
            ' Call the Remove method of MyControlArray.
            MyControlArray.Remove()
        End Sub
    End Class
```

First, this doesn't even do the job at Design Time like we used to do it in VB6! And second, they aren't in an array, they are in a VB.NET Collection - a much different thing than an array.

The Bad

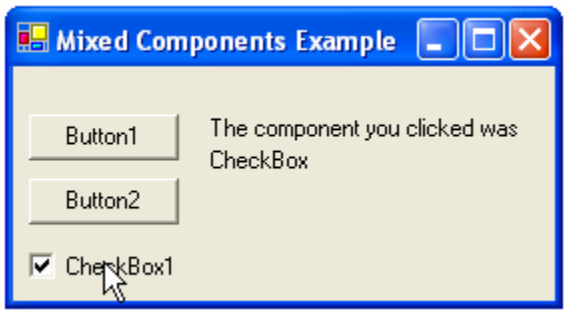
Microsoft has the ***nerve*** lie to all of us by calling this a "duplicate"!

The Good

Somehow ... I think that the marketing people at Microsoft are really to blame for problems like this. They are the ones who try to keep the fiction alive that VB.NET and VB6 are anything more than distantly related. If you just accept the fact that VB.NET is a whole new language ... with great new features! ... then you can understand things a lot better. Regular readers of this site know that I'm a real advocate of VB.NET. I think it's a great language and is destined to dominate the development landscape more than VB6 ever did.

The reason VB.NET doesn't support the VB6 "Control Array" is that there is no such thing as a "Control" "Array" (note the change of quotation marks). VB6 creates a collection "behind the scenes" and makes it appear as an array to the developer. But it's not an array and you have little control over it beyond the functions provided through the IDE.

VB.NET, on the other hand, calls it what it is: a collection of objects. And they hand the keys to the kingdom to the developer by creating the whole thing right out in the open.



As an example of the kind of advantages this gives the developer, in VB6 the controls had to be of the same type, and they had to have the same name. Since these are just objects in VB.NET, you can make them different types and give them different names and still manage them in the same collection of objects.

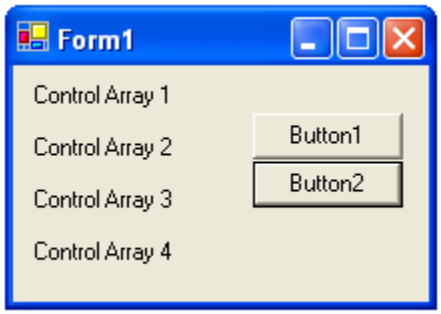
In this example, the same Click event handles two buttons and a checkbox and displays which one was clicked. Do that in one line of code with VB6!

```
Private Sub MixedControls_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles Button1.Click, _
            Button2.Click, _
            CheckBox1.Click
    ' The statement below has to be one long statement!
    ' It's on four lines here to keep it narrow
    ' enough to fit on a web page
    Label2.Text =
    Microsoft.VisualBasic.Right(sender.GetType.ToString, _
    Len(sender.GetType.ToString) _
    (InStr(sender.GetType.ToString, "Forms") + 5))
End Sub
```

The substring calculation is kind of complex, but it isn't really what we're talking about here. You could do anything in the Click event. You could, for example, use the Type of the control in an If statement to do different things for different controls.

(Part II)

VB.NET no longer supports VB6 Control Arrays
Or does it? The Computing Studies Group disagrees.



Frank was kind enough to forward some feedback that he received from a Computing Studies Group that he is a member of when he asked the same question about control arrays.

In brief, people in Frank's group said, "Hey! You can have a control array in VB.NET! And it's a lot more simple than the 'Ugly' code sample in the article." And they provided an example. Frank said it was "elegant".

Wellllllll ..... I'm not so sure. (And since I mainly quoted Microsoft in my article, I suppose I can claim that I have them on my side on this.) You can decide for yourself.

Frank's Study Group provided an example with a form that has 4 labels and 2 buttons. Button 1 clears the labels and Button 2 fills them. It's a good idea to read Frank's original question again and notice that the example he used was a loop that is used to clear the Caption property of an array of Label components. Here's the VB.NET equivalent of that VB6 code. This code does what Frank originally asked for!

```
Public Class Form1
      Inherits System.Windows.Forms.Form
  #Region " Windows Form Designer generated code "
         Dim LabelArray(4) As Label
         'declare an array of labels
      Private Sub Form1_Load( _
         ByVal sender As System.Object, _
         ByVal e As System.EventArgs) _
         Handles MyBase.Load
         SetControlArray()
      End Sub


      Sub SetControlArray()
         LabelArray(1) = Label1
         LabelArray(2) = Label2
         LabelArray(3) = Label3
         LabelArray(4) = Label4
      End Sub


      Private Sub Button1_Click( _
         ByVal sender As System.Object, _
         ByVal e As System.EventArgs) _
         Handles Button1.Click
         'Button 1 Clear Array
         Dim a As Integer
         For a = 1 To 4
            LabelArray(a).Text = ""
         Next
      End Sub


      Private Sub Button2_Click( _
         ByVal sender As System.Object, _
         ByVal e As System.EventArgs) _
         Handles Button2.Click
         'Button 2 Fill  Array
         Dim a As Integer
         For a = 1 To 4
            LabelArray(a).Text = _
            "Control Array " & CStr(a)
         Next
      End Sub
  End Class
```

If you experiment with this code, you will discover that in addition to setting properties of the Labels, you can also call methods. So why did I (and Microsoft) go to all the trouble to build the "Ugly" code in Part I of the article?

I have to disagree that it's really a "Control Array" in the classic VB sense. The VB6 Control Array is a supported part of the VB6 syntax, not just a technique. In fact, maybe the way to describe this example is that it is an array of controls, not a Control Array.

In Part I, I complained that the Microsoft example ONLY worked at run time and not design time. You can add and delete controls from a form dynamically, but the whole thing has to

be implemented in code. You can't drag and drop controls to create them like you can in VB6. This example works mainly at design time and not at run time. You can't add and delete controls dynamically at run time. In a way, it's the complete opposite of the Part I example.

The classic VB6 control array example is the same one that is implemented in the VB .NET code. Here in VB6 code (this is taken from Mezick & Hillier, *Visual Basic 6 Certification Exam Guide*, p 206 - slightly modified, since the example in the book results in controls that can't be seen):

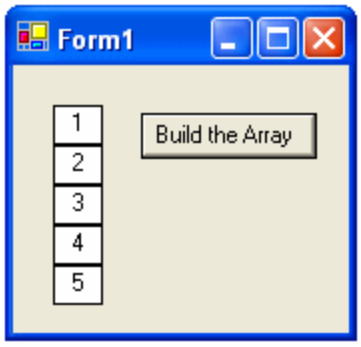
```
Dim MyTextBox as VB.TextBox
Static intNumber as Integer
intNumber = intNumber + 1
Set MyTextBox = _
  Me.Controls.Add("VB.TextBox", _
  "Text" & intNumber)
MyTextBox.Text = MyTextBox.Name
MyTextBox.Visible = True
MyTextBox.Left = _
(intNumber - 1) * 1200
```

But as Microsoft (and I) agree, VB6 control arrays aren't possible in VB.NET. So the best you can do is duplicate the functionality. My article duplicated the functionality found in the Mezick & Hillier example. The Study Group code duplicates the functionality of being able to set properties and call methods.

So the bottom line is that it really depends on what you want to do. VB.NET doesn't have the whole thing wrapped up as part of the language, but ultimately it's far more flexible.

(Part III)

Control Arrays in VB.NET don't have to be that hard!
But there are some interesting gotcha's involved!



About Visual Basic Reader John Fannon decided to take up the challenge of Control Arrays in VB.NET and came up with what he says is an easier approach. Let's see what he did.

Revision! John kept working his solution and now has new conclusions! That doesn't invalidate what he discovered and reported in this article however. His new ideas have been added at the end of this article.

John wrote:

I needed control arrays because I wanted to put a simple table of numbers on a form at run time. I didn't want the nausea of placing them all individually and I wanted to use VB.NET.

As pointed out on About Visual Basic, Microsoft offers a very detailed solution to a simple problem, but it's a very large sledgehammer to crack a very small nut. After some experimentation, I eventually hit upon a solution. Here's how I did it.

The About Visual Basic example from Part I shows how you can create a TextBox on a Form by creating an instance of the object, setting properties, and adding it to the Controls collection that is part of the Form object.

```
Dim txtDataShow As New TextBox
txtDataShow.Height = 19
txtDataShow.Width = 80
txtDataShow.Location = New Point(X, Y)
Me.Controls.Add(txtDataShow)
```

Although the Microsoft solution creates a Class, I reasoned that it would be possible to wrap all this in a subroutine instead. Every time you call this subroutine you create a new instance of the textbox on the form. Here's the complete code:

```
Public Class Form1
    Inherits System.Windows.Forms.Form
#Region " Windows Form Designer generated code "

    Private Sub BtnStart_Click( _
        ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles btnStart.Click

        Dim I As Integer
        Dim sData As String
        For I = 1 To 5
            sData = CStr(I)
            Call AddDataShow(sData, I)
        Next
    End Sub
    Sub AddDataShow( _
        ByVal sText As String, _
        ByVal I As Integer)
        Dim txtDataShow As New TextBox
        Dim UserLft, UserTop As Integer
        Dim X, Y As Integer
        UserLft = 20
        UserTop = 20
        txtDataShow.Height = 19
        txtDataShow.Width = 25
        txtDataShow.TextAlign = _
            HorizontalAlignment.Center
        txtDataShow.BorderStyle = _
```

```
                BorderStyle.FixedSingle
            txtDataShow.Text = sText
            X = UserLft
            Y = UserTop + (I - 1) * txtDataShow.Height
            txtDataShow.Location = New Point(X, Y)
            Me.Controls.Add(txtDataShow)
        End Sub
    End Class
```

Very good point, John. This is certainly a lot more simple than the Microsoft code ... so I wonder why they insisted on doing it that way?

To begin our investigation, let's try changing one of the property assignments in the code. Let's change

```
txtDataShow.Height = 19
```

to

```
txtDataShow.Height = 100
```

just to make sure that there is a noticeable difference.

When we run the code again, we get ... Whaaaat??? ... the same thing. No change at all. In fact, you can display the value with a statement like MsgBox (txtDataShow.Height) and you still get 20 as the value of the property no matter what you assign to it. Why does that happen?

The answer is that we're not deriving our own Class to create the objects, we're just adding things to another Class so we have to follow the rules of the other class. And those rules state that you can't change the Height property. (Wellllll ... you can. If you change the Multiline property to True, then you can change the Height.)

Why VB.NET goes ahead and executes the code without even a whimper that there might be something wrong when, in fact, it totally disregards your statement is a whole 'nother gripe. I might suggest at least a warning in the compile, however. (Hint! Hint! Hint! Is Microsoft listening?)

The example from Part I inherits from another Class, and this makes the properties available to the code in the inheriting Class. Changing the Height property to 100 in this example gives us the expected results. (Again ... one disclaimer: When a new instance of a large Label component is created, it covers up the old one. To actually see the new Label components, you have to add the method call aLabel.BringToFront().)

This simple example shows that, although we CAN simply add objects to another Class (and sometimes this is the right thing to do), programming control over the objects requires that we derive them in a Class and the most organized way (dare I say, "the .NET way" ??) is to create properties and methods in the new derived Class to change things. John remained

unconvinced at first. He said that his new approach suits his purpose even though there are limitations from not being "COO" (Correctly Object Oriented). More recently, however, John wrote,

" ... after writing a set of 5 textboxes at runtime, I wanted to update the data in a subsequent part of the program - but nothing changed - the original data was still there.

(DJM Note: RIGHT! That's because these inherited properties can't be changed at runtime!)

I found that I could get round the problem by writing code to take off the old boxes and putting them back again with new data. A better way to do it would be to use Me.Refresh. But this problem has drawn my attention for the need to supply a method to subtract the textboxes as well as add them."

John then enclosed code which used a global variable to keep track of how many controls had been added to the form so a method ...

```
Private Sub Form1_Load( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs) _
   Handles MyBase.Load
   CntlCnt0 = Me.Controls.Count
End Sub
```

Then the "last" control could be removed ...

```
N = Me.Controls.Count - 1
Me.Controls.RemoveAt(N)
```

John noted that, "maybe this is a bit clumsy."

Oh! I dunno about that, John. It's the way Microsoft keeps track of objects in COM *AND* in their "ugly" example code in Part I. I think it's a wonderful example that actually puts the last ten years of OOP thinking at Microsoft in a capsule view. In any case, John's latest conclusion is ...

I've now returned to the problem of dynamically creating controls on a form at run time and I have been looking again at the 'What Happened to Control Arrays' articles.

I have created the classes and can now place the controls onto the form in the way I want them to be.

John even enclosed code which demonstrated how to control the placement of controls in a group box using the new classes he has started using. Maybe Microsoft had it right in their "ugly" solution after all!

(Part IV)

Selden McCabe adds his version to the series

Controls Collections!

It seems to be a subject that has the power to attract a lot of attention! Another About Visual Basic Reader, Selden McCabe has written a new article that adds to everything that has been said before in Part I, Part II, and Part III of what is now a continuing series. Selden's article extends the ideas in Part I because he goes back to the concept of defining a new class to manage the controls on a form.

Selden adds a new feature to the ideas in Part I, however. But I'll let him describe it in his own words in a few paragraphs. Let's hear it from Selden now.

I found control arrays to be very useful when programming in VB6, and really missed them when I moved to .NET. But it turns out that the object-oriented features in .NET allow us to do just about anything, including replacing the missing control arrays.

VB.NET forms do have a controls collection. But one of the first problems I ran into when trying to use it was that the controls are not all children of the Form. Controls that exist on a Tab Control, for example, are children of the Tab Control, not of the Form. So if you look in a Form's controls collection, you won't find all the controls, if they exist on a frame, tab, etc.

Fortunately, all controls have a HasChildren property. This property returns a True if the control has any child controls in it.

So using the form's controls collection, plus the HasChildren property, and recursion (see VB.NET and Recursion at VisualBasic.About.com for an explanation of what recursion is), we can create a collection of all controls on a form with just a few lines of code.

Here is a class that does it:

```
Public Class ControlsCollection
    Private Shared m_controls As Collection
    Public Sub New(ByVal myForm As Form)
        m_controls = New Collection
        'create a control walker to get
        'all controls on the form
        Dim aControlWalker As New ControlWalker(myForm)
    End Sub
    'This property returns the collection of all controls
    'on the form
    ReadOnly Property Controls() As Collection
        Get
            Return m_controls


        End Get
    End Property
    Private Class ControlWalker
        ' This class recursively walks through all controls
        ' in a container, and all containers contained in
        ' this container, visiting all controls throughout
        ' the hierarchy
        Private mContainer As Object
        Public Sub New(ByVal Container As Object)
            Dim cControl As Control
            If Container.haschildren Then
                For Each cControl In Container.controls
                    'add this control to the controls collection
                    m_controls.Add(cControl)
                    If cControl.HasChildren Then
                        'This control has children, create another
                        'ControlWalk go visit each of them
                        Dim cWalker As New ControlWalker(cControl)
                    End If
                Next cControl
            End If
        End Sub
    End Class
End Class
```

The class ControlsCollection contains a collection (m_controls) and an internal private class named ControlWalker. The ControlWalker class does all the work.

When a new instance of the ControlWalker is created, it is handed a reference to the form. This new instance then looks at each control on the form, adding it to the m_controls collection. Then, if this control has children, a new instance of ControlWalker is created and handed this control as the container. This new ControlWalker goes through all the controls in that container, adding them to the m_controls collection.

As the new ControlWalker goes through the controls in the container, it looks to see if any of them has children. If so, it creates a new instance of the ControlWalker, handing it a reference to this container.

This process continues until no more controls and containers are found.

At this point, the ControlsCollection class has a collection, m_controls, which contains a reference to every control on the original form, no matter how deeply nested within other controls.

To add the ControlClass to your form, add a private instance of it, construct it in the Form_Load, and add a ReadOnly property to expose it's Controls collection from the form, as shown in the following code:

```
Private m_ControlsCollection As ControlsCollection
Private Sub Form1_Load( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles MyBase.Load
    'create the controlscollection class
    m_ControlsCollection = New ControlsCollection(Me)
End Sub



Public ReadOnly Property MyControls() As Collection
    Get
        Return m_ControlsCollection.Controls
    End Get
End Property
```

An example of how you could access the Controls collection from Form1 would look ike this:

```
Dim fTest As New Form1
fTest.Show()
Dim aControl As Control
For Each aControl In fTest.MyControls
    Debug.WriteLine(aControl.Name)
Next aControl
```

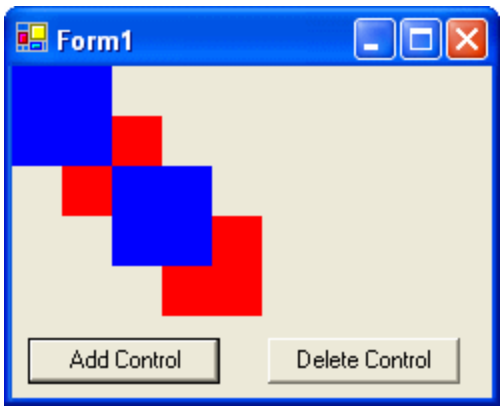
Any time you want to have a Controls collection for a Form, you can add the ControlsCollection class to the form, and your Form now has a MyControls property.

I haven't explored Selden's ideas as much as I'd like to. There are some interesting and sometimes unexpected things that can happen. For example, I discovered that it's possible to put the VB.NET compiler *itself* into an infinite recursion by trying to include the last segment of Selden's code into the Form1 Load event.

That was fun!

(Part V)

William Benton shows us YET ANOTHER WAY to do Control Arrays IN VB.NET



An About Visual Basic Reader, has created his own unique solution to a problem that has attracted more attention than any other single issue here. In four previous articles, we have looked at various ways to provide the familiar - and useful - control array that VB6 made available, but which is totally missing in VB.NET. Check out the Related Resources links to read the other articles. William's solution uses real arrays, not a collection like Selden McCabe used in part 4 of this series.

William writes that he looked, "for a more object oriented solution to the control array issue and I cracked it. Through the creation of a class, and the use of inheritance, I found that I can do everything I want to do, and still understand how it's happening!"

William inherits from the PictureBox control for this example. His class includes two properties and an event. One of the properties is used to index the array in the program that instantiates the class. The other is used to keep track of whether a PictureBoxClass object is "flipped" or not. "Flip" is just a change in color, but something like this could work very well in a card game program.

The event combines the Click and DoubleClick events so you get the same result for either one. But it's a great example showing how to code an event in a class.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Public Class PictureBoxClass
  Inherits PictureBox
' used as a ID within the array
  Private I As Integer
' flips the control color
  Private F As Boolean
  Public Event PicBoxClick(ByVal ID)

    Public Property ID() As Integer
    Get
      Return I
    End Get
    Set(ByVal Value As Integer)
      I = Value
    End Set
  End Property

  Public Property FaceUp() As Boolean
    Get
      Return F
    End Get
    Set(ByVal Value As Boolean)
      F = Value
    End Set
  End Property

' To make "Click" and "DoubleClick" act the same way
```

```
    Public Sub Picturebox_Click( _
      ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
      Handles MyBase.Click, _
      MyBase.DoubleClick
      RaiseEvent PicBoxClick(ID)
    End Sub
  End Class
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

To demonstrate William's solution, you first need a form with two buttons. One adds controls, the other removes them.

William's example starts out by adding four instances of the class. Each instance is saved in an array indexed by the ID variable. ID is one of the properties of each instance of the class.

The Dim for PB(0) creates the "control array" that will store our unique PictureBoxClass objects. The PBEmpty variable is just a "one time initialization flag" used to start things out right.

Here's the code to start William's demonstration program:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Dim PB(0) As PictureBoxClass
Dim PBEmpty As Boolean = True
Private Sub Form1_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load
  For a As Integer = 1 To 4
    Call AddToPB()
  Next a
End Sub
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

The code that adds and deletes instances of the class from the array is as critical as the class coding. In fact, it seems to me that at least half of the magic happens in this part of the program.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Private Sub AddToPB()
  Dim a As Integer
  If PBEmpty = False Then
    a = PB.Length - 1
    Dim temp(a) As PictureBoxClass
    PB.CopyTo(temp, 0)
    a += 1
    ReDim PB(a)
    temp.CopyTo(PB, 0)
  Else
    a = 0
    PBEmpty = False
  End If
  PB(a) = New PictureBoxClass
  With PB(a)
    .ID = a
    .Height = 50
    .Width = 50
```

```
    .FaceUp = False
    .BackColor = Color.Red
    .Top = PB(a).Height * a / 2
    .Left = PB(a).Width * a / 2
    .Visible = True
    .BringToFront()
    AddHandler PB(a).PicBoxClick, _
      AddressOf FlipPB
    Me.Controls.Add(PB(a))
  End With
End Sub
~~~~~~~~~~~~~~~~~~~~~~~~~~
```

The Else clause on PBEmpty is only executed when the program starts up, mainly to prevent the other code from executing the first time.

When a new PictureBoxClass instance is created, it's saved in the array and then all of the properties, including the two new ones added to the inherited PictureBox class, are set. Then FlipPB is added as a Handler for the current element of the PB array - an instance of the class - and it's added to the controls collection of Me - which is Form1.

Here's the routine to delete.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~
Private Sub RemoveFromPB()
  If PB.Length > 0 Then
    Dim a As Integer = PB.Length - 1
    Dim temp(a) As PictureBoxClass
    PB.CopyTo(temp, 0)
    ReDim PB(a - 1)
    temp.Copy(temp, PB, a)
    Me.Controls.Remove(temp(a))
    temp(a).Dispose()
  End If
End Sub
~~~~~~~~~~~~~~~~~~~~~~~~~~
```

These routines work by:

- copying all of the control array to a temporary array
- Adjusting the length of the control array (to both add and delete)
- copying the temporary array back to the control array

In the RemoveFromPB routine, Copy had to be used rather than CopyTo because the temp array (the source) is one longer than the adjusted length of PB (the destination) and Copy has a length parameter.

Of the three final subroutines, only the first is interesting. (The last two are just the click events that call the add and delete subroutines.) The FlipPB subroutine isn't complex either. It just checks the value of the FaceUp property of the class, reverses it, and sets the color of the control accordingly.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~
Private Sub FlipPB(ByVal ID)
  PB(ID).FaceUp = Not PB(ID).FaceUp
```

```
  Select Case PB(ID).FaceUp
    Case True
    PB(ID).BackColor = Color.Blue
  Case False
    PB(ID).BackColor = Color.Red
  End Select
  PB(ID).BringToFront()
End Sub

Private Sub Button1_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles Button1.Click
  Call AddToPB()
End Sub

Private Sub Button2_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles Button2.Click
  Call RemoveFromPB()
End Sub
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

This method is an excellent study for learning about events, handlers, properties, classes and so forth, but copying of entire object arrays back and forth can cost a lot of CPU time, especially if you have a significant number of controls in the arrays. So this may not be the method of choice for all your programs. But it's a great piece of work anyway.

Thanks William!

(Part VI)

Patrick Krawec's solution must be "best of class"!

Patrick Krawec, one of the intrepid crew of programmers at Inovision Software Solutions has created a solution that *must* be "best of class" in several ways.

- Once the class is included in your project, it's really easy to use.
- It's an excellent example showing how to pass objects and collections to a class so that the involved details can be taken care of in the class. The logic itself is pretty straightforward and obvious. The basic idea is for the class to search the form for all controls that fit a common naming convention and stick them in an **ArrayList**. (An ArrayList is an interesting new class in .NET. Microsoft documents it this way: "An ArrayList is a sophisticated version of an array. The ArrayList class provides some features that are offered in most Collections classes but are not in the Array class.")

But possibly most important ...

- It actually does create something that does have many of the characteristics a good old VB6 control array! But, like every solution proposed so far (including Microsoft's - the original Part I article), it does have some problems.

Here's how Patrick explains his solution:

I implemented a very simple solution which links Controls on a form to the code behind it as a control array, much like VB6.

I have come across the same problem as others. I have many screens with many control arrays that we created in VB6.

I created a way to do this in the application code which requires only one function call. The call returns an array of the type of control desired. (Of course you need to include a helper class in the project, or compile it as a Class Library.)

The key is: You have to name your controls on the screen with a standard convention, so the function can find them.

Example: myTextBox1, myTextBox2, myTextBox3. This will fill the array with the three controls in positions 1, 2, and 3.

A sample of the application code is:

```
Dim textBoxes as TextBox() = getControlArray(me,"myTextBox")
```

Thats it! From that point the textBoxes array is very similar to a VB6 control array and the function works with any type of control.

Patrick's solution is a variation on the theme of previous solutions. Like Microsoft and several previous contributors, Patrick codes a class to do most of the work. In Part III, I complain that because the code doesn't inherit the class, things like changing the Height property of a control don't work. That's also the case here. William Benton, in Part V, shows how to do a VB.NET control array with an inherited class.

Selden McCabe, in Part IV of this series, pointed out that controls which aren't direct children of the form, such as Tabcontrol, won't be handled with this type of solution. Combining Selden's solution with Patrick's will be left as an exercise for the reader!

The great advantage of Patrick's solution is that it's easy to add and delete controls at design time. All you have to do is make sure they are named correctly. And gaps in the sequence are OK too. There are five Label and Textbox controls that are all named Mixed with a number suffix in a controls( ) collection in Patrick's example, I added a new Button control and named it Mixed29. The code ...

```
controls(29).Text = "About Visual Basic"
```

... worked perfectly!

Code download

http://visualbasic.about.com/library/download/ControlArray.zip

(Part VII)

Clif Gay's compatibility library based solution

Clif Gay, an Engineering Specialist / Programmer in the R&D Engineering department of Anaren, Inc. has sent in yet another way to do it. Explaining his idea is a great way to review a part of VB.NET that hasn't been discussed much recently, the Compatibility.VB6 namespace in VB.NET. And Clif's idea also provides a great reason to discuss some of the changes that have been made in Visual Studio 2005.

One of the big advances in .NET is that all of the code that is in your application is now open to you to see. In VS 1.X, this is in the hidden "Region" code. In VS 2.0, this is in the *component name*.Designer.vb file that is hidden by default in the Solution Explorer window. Click the Show All Files button to display it. Clif wrote that he discovered his idea by studying this hidden code that Visual Studio generates - a clear demonstration of the value of having it available rather than buried in the system code as it is in VB6.

In fact, I suggested that this could be tried way back in the very first article I wrote about simulating VB6 control arrays way back in the very first article I wrote about it:

"If you reference the VB6 compatibility library, there are objects in there that act pretty much like control arrays. To see what I mean, simply use the VB.NET upgrade wizard with a program that contains a control array." Clif went above and beyond and figured out how to use this code.

Microsoft documented a different way to do the job. But, due to changes in the way Visual Studio 2005 and VB 2005 Express now manage generated code, their solution doesn't work too well today. Both Clif's idea and the Microsoft recommendation require a modification to Visual Studio generated code. But today, VS 2005 overwrites any changes you make whenever a change is made to the form. If the link above stops working, it's probably because Microsoft has removed the article from MSDN when they figured out that following their advice creates problems.

Clif has figured out how to overcome the problem and adds some new techniques that have not been explored before. Let's see how Clif's idea works.

ps ... This article was developed and tested with Visual Basic 2005 Express Edition. Clif uses "top of the line" Visual Studio 2005 Team Edition. So between us, we've got you covered.

VB6 is fading far enough in the past to make it worthwhile to review the feature we're trying to duplicate. Here's what the Microsoft VB6 documentation says about control arrays:

A control array is a group of controls that share the same name and type. They also share the same event procedures. A control array has at least one element and can grow to as many elements as your system resources and memory permit ... Elements of the same control array have their own property settings. Common uses for control arrays include menu controls and option button groupings. Visual Basic includes the ability to dynamically add unreferenced controls to the Controls collection at run time.

To illustrate how to use the Compatibility.VB6 namespace in VB.NET, let's take the suggestion of the first article and actually upgrade one. Microsoft's documentation tells you how to create a dynamic control array in VB6:

You can add and remove controls in a control array at run time using the Load and Unload statements. However, the control to be added must be an element of an existing control array. You must have

created a control at design time with the Index property set, in most cases, to 0. Then, at run time, use this syntax:

Load object(index%)

Unload object(index%)

Using Microsoft's instructions, I created this VB6 program:

```
Dim ControlIndex As Integer
Private Sub Form_Load()
  ControlIndex = 0
End Sub
Private Sub AddLabel_Click()
  ControlIndex = ControlIndex + 1
  Load LabelControlArray(ControlIndex)
  LabelControlArray(ControlIndex).Visible = True
  LabelControlArray(ControlIndex).Top = _
    LabelControlArray(ControlIndex - 1).Top + 300
End Sub
Private Sub DelLabel_Click()
  If ControlIndex > 0 Then
    Unload LabelControlArray(ControlIndex)
    ControlIndex = ControlIndex - 1
  End If
End Sub
```

Opening the VB6 project with VB.NET automatically calls the VB.NET Upgrade Wizard and the project coverts with zero errors.

The actual code for the form and the two buttons is almost unchanged in the converted project. Only the substitution of Short for Integer and the VB6.PixelsToTwipsY conversion seem to be present. The real changes show up in the normally hidden Form1.Designer.vb. (Remember that Show All Files has to be selected to see most of what we discuss from this point on.) Let's see what they are.

First, note that the Microsoft.VisualBasic.Compatibility namespace has been added to the references in Solution Explorer.

Now check the code in Form1.Designer.vb. To set up the control array, VB.NET adds these statements to the Form Designer generated code (line continuations added here):

```
Public WithEvents _LabelControlArray_0 _
    As System.Windows.Forms.Label
Public WithEvents LabelControlArray _
    As Microsoft.VisualBasic.Compatibility.VB6.LabelArray
```

In the InitializeComponent() sub, these statements are found:

```
Me._LabelControlArray_0 = _
  New System.Windows.Forms.Label
Me.LabelControlArray = _
  New Microsoft.VisualBasic.Compatibility.VB6.LabelArray(Me.components)
```

```
CType(Me.LabelControlArray, _
   System.ComponentModel.ISupportInitialize).BeginInit()

.....
```

The properties of first element of the array are initialized. The initialization that makes a real difference is this one:

```
Me.LabelControlArray.SetIndex(Me._LabelControlArray_0, CType(0, Short))
```

Finally, this statement is executed:

```
CType(Me.LabelControlArray, System.ComponentModel.ISupportInitialize).EndInit()
```

ISupportInitialize is used when a component has properties that depend on each other. So you can see the BeginInit and EndInit methods at the top and bottom of the initialization.

Keep in mind that VB.NET (and VS 2005) are pretty sensitive to changes. If you change anything here and reopen the form designer, the VS and VB.NET interface may make changes that will generate errors which prevent the form designer from even being opened. Clif Gay has some advice about this later on.

The bottom line is that VB.NET *can* support control arrays, in spite of what the documentation says. The easiest way to make it happen is simply to create the code in VB6 and then convert it. But since a lot of you don't even have VB6 anymore (and Microsoft is making it impossible to buy), you need to know how to use these tools. That's where Clif Gay's advice comes in handy.

Clif Gay writes, "There are 37 array controls in .NET 2.0, and 20 in .NET 1.x. These include controls like: CheckBoxArray, CheckedListBoxArray, ColorDialogArray, ComboBoxArray, DirListBoxArray, DriveListBoxArray, FileListBoxArray, etc."

Step 1 is to add a reference to Microsoft.VisualBasic.Compatibility to your project and then add an array component to the Toolbox. Right-click the project and select Add Reference ... To add the control you're interested in (for example, LabelArray) to the Toolbox, right-click the Toolbox and select Choose Items ... or select Choose Toolbox Items ... from the Tools menu, and check LabelArray.

The critical step is to add a line of code to Form1.Designer.vb. Clif suggests adding it in the initialization code for the Label:

```
Me.myLabelArray.SetIndex(Me.myLabel, CType(0, Short))
```

This assumes that you have named the LabelArray component myLabelArray and the starting Label myLabel. Note that 0 is the value of the index.

Then close the Form designer (this saves the code - VB.NET 2003 will confirm the save) and open it again. Notice that in the properties for the label, there's a new property now:

```
Index on myLabelArray     0
```

This is enough initialization to allow the same code used in VB 6 to be reused here. (But keep in mind that pixels are much larger than twips in incrementing the Top property.)

Before you jump on board and start using these ideas wholesale, keep this statement from Microsoft in mind:

The compatibility classes should not be used for new development. The Microsoft.VisualBasic.Compatibility namespace adds a layer of complexity to your Visual Basic .NET application and introduces some minimal performance costs that could be eliminated by recoding portions of the application. In addition, the Compatibility namespace often contains many classes that wrap COM objects, and ... depending on COM objects is not as optimal as a pure managed implementation.

Fortunately, you never have to use control arrays in VB.NET.

Clif Gay was kind enough to supply an example of his own using this idea. Clif's example is a great example of some advanced coding in several other ways too, so it's worth paying attention to.

Clif's code demonstrates a ComboBox control array with five elements. Clif added code to each "Designer.vb" initialization routine like this:

```
'ComboBox0
'
Me.ComboBoxArray1.SetIndex(Me.ComboBox0, CType(0, Short))
<other initialization>


'ComboBox1
'
Me.ComboBoxArray1.SetIndex(Me.ComboBox1, CType(1, Short))
<other initialization>
```

Notice how the index value in both the CType function and the name of the control are changed in each initialization.

When the form designer in VB.NET and VBE are closed, the "Designer.vb" file is rewritten and, since the form designer considers everything to be internal code, statements are even resorted and moved. Don't expect the statements you add to even be in the same location.

The example helpfully provides two different ways of doing the same thing ... with, and without the use of the compatibility.vb6 namespace and VB.NET control arrays. So one lesson you should take away with you is that there still isn't any absolute requirement to use VB6 style control arrays. You can still do whatever you need to do without them. (That's one reason Microsoft decided not to support them in .NET.)

Code Download:

http://visualbasic.about.com/od/usingvbnet/a/library/download/ControlArray7.zip