Visual Basic 6 Collections

Simply speaking, a Visual Basic Collection is defined as a group of logically related items. Visual Basic Collections can be placed in two categories---Systems Collections and Programmer defined Collections. Regardless of the type, Visual Basic Collections are great in that they allow you to easily 'get at' each item in the Collection, and if necessary, perform the same operation on all the items in a Collection. I know this definition sounds suspiciously like the definition of an Array, and I can tell you there are many similarities. The main difference between an Array and a Collection is the 'Object Oriented' Nature of the Visual Basic Collection. Later on, we'll see that we can create a Collection in place of a one dimensional Array, with some definite advantages in terms of quickly accessing the individual items in the Collection.

Collections contain items---and these items can be values or objects---which for those of you unfamiliar with the Object Oriented nature of Visual Basic, means a pointer or a reference to a Visual Basic Object---such as a Form or a Control on a form. In fact, the two types of System Collections I'll be discussing in this article are the Forms Collection and the Controls Collection.

Collection Characteristics

Let's look at the Characteristics of a Visual Basic Collection now. In some circles, the Visual Basic Collection is considered an Object---and as such, we would expect it to possess Properties and Methods---and it does. Collections, however, don't respond to any events---they are simply repositories the for the items they hold. In terms of Properties, Collections have but one---the Count Property. Collections have three Methods---Add, Item, and Remove, although I think you'll find that the Item Method acts less like a Method than anything you've seen so far.

In terms of the Forms and Controls Collection, Visual Basic automatically takes care of 'adding' newly loaded Forms to the Forms Collection, and Controls you place on the form are automatically 'added' to the Controls Collection. Adding a Form or Control to these Collections on your own can be done, but the reasons for doing so are beyond the scope of this article. Suffice to say that in this article, I'll show you how to 'navigate' through the items of these two System Collections, and also how to access their Count Properties, but I won't showing you how to add and remove your own Forms and Controls to those Collections.

However, when it comes to creating a Collection of your own, I will cover all three methods.

The Count Property

The Count Property, as you might have guessed, represents the total number of items in the Collection. It isn't zero based or anything like that, so if the Count Property of a

Collection reads 10, then there are 10 items in the Collection.

The Add, Remove, and Item Methods

Not surprisingly, the Add Method permits you to add an item to a Collection. The Remove Method removes an item, and the Item Method allows you retrieve the value of an Item in a Collection. Some students find that if they compare the behavior (Properties and Methods) of a ListBox to a Collection, they find Collections a little easier to understand. In some ways, you can think of a Collection almost like an invisible ListBox. I should point out here that unlike the ListBox, there is no Collection method that will remove every item in a Collection---you need to do that yourself step by step using the Remove method.

Speaking of the Item Method, items in a Collection can either be referenced by their Index, which is a numeric value, or in the case of a Programmer created Control, by an optional key value specified when the Item is added to the Collection. Interestingly, System Collections are zero based---that means the first item in the Collection has an Index value of 0. Programmer created Controls are one based---that meant that the first item added to the Collection has an Index value of 1.

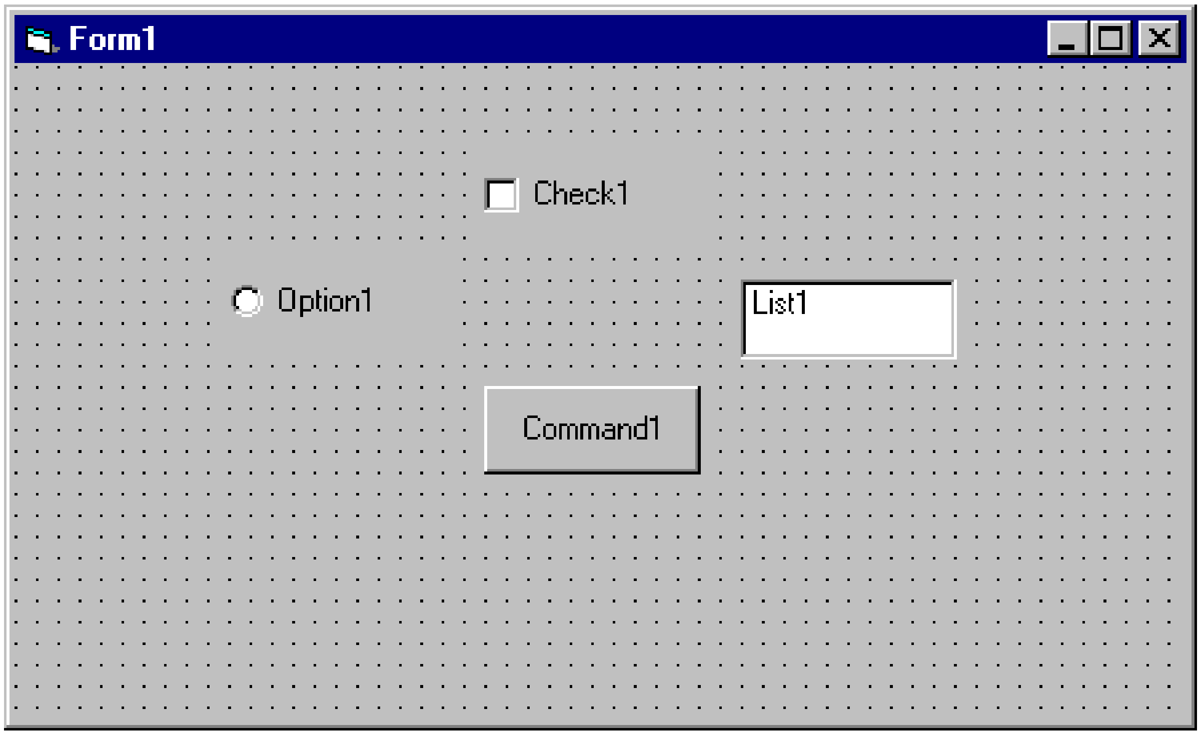
System Collections

Let's take a look at the Systems Collections in Visual Basic---in this article, we'll be examining two---the Forms Collection, and the Controls Collection. It's easiest to demonstrate the Controls Collection, so let's start with that.

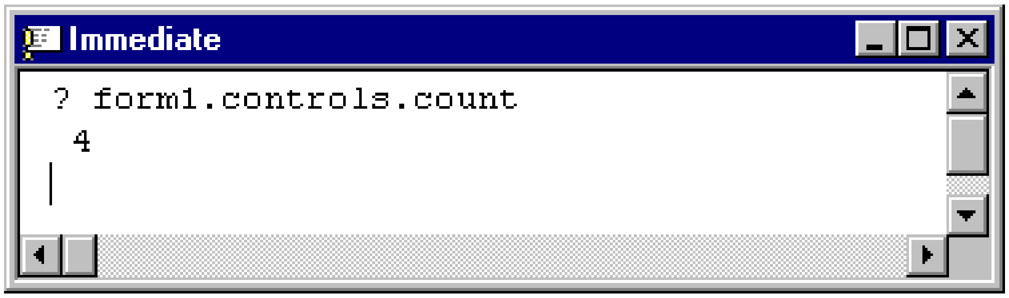
The Controls Collections

The Controls Collection is a Collection containing items where each item represents a control on a form. Each Form has its own Controls Collection, and Visual Basic automatically takes care of updating the Controls Collection for each form placed on the form.

Here's a form in which I have placed an OptionButton, CheckBox, Command Button and ListBox on the form.

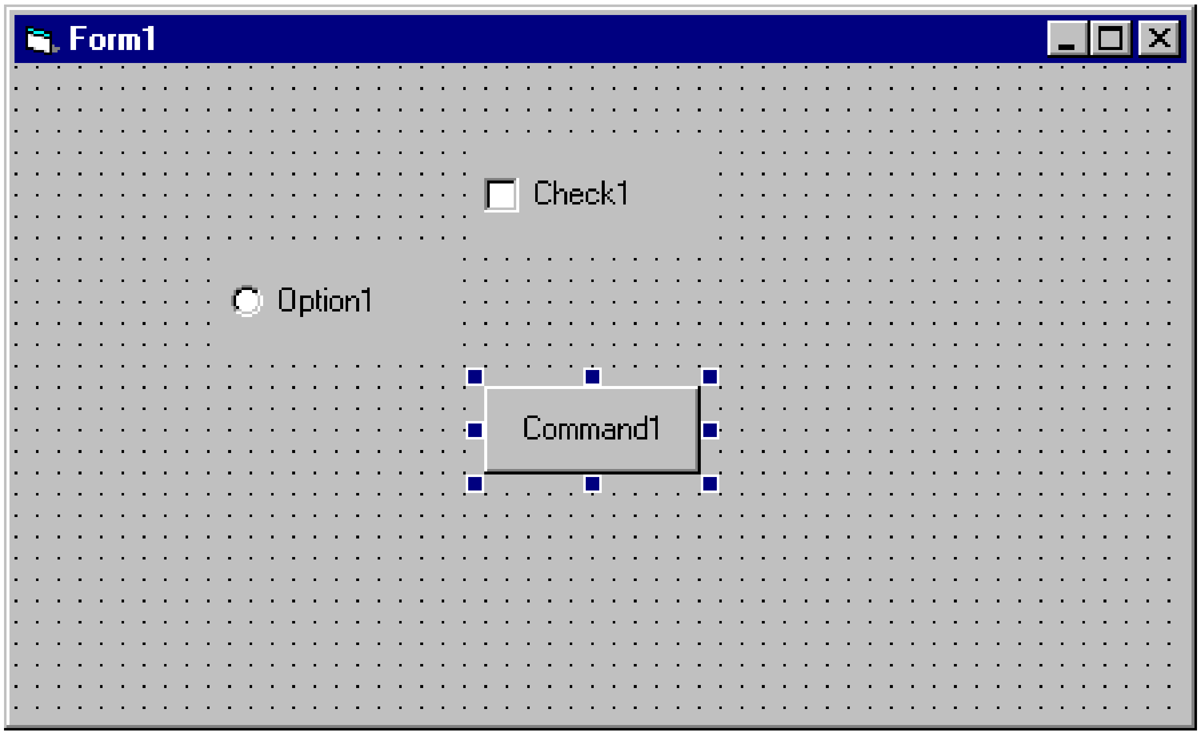


By my count, that's a total of four Controls that I've placed on the form. Let's run this program now, then pause it and use the Immediate Window to examine the Controls Collection by typing the following statement into the Immediate Window.

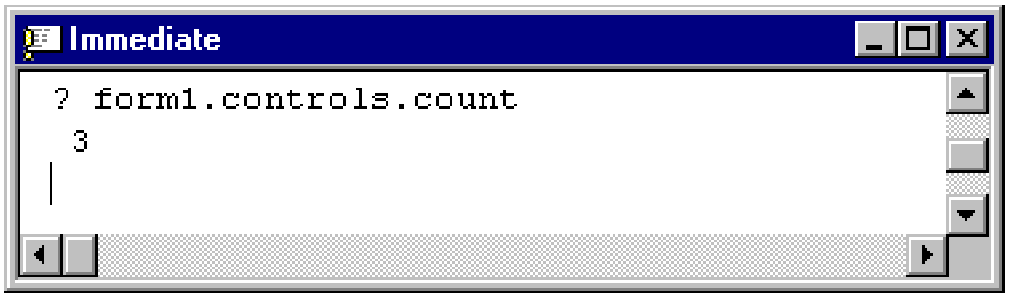


```
? form1.controls.count
 4
```

Count is the one and only property of the Collection Object, and as you can see, this Collection---the Controls Collection---has a total of four items. Let me delete one of the controls---how about the ListBox---and see what happens.



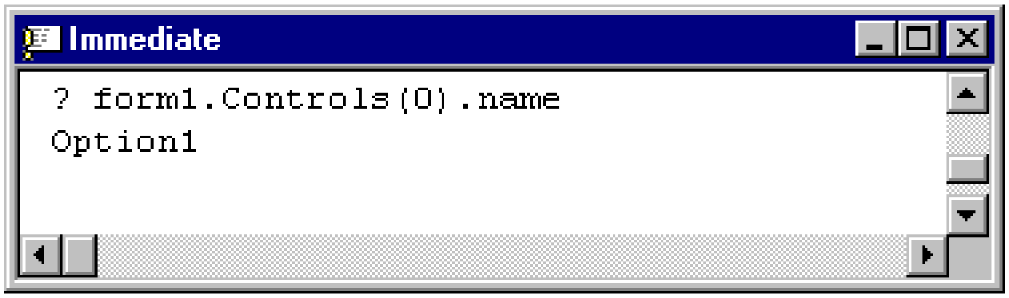
Now the form contains just three controls---if I now type the following statement into the Immediate Window….



```
? form1.controls.count
 3
```

Now the Count property of the Controls Collection reads three---we're now down to just three controls.

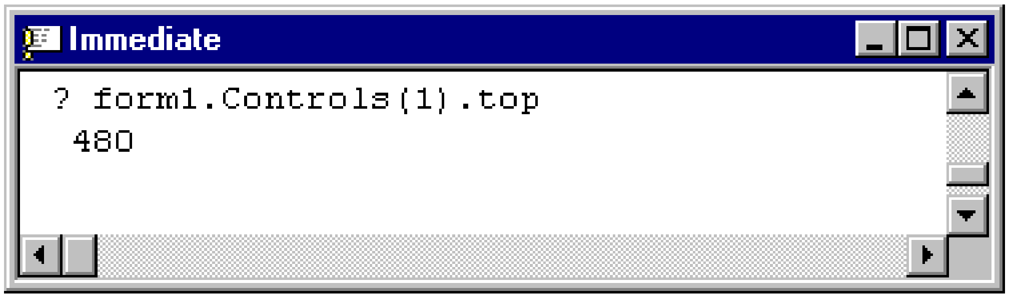
I mentioned earlier that an item added to a Collection has a unique Index value--just like an array elements with which you should be familiar. Watch this…



```
? form1.Controls(0).name
Option1
```

As you can see, we can reference an individual item within the Collection by using its Index. And if you append a Control Property to the end of that combination---in this case the Name property---you can determine that Property's value. In this case, Controls(0) refers to the first item in the Collection, the Option button. I must caution you though not to draw any conclusions about the order of the items within the Controls Collection. In this case, the Option Button happens to be the first control I placed on the form---but Microsoft warns you that Visual Basic can 'shuffle' the items in the Controls Collection at any time. So if we another control to this form, the new control may wind up being placed in the first position of the Collection.

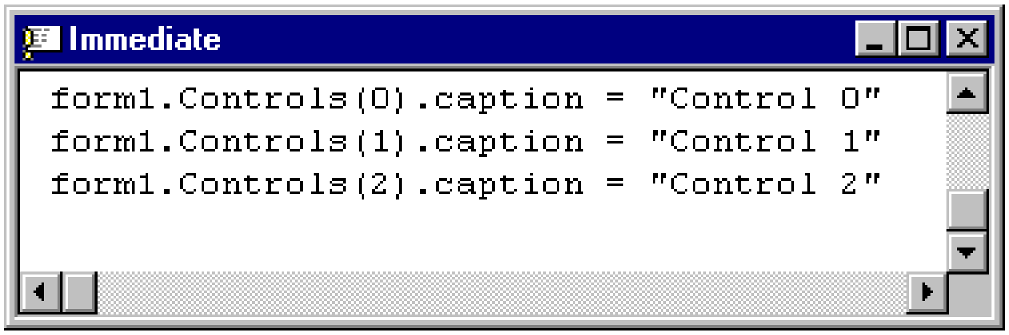
Here's another example of how we can use the Controls Collection to interrogate the Properties of a Control with the Collection…



```
? form1.Controls(1).top
 480
```

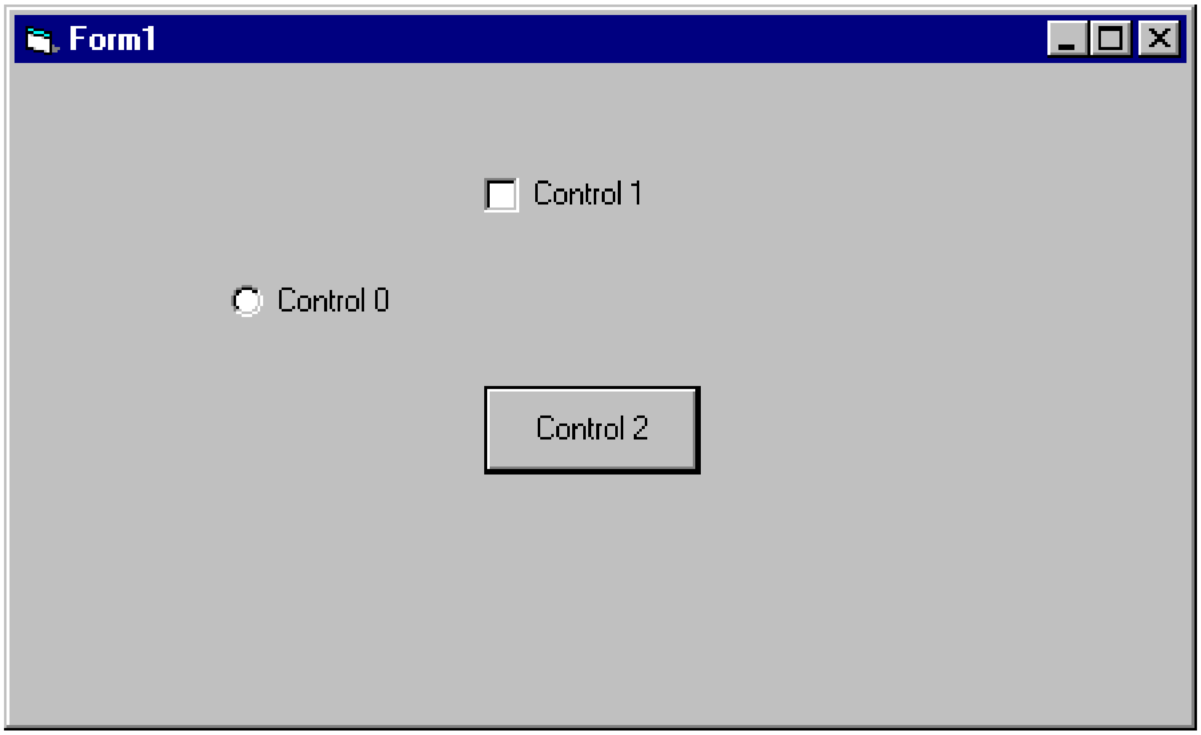
In this case, Visual Basic is telling us that the Top Property of the second item in the Controls Collection is 480--but to determine what that Control is we would need to display its Name Property.

In addition to retrieving values of Properties this way, it's also possible to change Properties using the Controls Collection. For instance, look at this …



```
form1.Controls(0).caption = "Control 0"
form1.Controls(1).caption = "Control 1"
form1.Controls(2).caption = "Control 2"
```

In this instance, I'm changing the Caption properties of every item in the Controls Collection, and the effects are immediate …

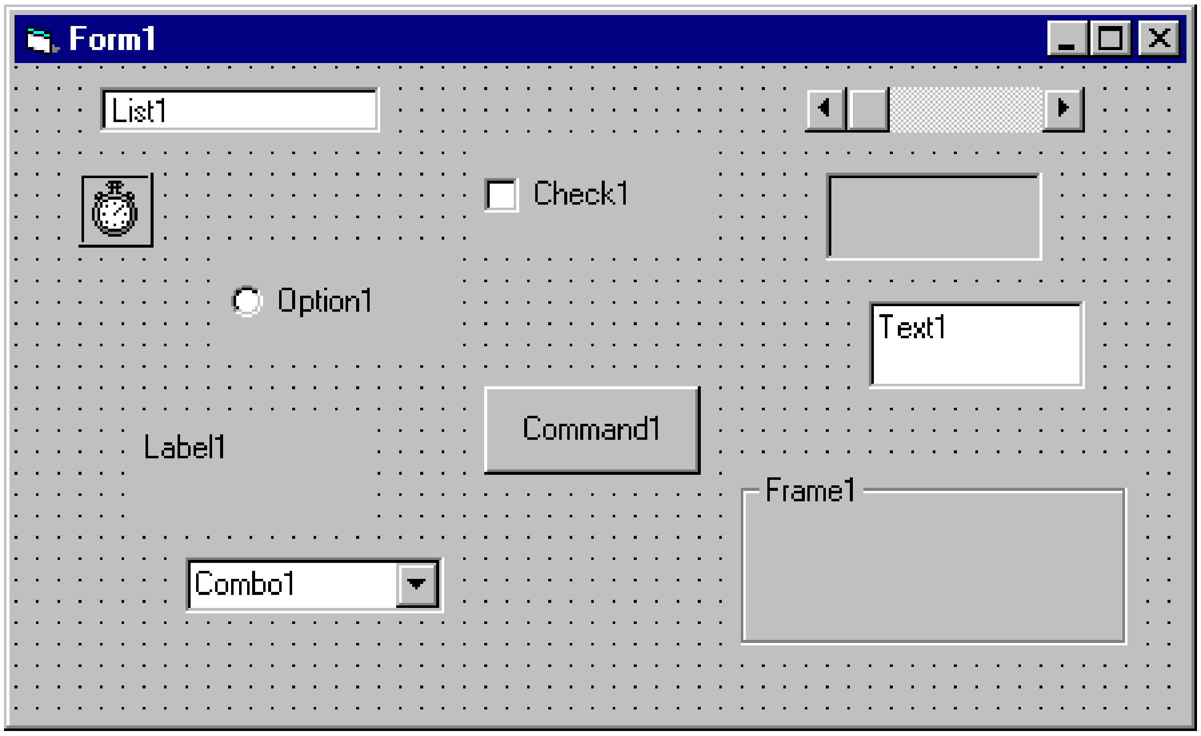


As you can see, using the Controls Collection can be quite powerful. In addition to doing something like this using the Immediate Window, I can also modify Control Properties at run time. This makes using the Controls Collection quite powerful---it can be used to change the BackColor Property of every control on a form for instance---without the programmer needing to know the names of the individual controls ahead of time.

The For…Each Statement

You're probably thinking at this point that you don't see what the hype is about---the last thing you want to do is code an individual statement for each item in the Collection---besides, who needs a Collection to do this, you could just code the Property assignment statements individually anyway, using the Control name instead of the Controls Collection index value.

Fortunately, it's possible to 'loop' through each item in a Collection using the Visual Basic For…Each Statement, which eliminate the need to code an individual statement for each item in a Collection. You can avoid this by declaring something called a Visual Basic Object Variable. Here, let's complicate our form a little bit by adding a few more controls, and I'll show you an Object Variable, and the For…Each statement in action.

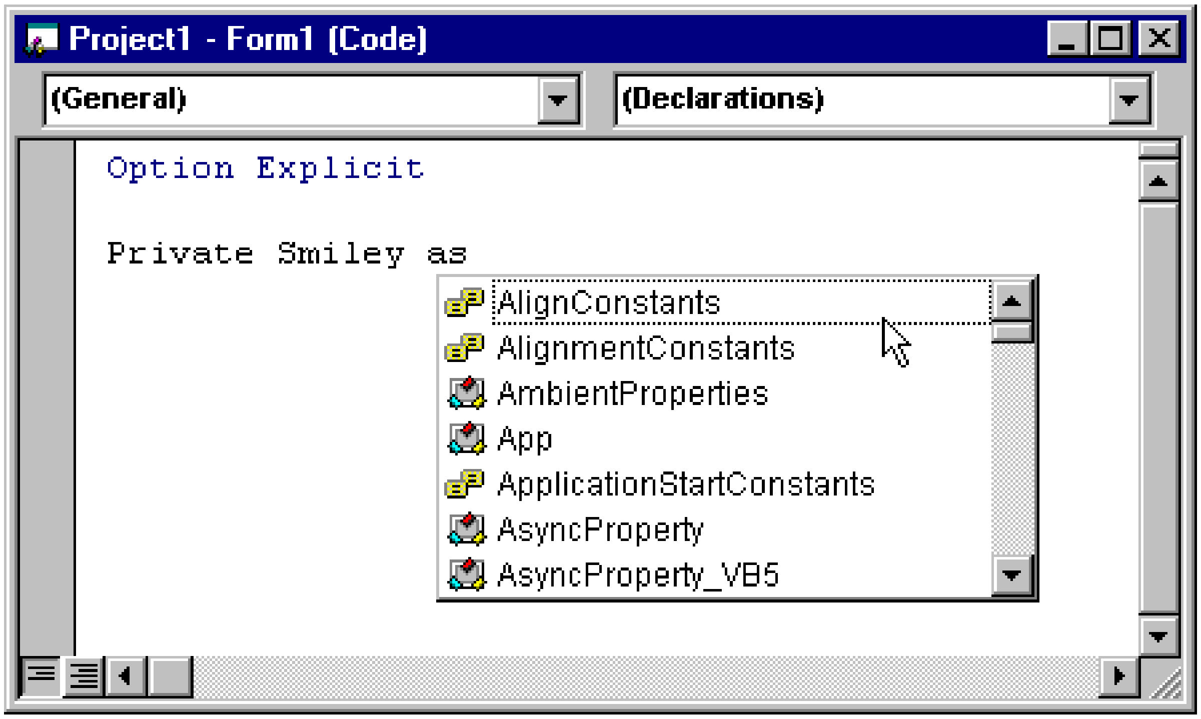


As you can see, I've added a number of new controls.

Object Variables

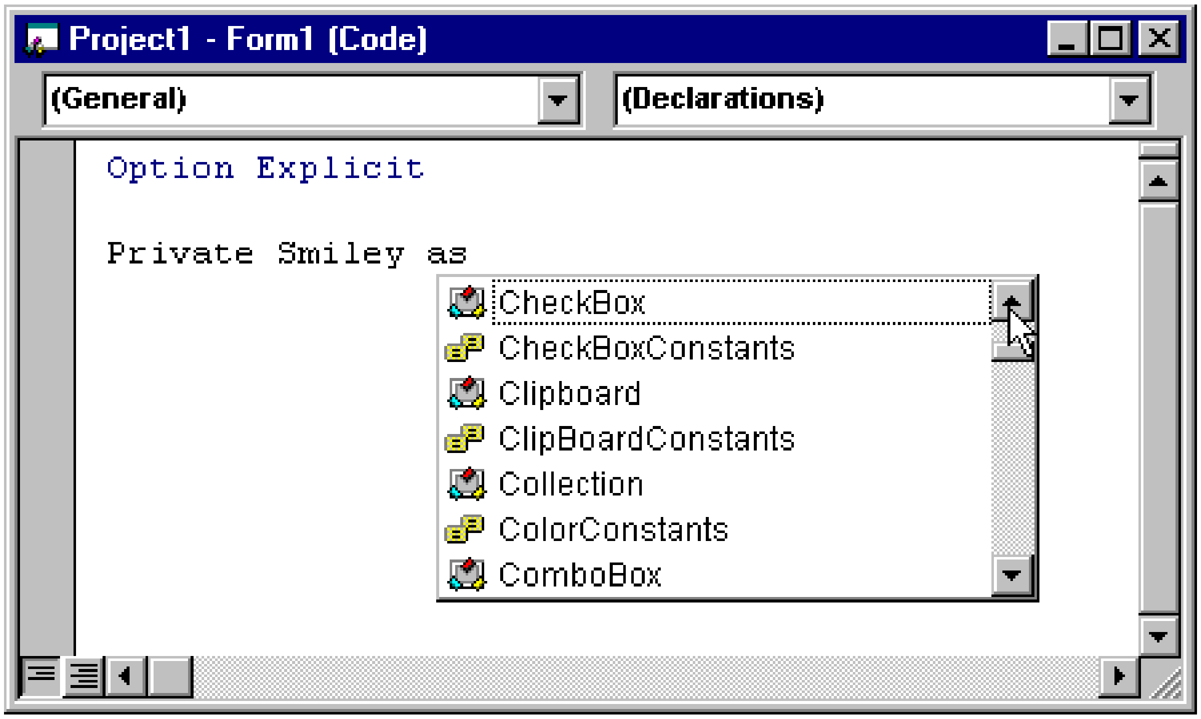
Now let's get back to the term Object Variables.

As I mentioned just a few lines back, one of the great things about Collections in Visual Basic is the ability to 'loop' through them, in much the same way you would with an array. I just showed you how you can refer to individual items within a Collection by referring to the Index value of the item. To be able to do that within a For…Each loop, you need to declare and use something called an Object variable. Unlike other types of Variables, an Object Variable is more like a pointer or a reference to an object---in this case, to an item in a Collection. Here, take a look at this. If I begin to type a Variable Declaration into the General Declarations Section of a form, as soon as I type the word 'as' Visual Basic will display the various types of Variables that I am permitted to declare.



```
Option Explicit

Private Smiley as
```

I don't know whether you've ever noticed, but there are a bunch of Variable types besides the common Integer, String, Single and Double that you may be familiar with. Many of these variable types are beyond the scope of this article, but if we scroll down through the list of possible types, you'll see that there are some interesting variable types that you may not have known existed. For instance…



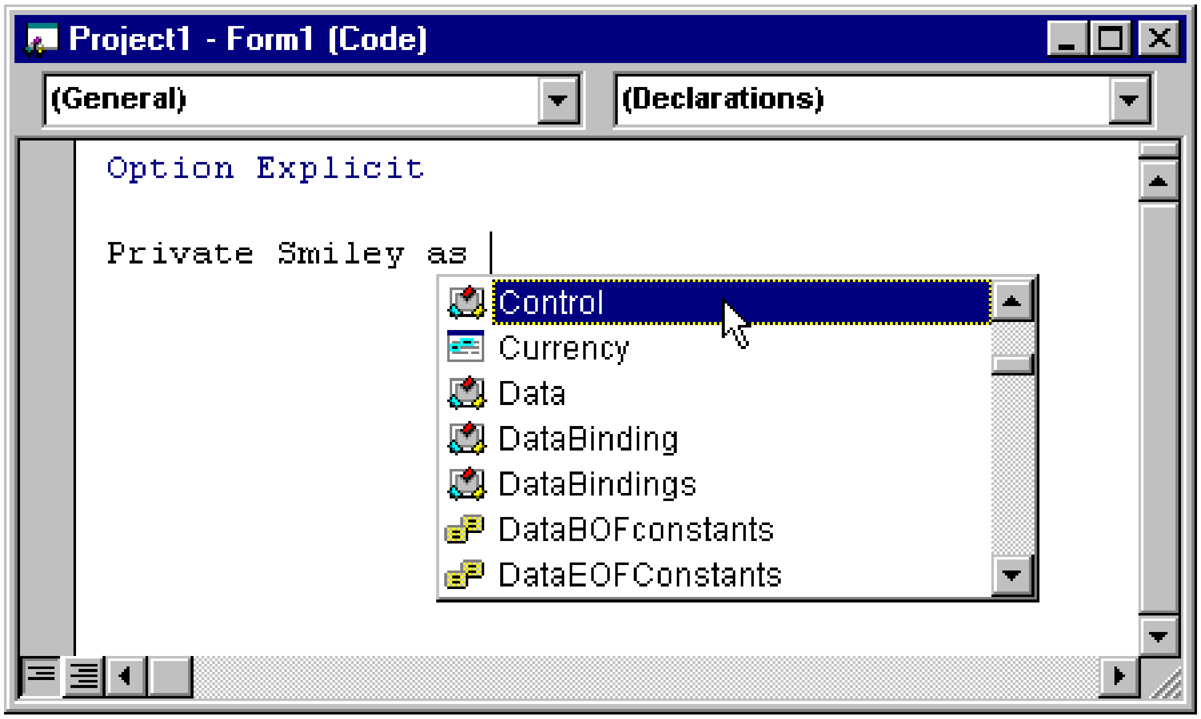
```
Option Explicit

Private Smiley as
```

The Checkbox!

This may surprise you that a Checkbox is a possible Variable Type, but it can be used as a Variable type to point to an actual Checkbox on the form. Notice that the symbol next to the

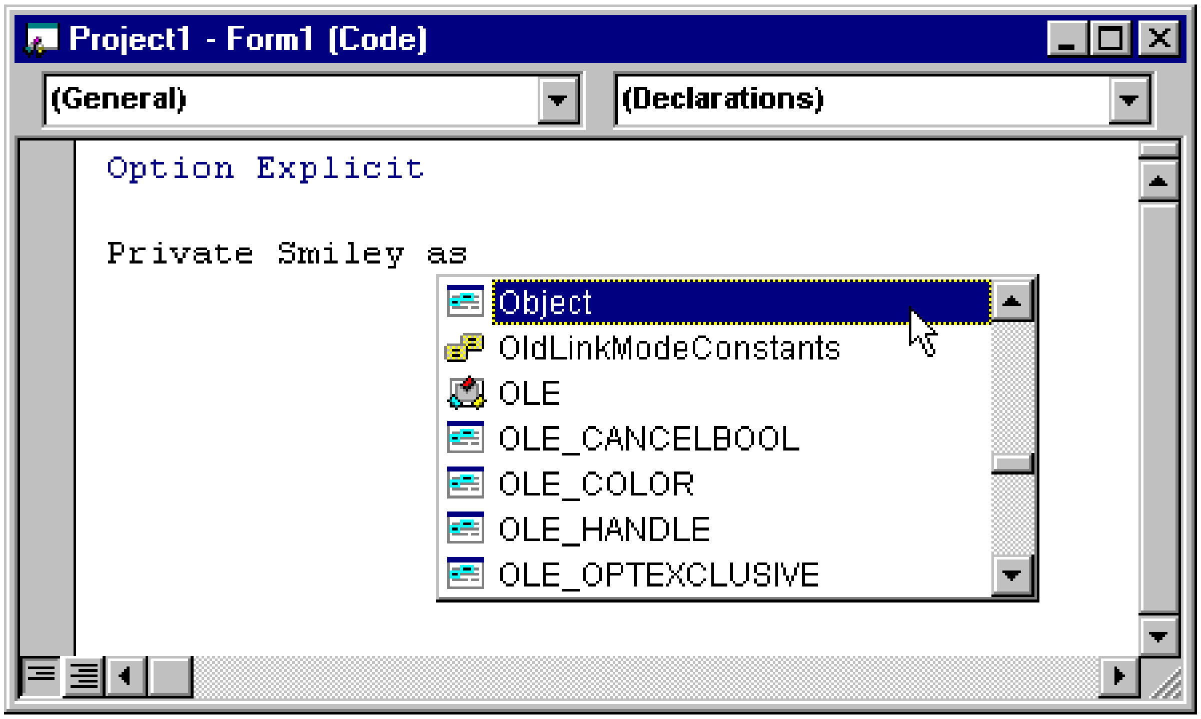
Checkbox in the ListBox is different than the symbol for the conventional Variable types you've learned about. The symbol you see next to the CheckBox is the Visual Basic symbol for an Object or Class---something you'll learn about when you start to create your own Visual Basic Class Modules and Classes.

I should also mention that there is a more generic Object type which can be used to point or reference any Control---and that's the Control variable type…



```
Option Explicit

Private Smiley as
```

Declaring a variable of type Control allows your variable to reference or point to any type of control. Even more generic is the variable type Object…



```
Option Explicit

Private Smiley as
```

Interestingly enough, an Object Variable can also be declared as a Variant.

You might be wondering if there is a preference for the type of an Object Variable? That is, should you declare everything as an Object type, or be more specific, such as CommandButton, OptionButton, or Textbox.

The answer to that question is similar to the answer about declaring Data Variables as Variants? The answer in that case is to avoid declaring a variable as a Variant if you know the type of data that the variable will hold.

The same answer applies here in a slightly modified version. Avoid declaring an Object Variable as type Object (or Variant). Declare your variable as specifically as possible. For instance, if you know you are declaring an Object variable to reference only Command Buttons, declare the Object Variable as a CommandButton variable type. If you know the variable will reference or point to any number of various controls on a form, then you have no choice but to declare it as a Control type, which is better than declaring it as an Object, and better than declaring it as a Variant. If however, you know that the variable may reference both controls on the form and the form itself, you will need to declare the Object variable as an Object type."

Early and Late Binding

I'd be remiss if I didn't mention the terms Early Binding and Late Binding as they are applied to Object Variables.

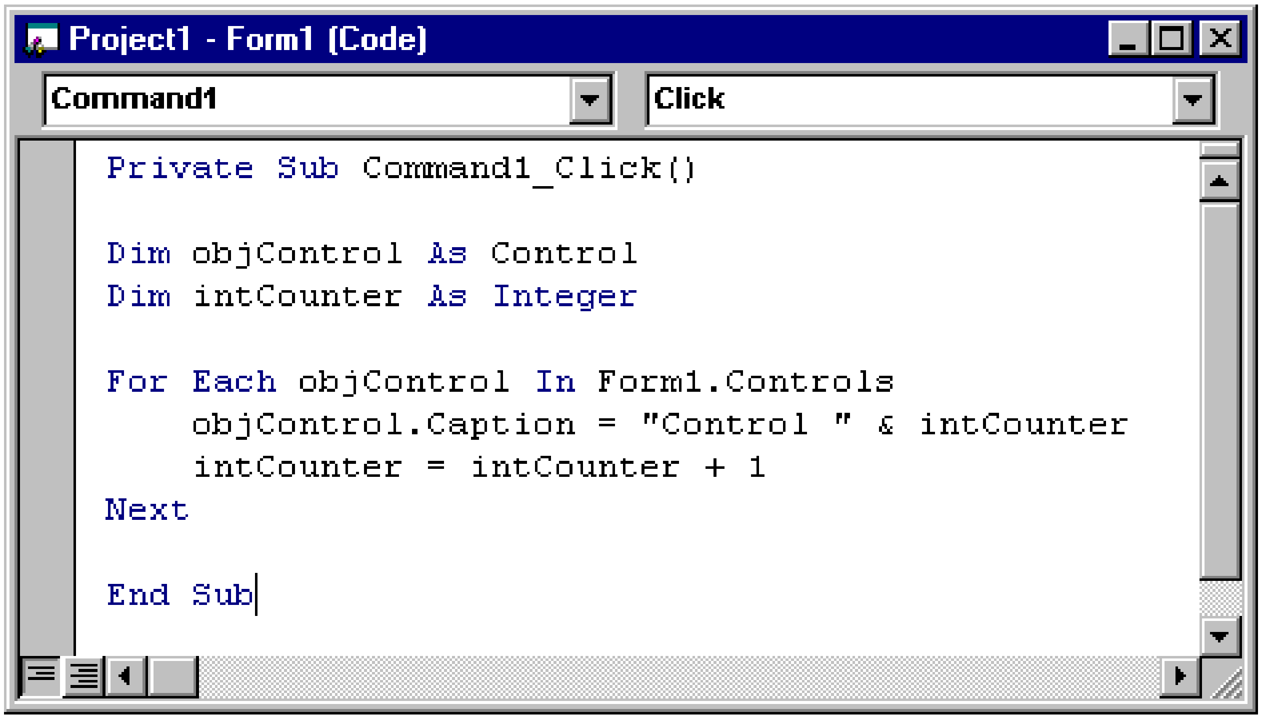
Binding is a term that describes how quickly Visual Basic can determine to what type of Object your variable is pointing. Depending upon the Object Variable type, this determination may be made as early as Design time, and as late as when the Object Variable is referenced in code in your running program.

The bottom line is that the earlier Visual Basic knows the details of the Object Variable it is working with, the better the performance of your programming. As an off the wall analogy, Early Binding is like throwing a dinner party and knowing ahead of time what you will be serving for dinner that night. Late Binding is like throwing a dinner party and calling a caterer for menu suggestions after your guest arrive---dinner will be late!

With Early Binding, we declare an Object Variable as a specific type of object---CommandButton, for instance--and Visual Basic knows very early on the type of Object it is dealing with---so early in fact, that Visual Basic can identify the Object prior to the program actually running.

The bottom line to always Early Bind---if that is possible.

Let me show you an example of that For…Each statement now. Let's place this code in the Click Event Procedure of the Command Button on the form…



```
Private Sub Command1_Click()

Dim objControl As Control
Dim intCounter As Integer

For Each objControl In Form1.Controls
    objControl.Caption = "Control " & intCounter
    intCounter = intCounter + 1
Next

End Sub
```

What we're doing here is using the For…Each statement, in conjunction with an Object Variable called objControl, to 'loop' through each control in the Controls Collection, and when we access that Control (via an Object variable), we're changing the Caption of that Control.

First, let's turn our attention to the declaration of that Object Variable, which is then used within the For…Each loop to reference an individual control in the Controls Collection…

```
Dim objControl As Control
```

You might be wondering, is this an example of Early Binding, or an example of Late Binding?

I guess it's really a little of both. A variable declared as a 'Control' type doesn't tell Visual Basic a lot about the type of Object it will be encountering at run-time---only that the Object will be a Control of some type.

Why didn't we declare the Object Variable as one of the other Object types? Perhaps a CommandButton or Checkbox?

Well, there's a problem there. We can't get too 'fine' with our Object Variable declaration type--as some of the items in the Controls Collection are CommandButtons, and some are Checkboxes---plus there are other Control types. If we declare the wrong type of Object variable, and then use that Object Variable to reference an Object of the wrong type, Visual Basic will bomb.

.

In essence, by choosing the Control Type of Object Variable, we've really chosen the lowest common denominator---after all, we can be certain that each item in the Controls Collection is a Control.

Let's take a look at the rest of the code now. This line of code is a Integer variable declaration---the type of Variable declaration that you are most familiar with. We're using this variable as a counter variable to give each Caption a unique Caption name…

```
Dim intCounter As Integer
```

The rest of the code in this event procedure shows us the magic of the For…Each statement. With just a few lines of code we'll be able to change the Caption of every control on the form, without having to know the names of the Controls or even how many are on the form!

This line of code starts the ball rolling by telling Visual Basic to 'loop' through each item in the Controls Collection of Form1, and then assign a reference to that item to the Object Variable objControl…

```
For Each objControl In Form1.Controls
```

Now that we've assigned a reference to a particular control within the Controls Collection to our Object variable objControl, this next line of code modifies the Caption property of that particular control, using the name of the Object Variable in conjunction with the Caption

Property…

```
objControl.Caption = "Control " & intCounter
```

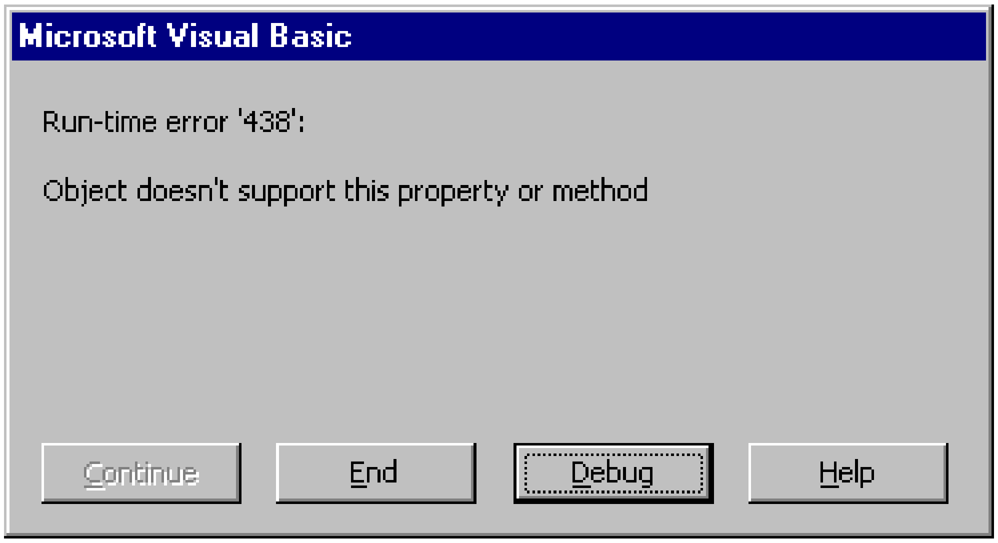
Now we increment the value of our Counter variable so that the next Caption is unique…

```
intCounter = intCounter + 1
```

and finally, this line of code tells Visual Basic to continue the process until it has 'looped' through all of the items in the Controls Collection…

```
Next
```

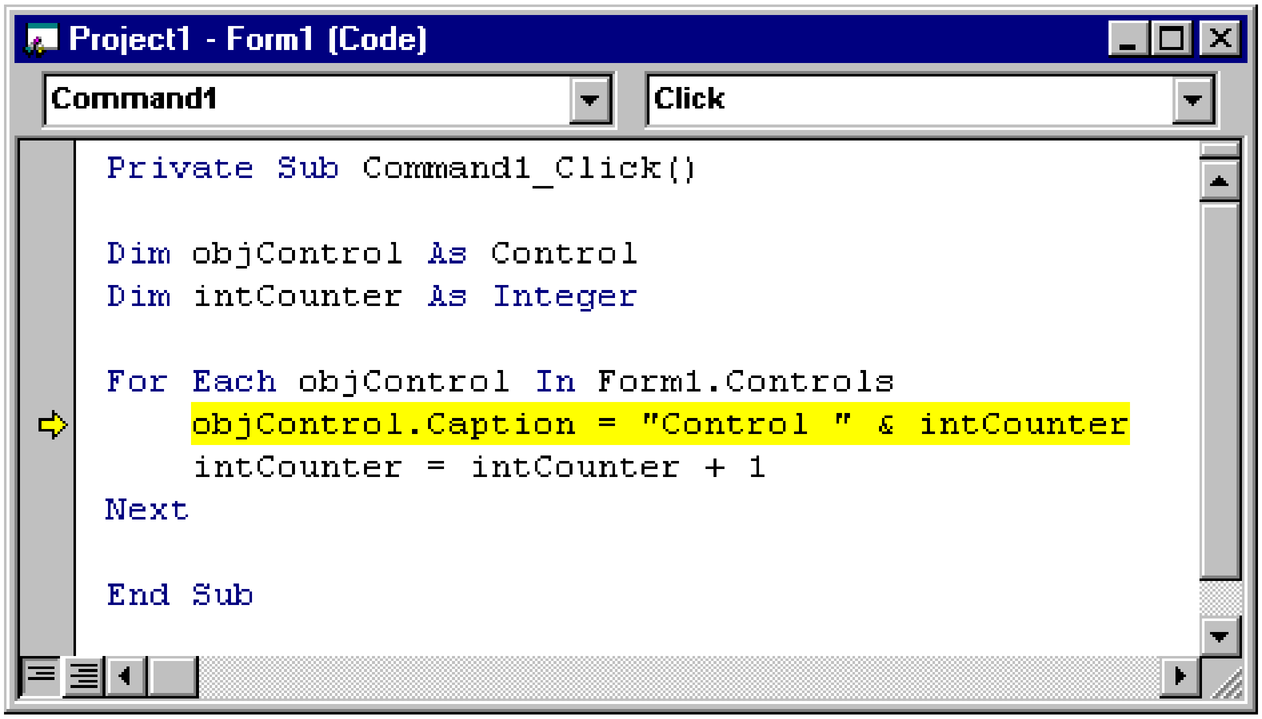
It's time we run this code now, but I must I warn you, there's a slight problem with the code when we click on the Command Button.



Run-time error '438':

Object doesn't support this property or method

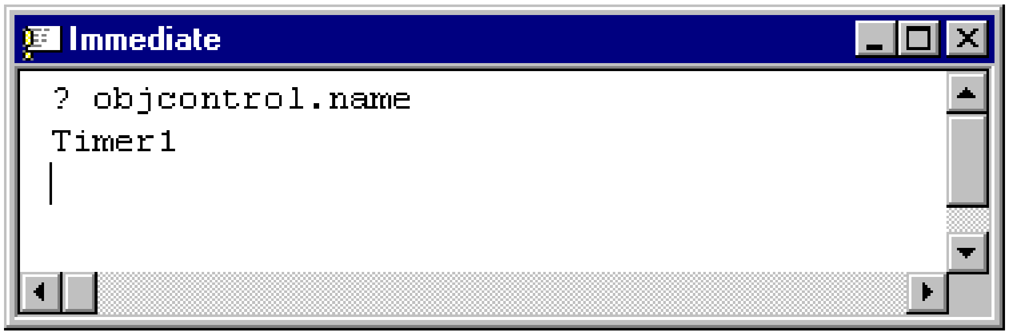
The program has bombed. If we click on the Debug button ….



```
Private Sub Command1_Click()

Dim objControl As Control
Dim intCounter As Integer

For Each objControl In Form1.Controls
    objControl.Caption = "Control " & intCounter
    intCounter = intCounter + 1
Next

End Sub
```

Visual Basic is telling us there's something wrong with the line of code where we try to change a Control's Caption. The assignment statement looks fine. What could be wrong? Is there something wrong with the code.

No, the code is fine, the problem is the Control that we happen to be working with. Let me show you. If we type this statement into the Immediate Window (bear in mind, the program is paused now, and we can determine the item in the Controls Collection that our Object Variable is pointing to)…



```
? objcontrol.name
Timer1
```

It's the Timer Control. What's wrong there?

The answer is that the Timer Control has no Caption property.

You must be careful when working the Controls Collection, particularly when you start modifying Property values of Controls within the For…Each loop. You may code a statement that seems perfectly fine, yet have it bomb like this. Remember, many controls share the same Properties in common (Top, Left, Height and Width, for instance), but with the exception of the Name and Index properties, there isn't another property that is common to every control.

Because of that, when Visual Basic executed this line of code upon 'reaching' the Timer Control in the Controls Collection

objControl.Caption = "Control " & intCounter

it interpreted that to mean

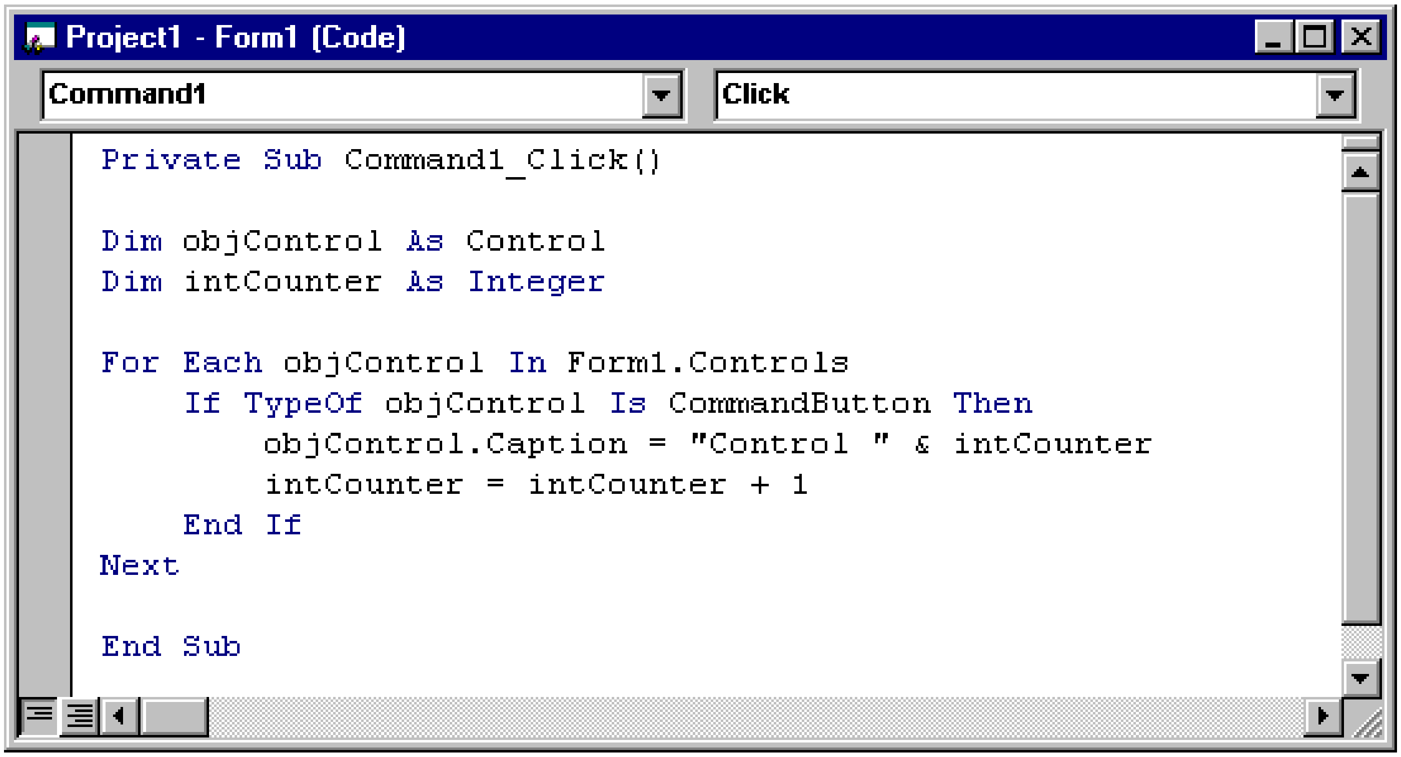
Timer1.Caption = "Control " & intCounter

which is impossible. The Timer control does not have a Caption. The result is the error message that we received here.

At this point, you are probably saying, what good is the For…Each statement and the Controls Collection if the program is going to bomb---that doesn't seem very useful, does it.

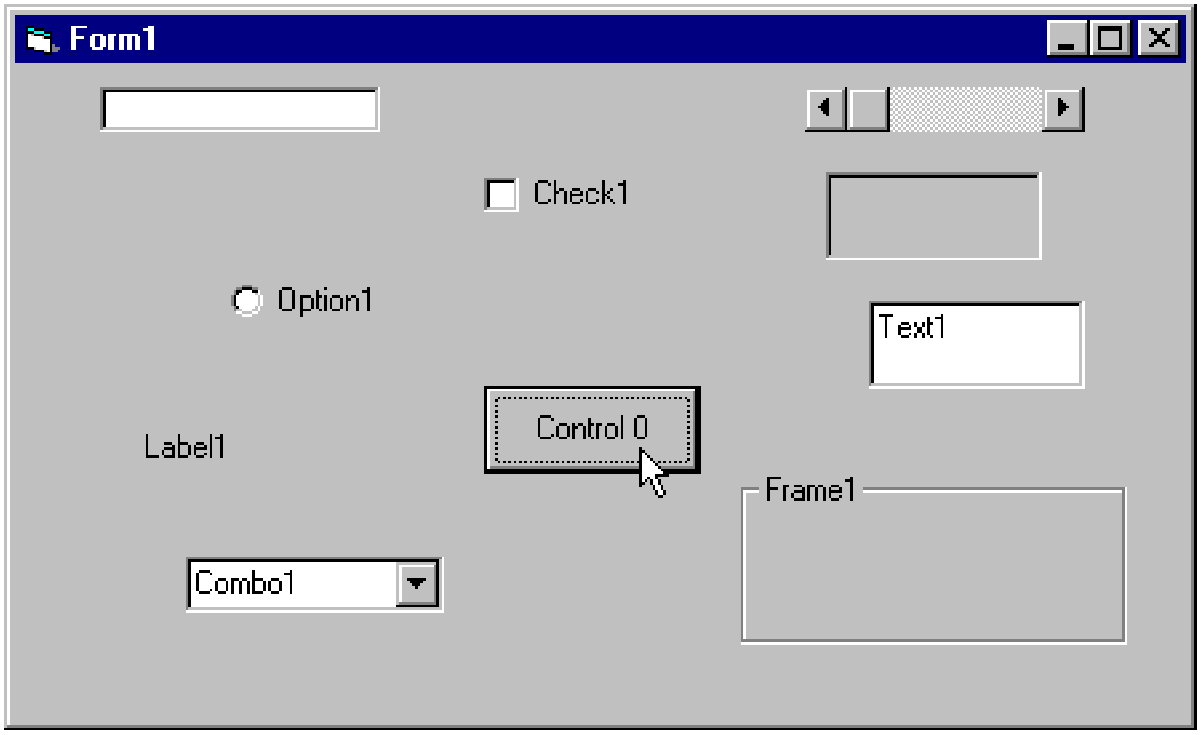
Not to worry, though. There are a couple of things we can do to prevent this from happening.

Visual Basic has a TypeOf operator that we can use to determine the type of Control represented by an Object Variable. Look at this code…

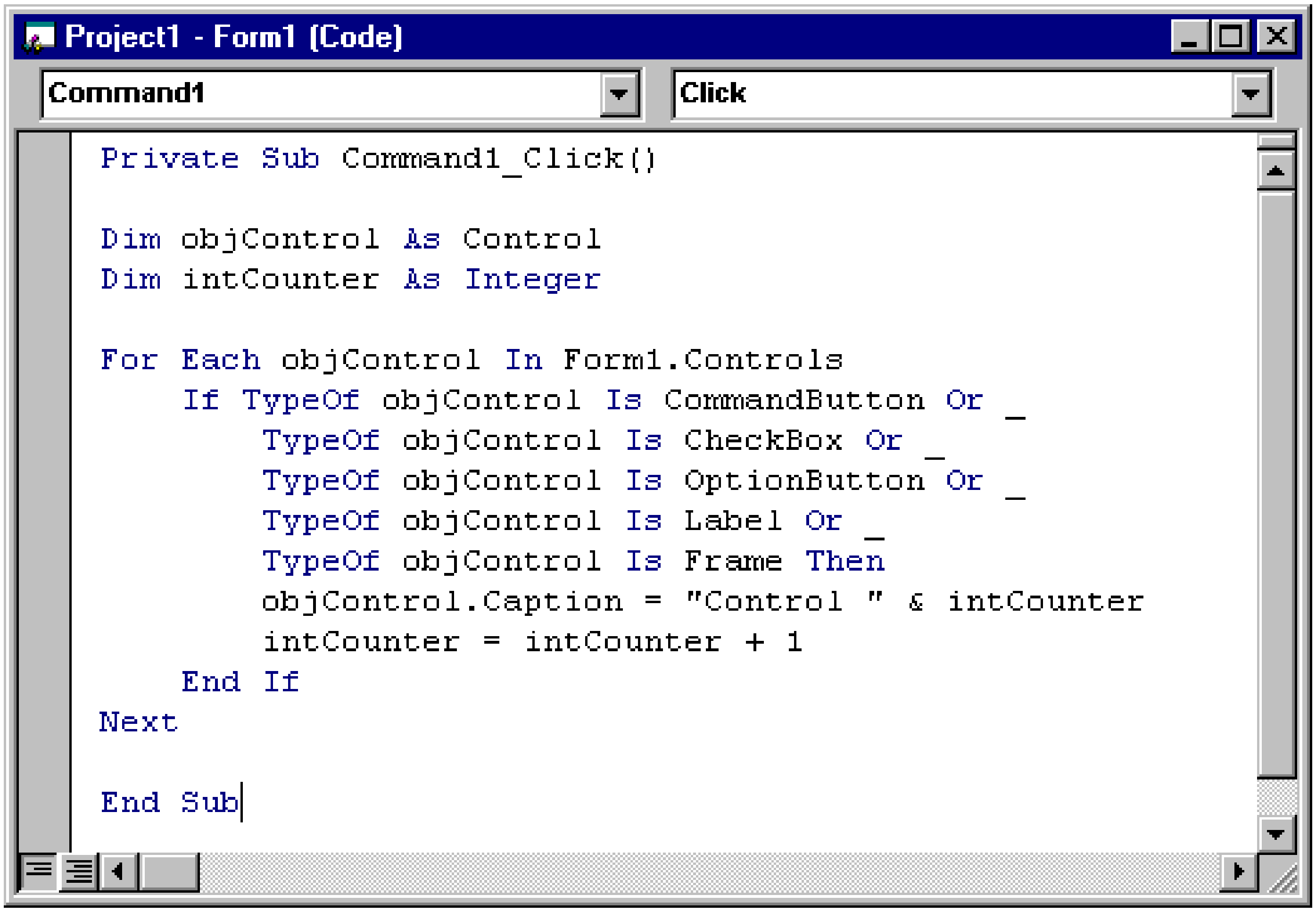


```
Private Sub Command1_Click()

Dim objControl As Control
Dim intCounter As Integer

For Each objControl In Form1.Controls
    If TypeOf objControl Is CommandButton Then
        objControl.Caption = "Control " & intCounter
        intCounter = intCounter + 1
    End If
Next

End Sub
```

If we now run the program, when we click on the Command button, the program no longer bombs. Unfortunately, since we checked to see if the Object Variable pointed to a Command Button, the only control whose Caption we changed is the Command Button---all of the other Controls remained unchanged from their defaults.

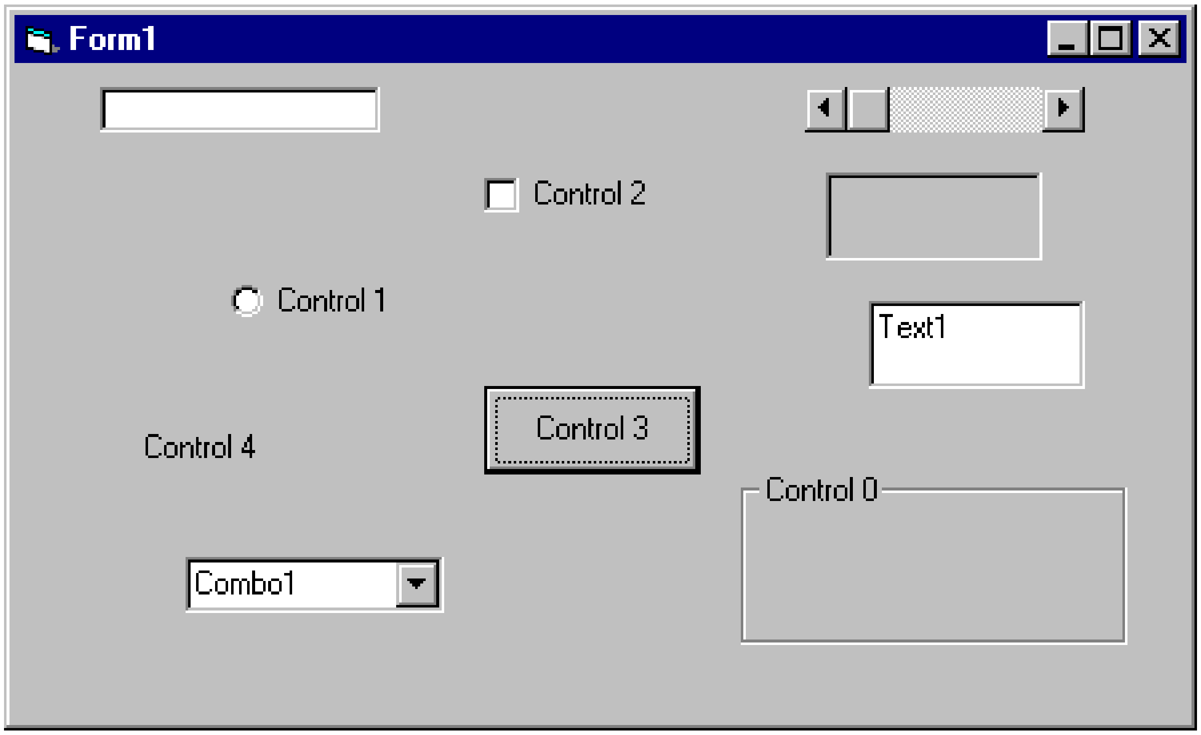


While this is an improvement, it's not really what we wanted to do. We wanted to change the Caption property of every control with a Caption. We could remedy that by adding some code for every control that supports the Caption property, such as the Checkbox, OptionButton, Label and Frame…



```
Private Sub Command1_Click()

Dim objControl As Control
Dim intCounter As Integer

For Each objControl In Form1.Controls
    If TypeOf objControl Is CommandButton Or _
        TypeOf objControl Is CheckBox Or _
        TypeOf objControl Is OptionButton Or _
        TypeOf objControl Is Label Or _
        TypeOf objControl Is Frame Then
        objControl.Caption = "Control " & intCounter
        intCounter = intCounter + 1
    End If
Next

End Sub
```

Now if we run the program again, and click on the Command button, every control's Caption has been changed.

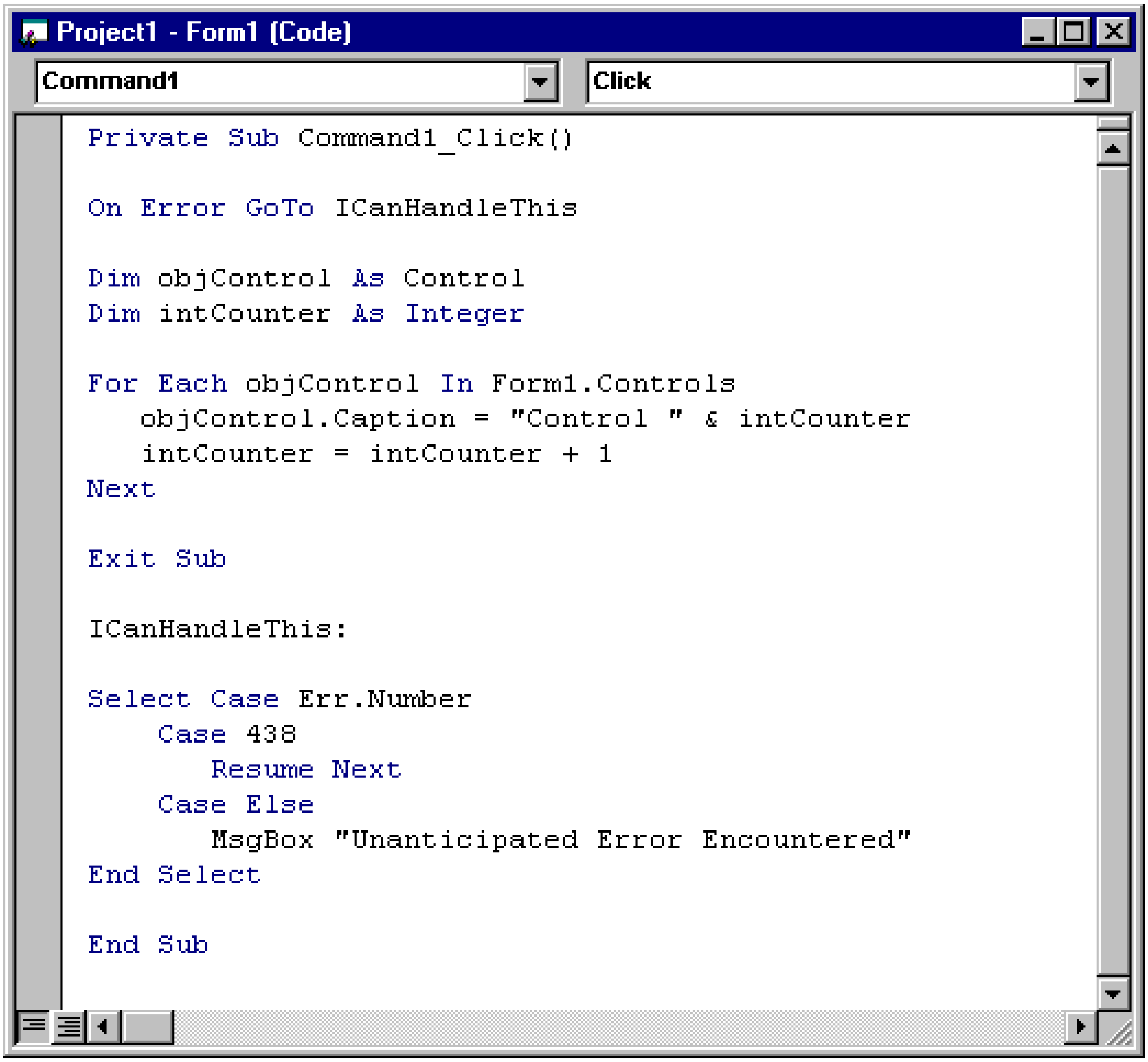


Notice that this time the Caption for the Command button is Control3, not Control0 as it was before. That tells us that the first control with a Caption property encountered within the For…Each loop was the Frame, followed by the OptionButton, followed by the Checkbox, and then the Command Button and finally the Label control.

Again, I must caution you---that order does not necessarily reflect the sequence that the controls were placed on the form. In fact, I had placed the Frame Control on the form *after* the Command Button, yet it appears earlier in the Controls Collection than the Command Button. In fact, it's first! The only thing that you should count on when working with the Controls Collection is the knowledge that every control on the form is in it--and that you can work with the individual controls just like we did here.

Again, the cynics among you will be questioning the TypeOf operator within the For…Next loop. Is it really necessary to manually check for each Control that has a Caption property like this. Surely, there must be a better way. And there is.

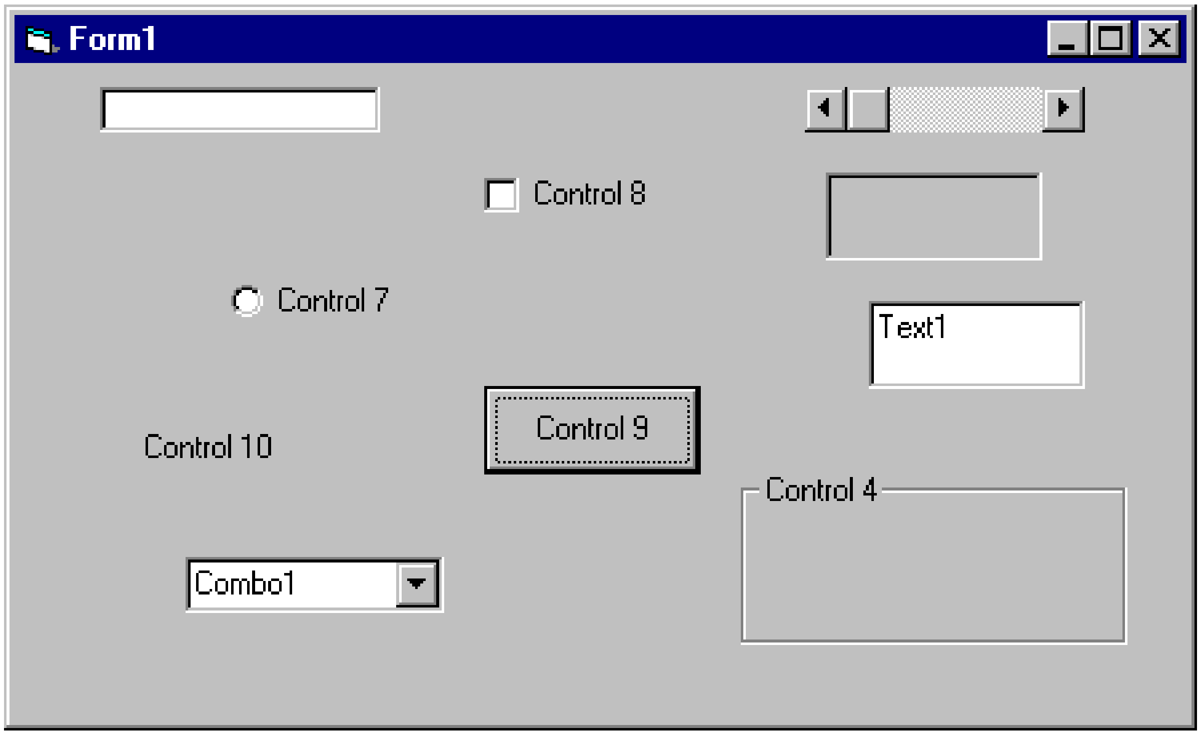
If you think back to my Intro to Programming with Visual Basic 6 book, and recall the discussion of Error Handling, you might be thinking that we can 'trap' for the error that is generated when we try to refer to a property that doesn't exist---Error Number …438. If we add Error Handling (next month's topic by the way) to the code in the Click Event Procedure of that command button, it would look like this…



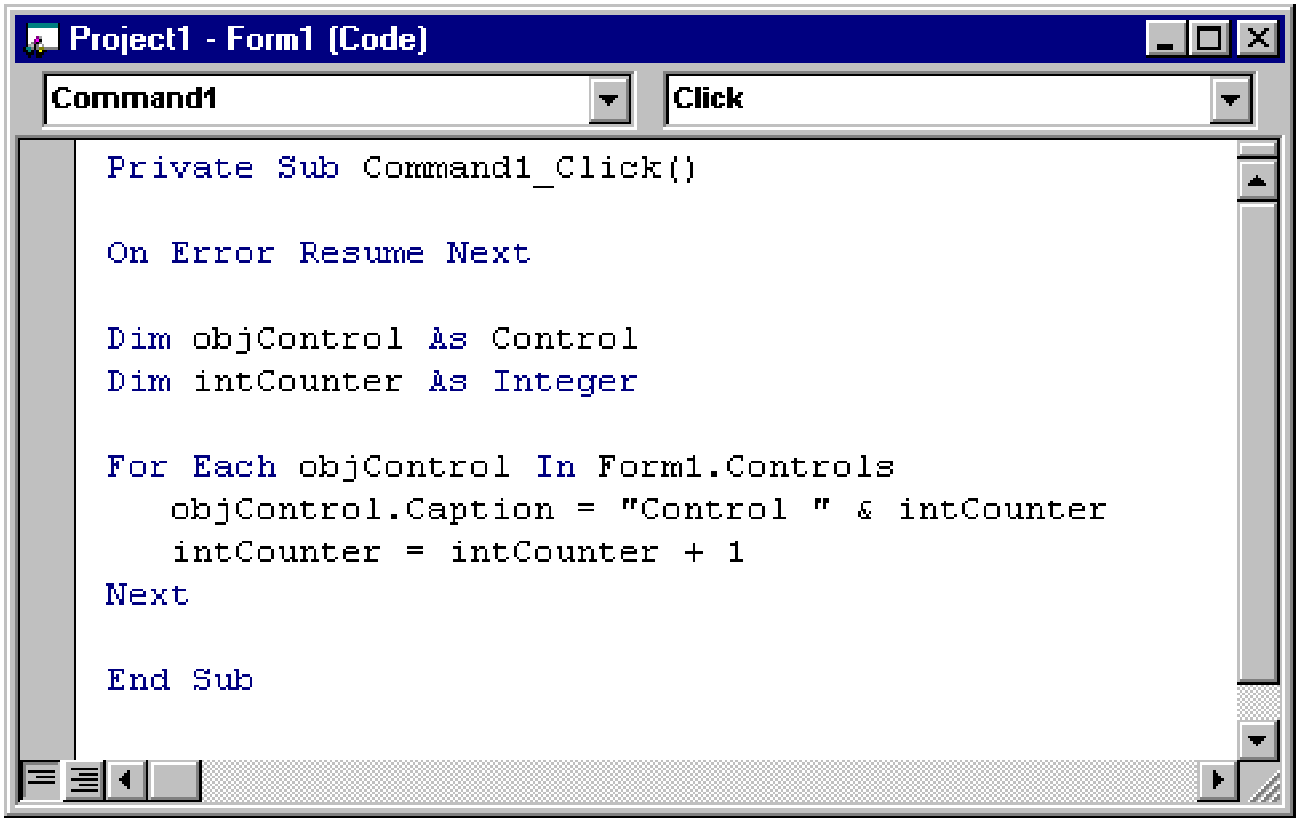
```
Private Sub Command1_Click()

On Error GoTo ICanHandleThis

Dim objControl As Control
Dim intCounter As Integer

For Each objControl In Form1.Controls
   objControl.Caption = "Control " & intCounter
   intCounter = intCounter + 1
Next

Exit Sub

ICanHandleThis:

Select Case Err.Number
    Case 438
        Resume Next
    Case Else
        MsgBox "Unanticipated Error Encountered"
End Select

End Sub
```

Notice the inclusion of the Error Handler. Readers of my Intro book should be familiar with this. Here we are using a Visual Basic Error Handler to 'intercept' the error generated when we try to reference an object's property that doesn't exist. If the error is generated, we execute the Resume Next statement which causes Visual Basic to resume execution at the next line of code---thus bypassing the attempt to set a property that the Control doesn't support.

If we now run the program again, and click on the Command button, we should see the following screenshot…



This gives you a chance to really see the benefits of the Controls Collection in action. Now, just one more thing before we move onto talking about the Forms Collection---and that's something call inline Error Handling. With Inline Error Handling, a single statement can be used to handle any errors which may occur in the Event Procedure---in other words, there's no Error Handling Paragraph. Like this…



```
Private Sub Command1_Click()

On Error Resume Next

Dim objControl As Control
Dim intCounter As Integer

For Each objControl In Form1.Controls
   objControl.Caption = "Control " & intCounter
   intCounter = intCounter + 1
Next

End Sub
```

As you can see, I've changed the code significantly. Instead of calling code to execute in a dedicated Error Handler when an error occurs, here we're simply telling Visual Basic to execute the next line of code. That's what this line of code does…

On Error Resume Next

That saves quite a bit of coding---and is used quite frequently when you can anticipate that the only errors generated in the procedure will require the resumption of code at the next line.

Let's move onto the Forms Collection now.

The Forms Collection

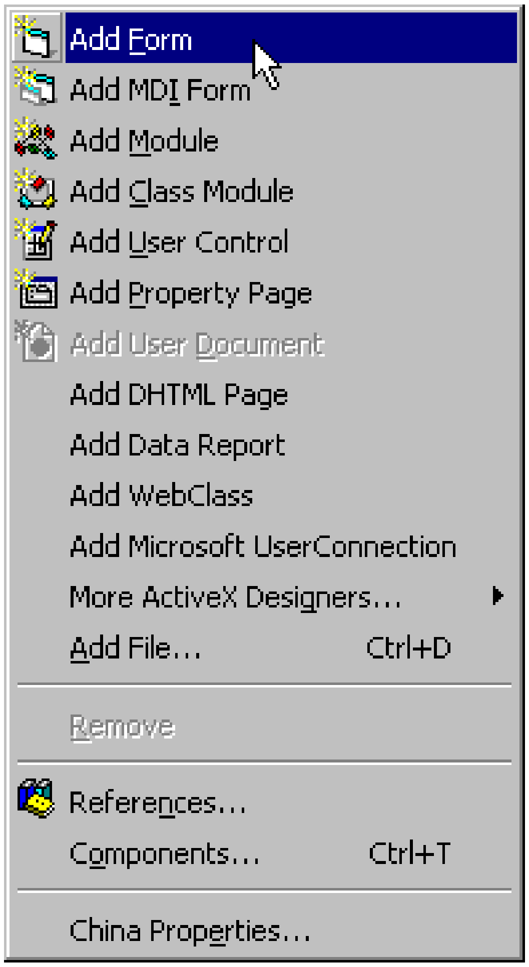
Like the Controls Collection, the Forms Collection is also a group of items---however, instead of each item representing a Control on a form, each item in the Forms Collection points to a loaded form in the Visual Basic application.

Those of you who read my Intro book know that it contains just a single form--frmMain---but it is possible for a Visual Basic Application to have more than one form (by the way, in the sequel to my Intro book, Learn to Program with Visual Basic Databases, we expand to several forms in the China Shop Project!). With an application that has several forms, the Forms Collection can come in quite handy.

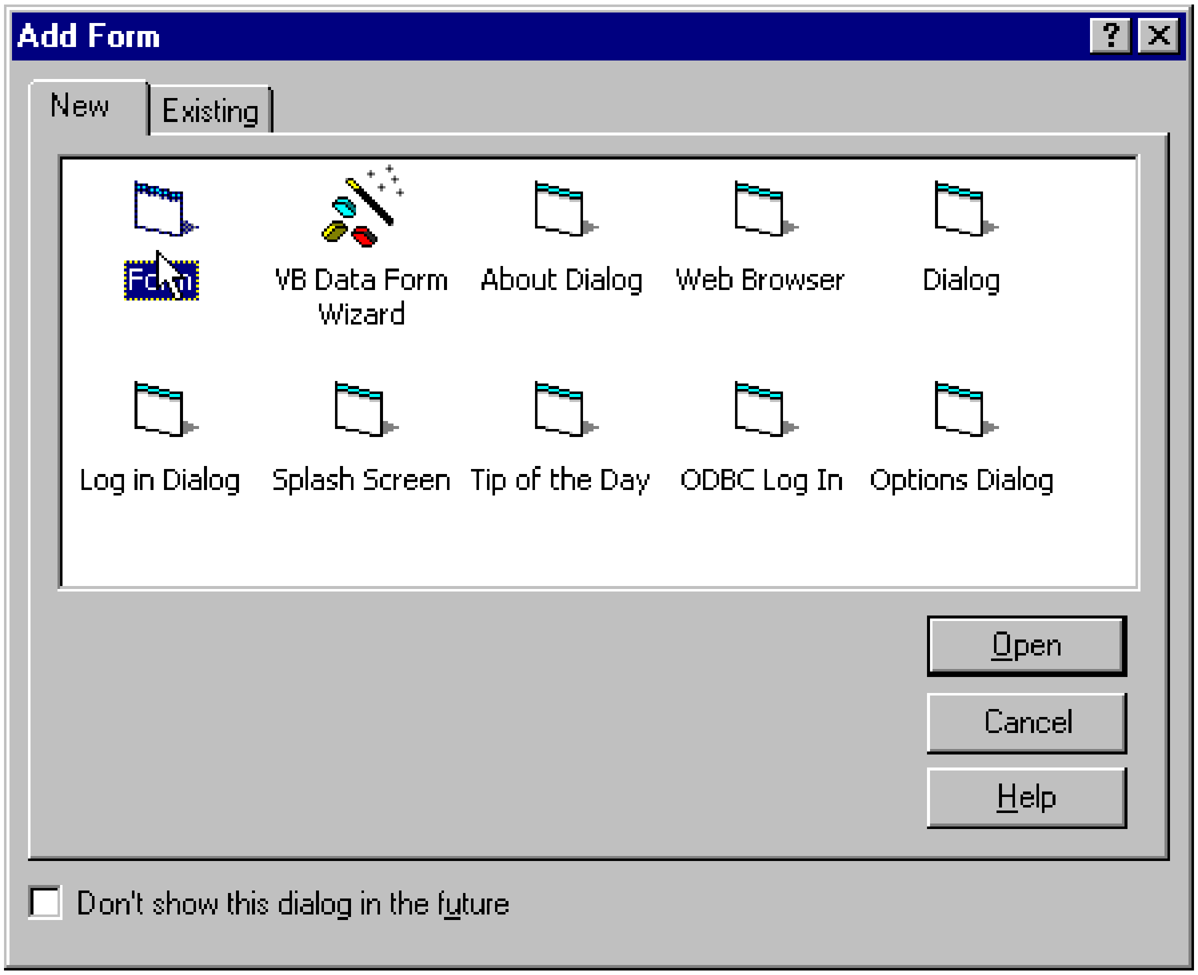
Let me create a new Visual Basic project to illustrate how the Forms Collection works.

Adding a second form to our project is relatively easy, all we need to do is select Project-Add Form from the Visual Basic Menu Bar…

.

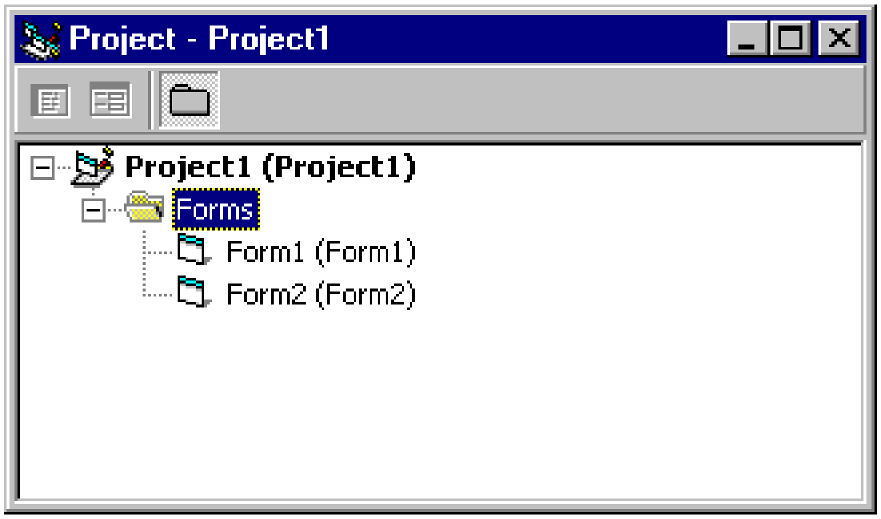


Then when this screen shot appears…



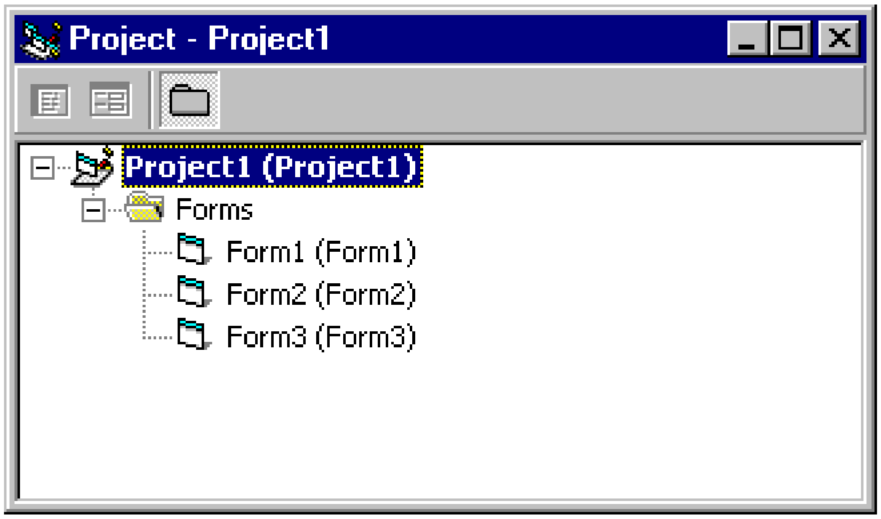
all we need to do is make sure the 'New' tab is selected, and then either double-click on 'Form' or select 'Form' and click on the 'Open' Button. Visual Basic will then display a new form for us, with a Caption reading 'Form2'. (Not shown, but it will happen!)

If we now bring up the Visual Basic Project Explorer Window, we should now see two forms---Form1 and Form2.



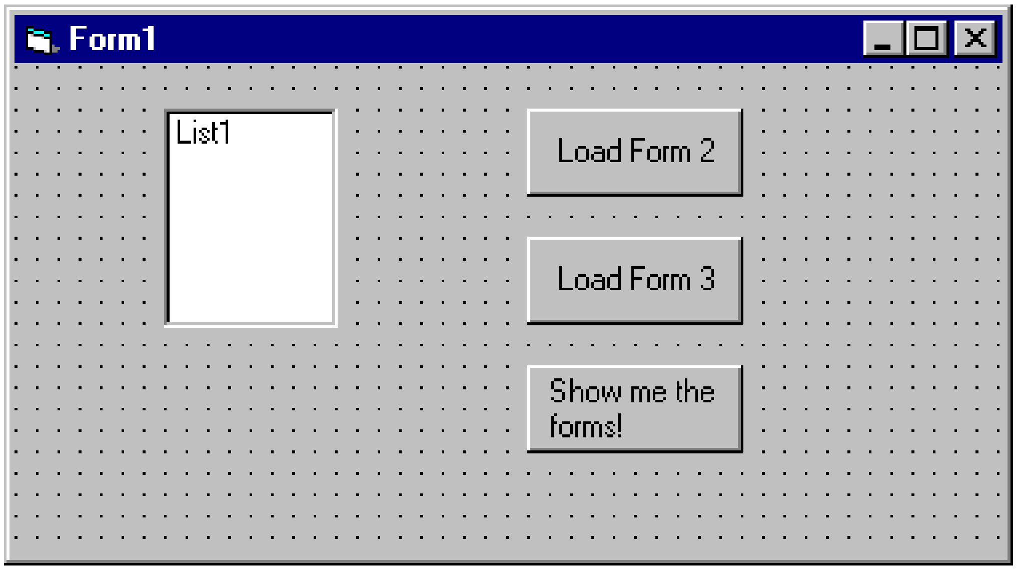
If we repeat this process to add a third form to the application once again by selecting

Project-Add Form from the Visual Basic Menu Bar, a third form will be opened up in the IDE (again, not shown but it will happen). Once again, if we bring up the Visual Basic Project Explorer Window, we can verify the number of forms in our project---three.



Now that we have three forms, let's see what we can do with them. If we just run the project at this point, the first form will become visible---and that's the end of the story. Without intervention on our part, we'll never see Form2 or Form3 at run time. There needs to be some way to make the second and third forms visible---and we can do that by placing code somewhere on the first form.

Let's add a ListBox control and three command buttons to the form, captioning the Command buttons 'Load Form 2', 'Load Form 3' and 'Show me the forms!' respectively.



Now let's place some code in the Click event procedure of that first command button which will display the second form. Frequently students and readers who have researched displaying a second form in their projects gravitate towards the Load Statement. If we place this line of code in the Click Event Procedure of the Command Button…

```
Load Form2
```

that will load the second form, but it won't be visible! That's right. The Load Statement by itself loads the form, but no one will be able to see it. A loaded form is not necessary visible.

At this point, quite a few students look to the main form of the project, and cite the fact that when the application first starts, the main form becomes visible, with no intervention required by the programmer. That's true---that happens to be the default behavior of the Startup form in your project---to be loaded and to become visible.

But that default behavior is only true of the Startup form. For additional forms after the first one, the programmer needs to explicitly display the additional forms, and you can do that in one of two ways: Execute the Load Statement, and follow that up with either with a statement to set the Visible property of the form to True…

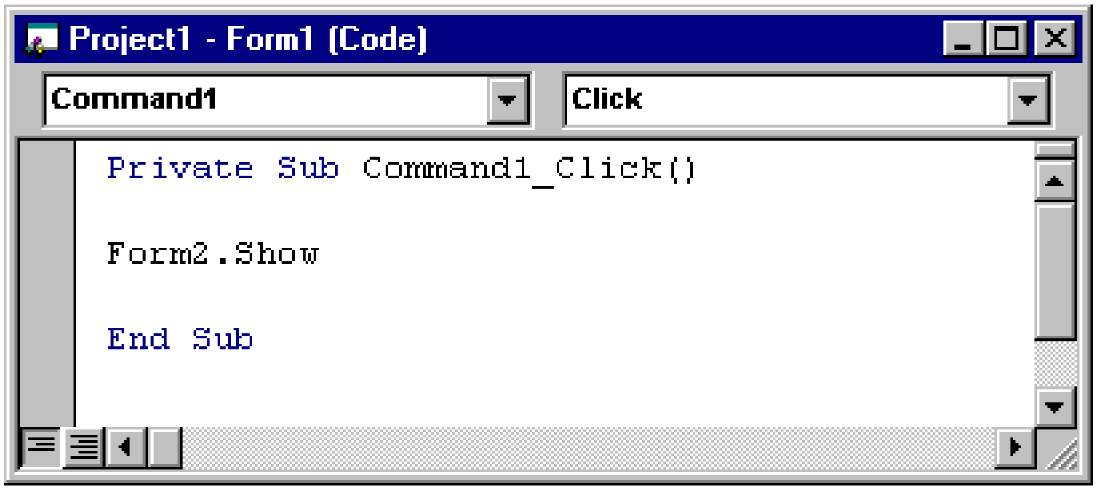
```
Load Form2
Form2.Visible = True
```

or execute the Show Method of the Form, like this…

```
Load Form2
Form2.Show
```

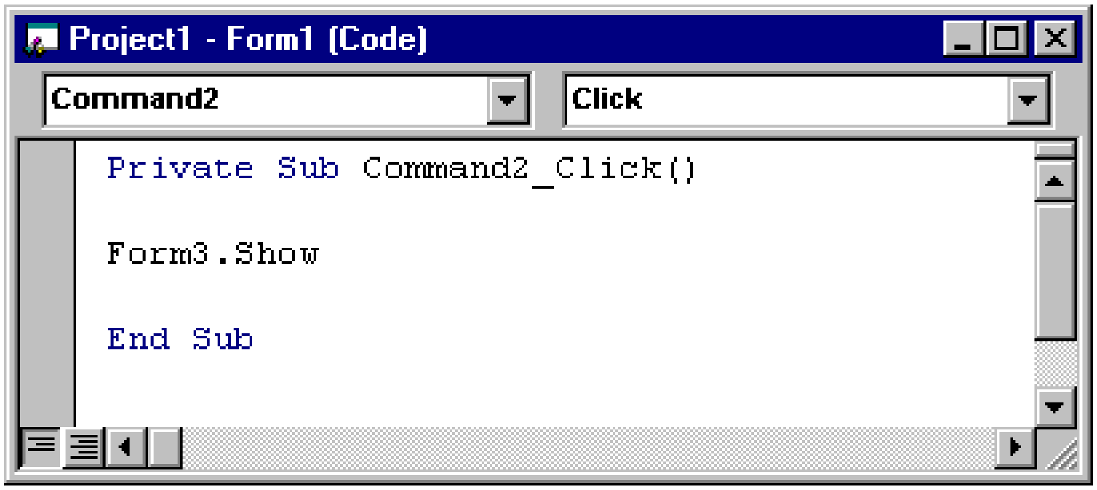
However, the easiest thing to do is to simply execute the Show Method of the form. Executing the Show method automatically loads the form for you, and displays it at the same time.

Let's place this code into the Click Event Procedure of the first Command Button…



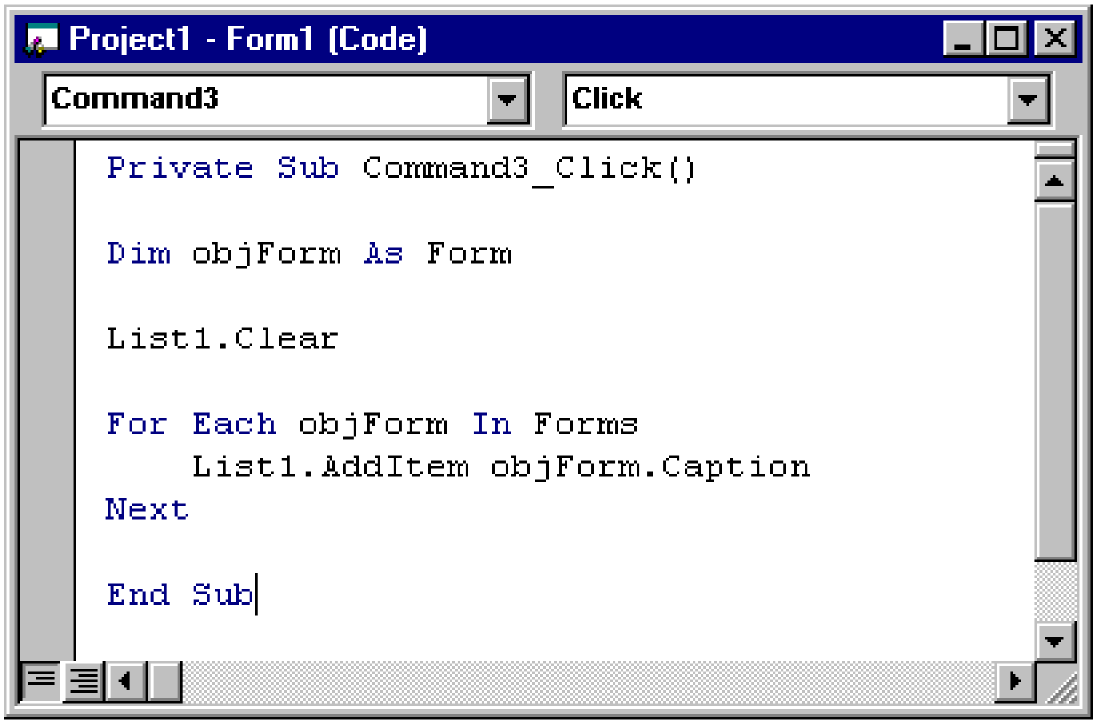
```
Private Sub Command1_Click()

Form2.Show

End Sub
```

and this code into the Click Event Procedure of the second Command Button…



```
Private Sub Command2_Click()

Form3.Show

End Sub
```

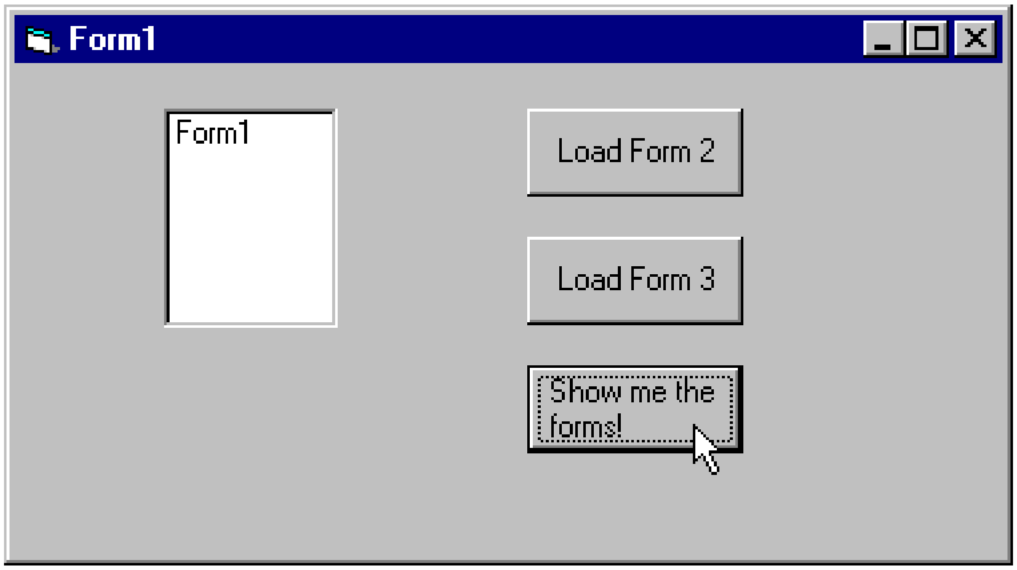
Now for the third command button---this code will use the Forms Collection of the application to load into the ListBox a list of all the loaded forms---remember, the items in the Forms Collection are only loaded forms---forms that are designed but not loaded will not be members of the controls Collection.



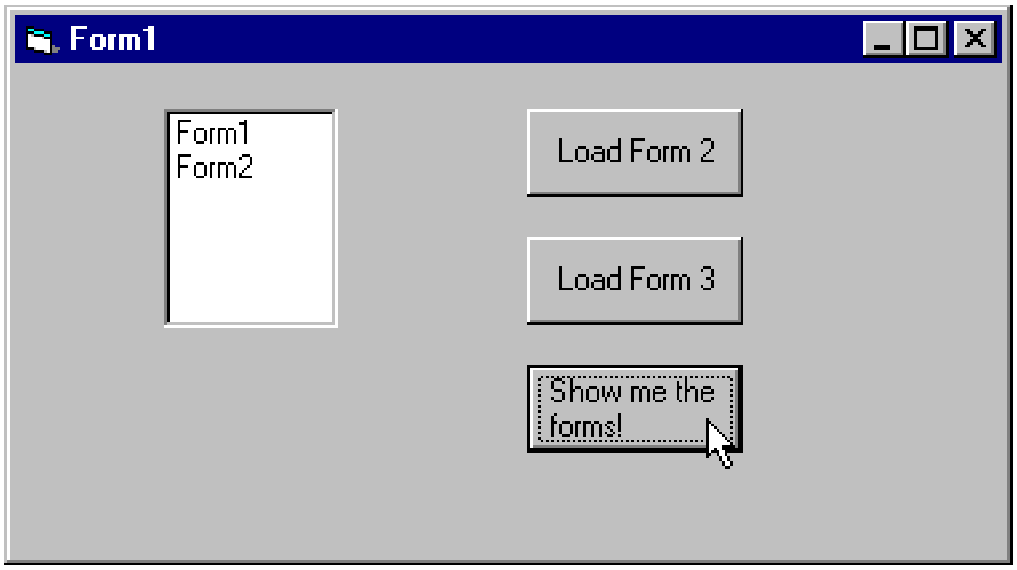
```
Private Sub Command3_Click()

Dim objForm As Form

List1.Clear

For Each objForm In Forms
    List1.AddItem objForm.Caption
Next

End Sub
```

This code looks very similar to the code we used when working with the Controls Collection. Once again, we are using an Object variable, in conjunction with the For…Each loop structure to loop through the loaded forms in the application, and after first clearing the contents of the ListBox, we then add an entry to the ListBox for each loaded form in the application.

If we now run the program, and them immediately click on the command button captioned 'Show me the forms!'. the following screen shot will be displayed.

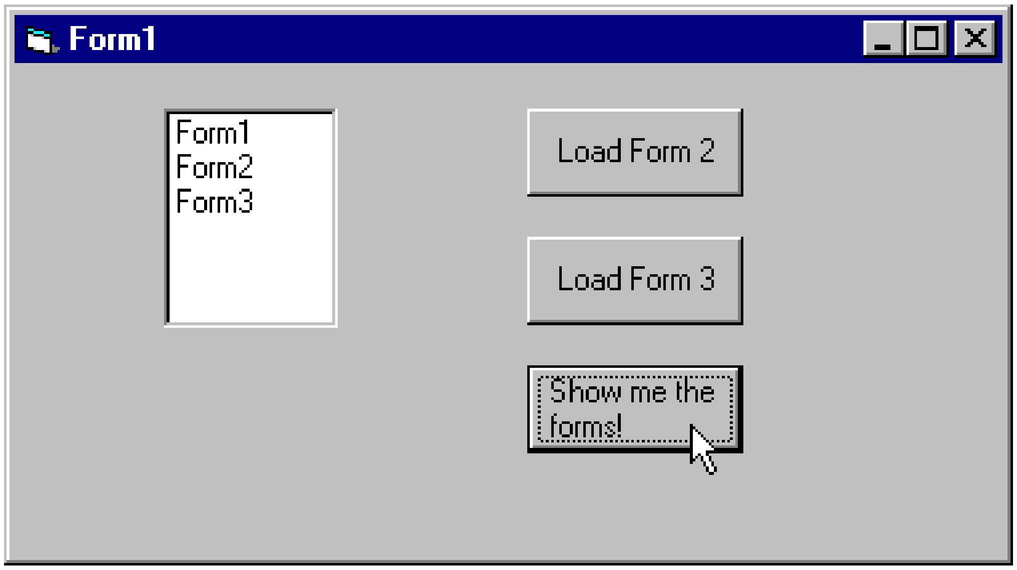


Nothing terribly fancy here---with just the single form loaded and displayed, this is all we would expect to see in the ListBox---just Form1. But if we now click on the first Command Button to load Form2…(you can't see it, but Form2 just loaded 'above' the first form…), and then clock on the command button on Form1 captioned 'Show me the forms!', our ListBox should now show two forms---Form1 and Form2---like this…



There are not two forms loaded, and therefore two Forms in the Forms Collection.

If we now click on the second command button to load Form2 (which will then appear 'above' Form1) and then click on the button captioned 'Show me the forms!', the following screenshot will be displayed.



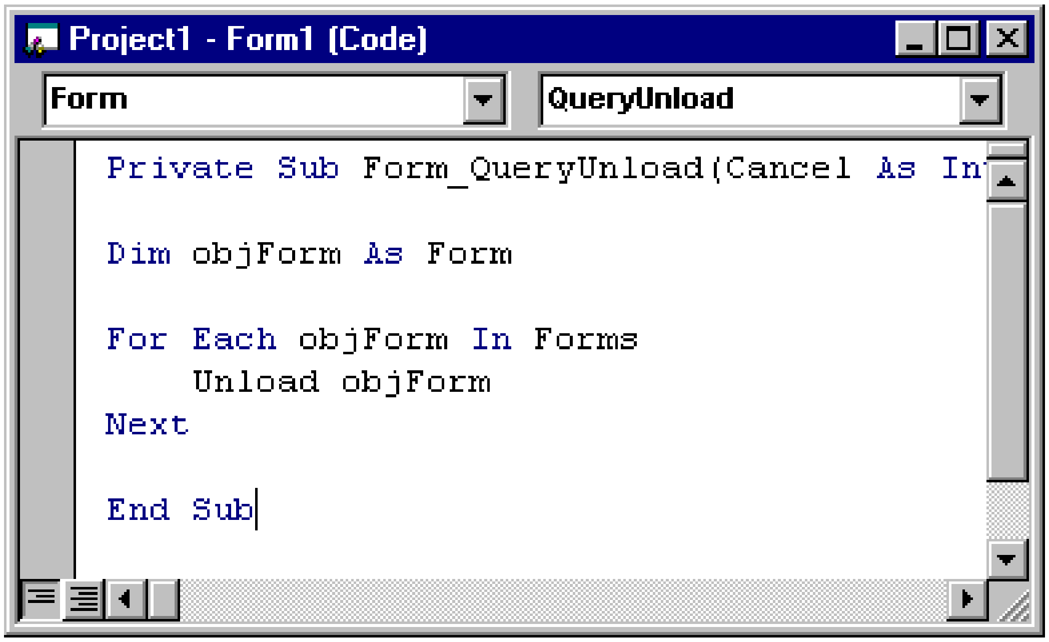
We now have three forms loaded---therefore, we also have three items in the Forms Collection.

Now in my University classes, usually someone at this point wonders out loud "What's the point? I don't see any practical use for the Forms Collection" and asks if I use it in my work.

The answer is 'yes', I do use it in my work. I typically use the Forms Collection, when I'm unloading the main form of my project. For instance, in an application where we have several forms, it may be possible for the user to unload the main form of the application while other forms are open. That's the last thing we want to do---in effect, we would be leaving the other forms orphaned. (By the way, you can test this for yourself. Run the sample project, load Form2 and then Form3, and then close Form1---Form2 and Form3 will remain loaded and visible!)

What we need to do is to ensure that if the user unloads the main form of the application, that we unload all of the loaded forms in the Application---and the Forms Collection is a perfect way to ensure that.

By placing this code in the QueryUnload Event Procedure of the main form…



```
Private Sub Form_QueryUnload(Cancel As In

Dim objForm As Form

For Each objForm In Forms
    Unload objForm
Next

End Sub
```

we can 'loop' through each loaded form in the Forms Collection, and then unload ---and then finally unload the main form of the application. If we now run our sample program again, and load Form2 and Form3, and then close Form1, Form2 and Form3 will also be unloaded.

Using a Collection instead of an array

We just saw two examples of Visual Basic System Collections, the Forms and Controls Collections. Now I want to show you how easy it is to create your own Collection, and give you a chance to compare the creation and use of a Collection with that of the creation and use of a single dimension array.

As we saw in my Intro book, the advantage of a Variable array over a regular variable is the ability to refer to the variable array with a single name---and to be able to refer to each element within the array through the use of a subscript or index.

As an example, suppose we own a company in the eastern portion of the United States with offices in thirteen states, and we want to keep track of the number of employees in each state. There are several ways, using Array processing, that we can do that. One way is to create a single Dimension array called States to hold both the names and totals of employees in each state.

Let's create a new Visual Basic project, and place three command buttons on the form. In the General Declarations Section of the form, we'll place this Variable Array declaration…

Private m_States(12) As String

This declaration creates a single dimensional, fixed array of 13 elements called m_States, with a lower bound of 0 and an upper bound of 12. We need to declare this array in the General Declarations Section of the form since we'll be referring to the Array from two different event procedures on the form. Next we need to write the code to load the array---you'll notice that I am including both the name of the State and the number of employees in each state as the value in the array element. Let's place that code in the Click Event Procedure of Command1.

```
Private Sub Command1_Click()

m_States(0) = "123-Connecticut"
m_States(1) = "523-Delaware"
m_States(2) = "12-Georgia"
m_States(3) = "311-Maryland"
m_States(4) = "23-Massachusetts"
m_States(5) = "11-New Hampshire"
m_States(6) = "43-New Jersey"
m_States(7) = "410-New York"
m_States(8) = "56-North Carolina"
m_States(9) = "1012-Pennsylvania"
m_States(10) = "22-Rhode Island"
m_States(11) = "12-South Carolina"
m_States(12) = "83-Virginia"

End Sub
```

Those of you comfortable and familiar with arrays may recognize right away that this example really calls for a two dimensional arrays. But the purpose of my illustration here is to compare an Array with a Collection---and Collections are really one dimensional animals---we really can only compare a Collection with a one dimensional array.

You may be wondering why I placed the number of employees first in the Array---as you'll see in a minute, we'll be using a little trick with the Visual Basic Val function to 'extract' the number of employees from the Array element---that number will then be used to sum the total number of employees in the array. That's why the number of employees must go first. This is a drawback with a single dimensional array where we are really 'storing' two discrete pieces of information.

Now that we've written the code to load the array, we need to write the code to 'loop' through the elements of the array, print the names of the states and number of employees on the form, and then display the total number of employees and states on the form. Here's the code to do that.

```
Private Sub Command2_Click()

Dim intCounter As Integer
Dim intTotal As Integer

For intCounter = LBound(m_States) To UBound(m_States)
  Form1.Print m_States(intCounter)
  intTotal = intTotal + Val(m_States(intCounter))
Next

Form1.Print

Form1.Print "Total Number of employees: " & intTotal " in " &
intCounter & " states"

End Sub
```

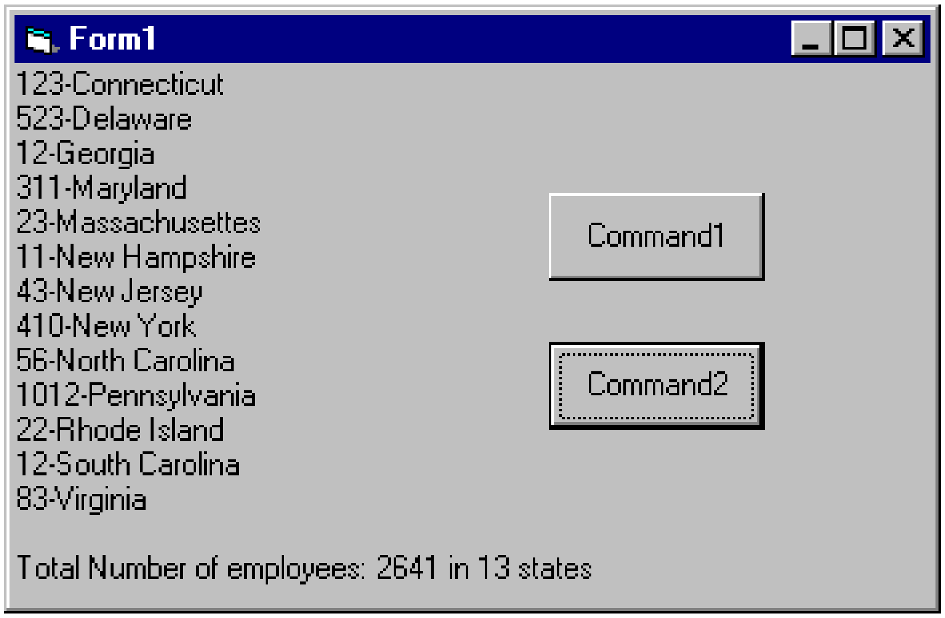
You should be familiar with what we're doing here from my Intro book. We're using a For… Next loop to 'loop' through the elements of the Array, using the LBound and UBound functions to determine the lower and upper bounds of the array---in this case 0 and 12.

Now back to that Val function I mentioned earlier.

The Val function takes a String value and returns a number. The interesting thing about the Val function is that it starts working from the left of the string, and moves to the right, continuing to return numbers until it encounters the first non numeric character. That's why we placed the number of employees on the left side of the string---as soon as the Val function hits the dash in the Array element's value, it will stop. That means it will return only the numeric part of the Array element's value.

Once we have that value, we then add it to the variable intTotal, which is used to total the number of employees in the company.

If we now run the program, and then click on the first command button to load the array, and then click on the second command button to total the number of employees in the company, the following screen shot will be displayed.



```
123-Connecticut
523-Delaware
12-Georgia
311-Maryland
23-Massachusettes
11-New Hampshire
43-New Jersey
410-New York
56-North Carolina
1012-Pennsylvania
22-Rhode Island
12-South Carolina
83-Virginia

Total Number of employees: 2641 in 13 states
```

Notice how we have displayed the Array elements on the form, followed by a total line containing the total number of employees in the Array and the number of states in the Array. Granted, a two dimensional array could accomplish the same thing---but remember, we're doing this to contrast a Collection with an Array.

How does a Collection fit in here? Can we use a Collection in place of an Array?

The answer is 'yes'---and you may find that a Collection is a great tool.

A Collection can be used in place of a one-dimensional array. In fact, because of the unique feature of the Collection Object the allows us to add an item to it using a key value, we can even use a Collection to 'intercept' duplicate values---something that an Array won't stop.

Key?

That's right, items in a Visual Basic Collection are added using the Add method. The Add method has one required argument, the Item name, and an optional argument, *key*. If we add an item to the Collection object using the optional key argument, we can later retrieve the item from the Collection using the same key value.

Let's look at the example we just programmed with the company's thirteen different branch offices. Suppose we wanted to know how many employees are in the New York office? With a Variable Array, we would need to 'loop' through each element of the Array until we found the New York office---which with the One dimensional array we just looked at, would also require us to use some fancy String Manipulation to find the words 'New York' in the array elements.

By the way, looping through an array with just 13 elements may not seem like a big deal---

but suppose there were thousands of elements in the array. That 'loop' through the array could start to become pretty time consuming.

With a Collection, it's possible to add items to the Collection using a key value---for instance the name of the State---and then to later retrieve that Collection item with a single statement, specifying the item's key value of New York.

As I always say, a picture is worth a thousand words. Look at this. Let's start a new Visual Basic project. Once again we'll place two command buttons on the form, and place this code in the General Declarations Section of the form.

```
Private m_colStates As New Collection
```

This is the declaration for a Collection---as was the case with our Variable Declaration a little earlier, by placing this declaration in the General Declarations Section of the form, we ensure that the Collection can be accessed by any procedure within the form---by the way, notice the Hungarian Notation. 'm' for module, and 'col' for Collection.

Further notice that we don't need to declare a Data Type for a Collection. Unlike an Array where each member of the Array must be the same data type, a Collection can contain items of any type---so there's no need in the declaration to specify the data type of the members.

The Add Method

Here's the syntax for the Add Method of a Collection…

Sub Add (*item* As Variant [, *key* As Variant] [, *before* As Variant]

[, *after* As Variant])

By default, if you add an item to a Collection, the first item is placed in position one of the Collection, the second item in position two, and so forth. For instance, this code …

```
m_colStates.Add "123-Connecticut"
m_colStates.Add "523-Delaware"
m_colStates.Add "12-Georgia"
m_colStates.Add "311-Maryland"
m_colStates.Add "23-Massachusetts"
m_colStates.Add "11-New Hampshire"
m_colStates.Add "43-New Jersey"
m_colStates.Add "410-New York"
m_colStates.Add "56-North Carolina"
m_colStates.Add "1012-Pennsylvania"
m_colStates.Add "22-Rhode Island"
m_colStates.Add "12-South Carolina"
m_colStates.Add "83-Virginia"
```

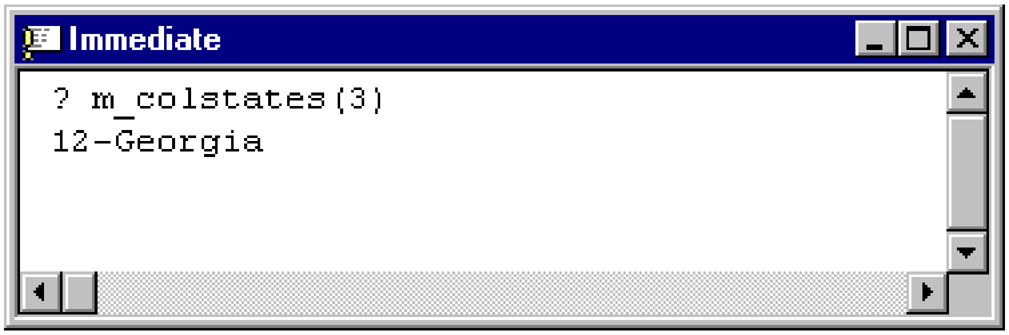
will add 13 items to the m_colStates Collection, with the first item being placed at position 1 of the Collection, and the last item being placed at position 13. Just to remind you, Programmer created Collections are one-based---not zero based like the Forms and Controls Collection.

Let me show you what this code would look like if we utilized the key argument…

```
Private Sub Command1_Click()

m_colStates.Add "123-Connecticut", "Connecticut"
m_colStates.Add "523-Delaware", "Delaware"
m_colStates.Add "12-Georgia", "Georgia"
m_colStates.Add "311-Maryland", "Maryland"
m_colStates.Add "23-Massachusetts", "Massachusetts"
m_colStates.Add "11-New Hampshire", "New Hampshire"
m_colStates.Add "43-New Jersey", "New Jersey"
m_colStates.Add "410-New York", "New York"
m_colStates.Add "56-North Carolina", "North Carolina"
m_colStates.Add "1012-Pennsylvania", "Pennsylvania"
m_colStates.Add "22-Rhode Island", "Rhode Island"
m_colStates.Add "12-South Carolina", "South Carolina"
m_colStates.Add "83-Virginia", "Virginia"

End Sub
```

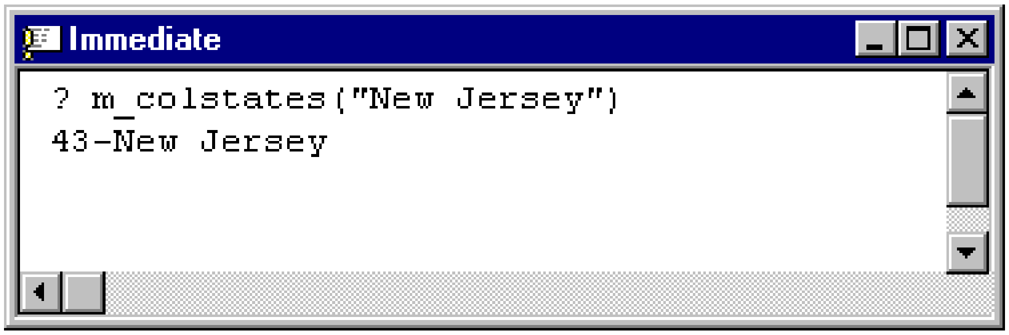
Notice the addition of the key argument, following the item value.

Watch what happens if we run this program, click on the first command button, pause the program and then type this statement into the Immediate Window…



```
? m_colstates(3)
12-Georgia
```

m_colStates(3) tells Visual Basic that we want to reference the third element of the Collection m_colStates. In this example we are referring to the Collection item by Index value. We can just as easily refer to it by key value, since we added the items to the Collection using the key value. Look at this…



```
? m_colstates("New Jersey")
43-New Jersey
```

As you can see, all we did was place the key value we were looking for within the parenthesis, and the value of that corresponding item was returned.

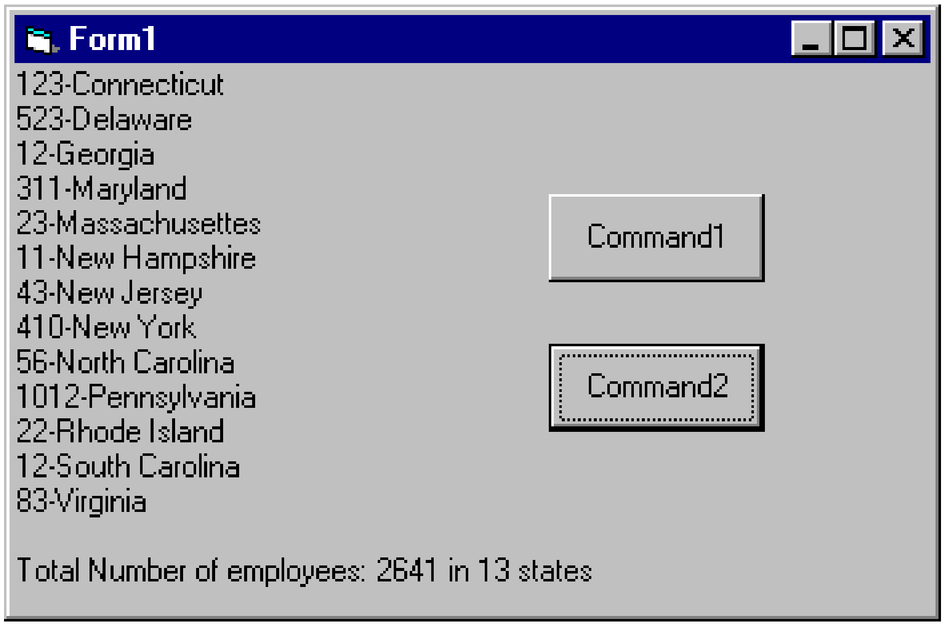
This is an extremely powerful feature, because it gives us the ability to directly access an item in a Collection---something we can't do with an Array. With an Array, if we want to find the item for the state of New Jersey, we would have to 'loop' through each element of the Array until we find it. Again, if there are many elements in the Array, this can be very time consuming. With a Collection, we can obtain the value of the Collection item immediately.

As was the case with the Forms and Controls Collection, we can use the For..Each statement to 'loop' through each item in our own Collection. This code, in the Click Event Procedure of Command2, will do exactly that…

```
Private Sub Command2_Click()

Dim obj
Dim intTotal As Integer

For Each obj In m_colStates
  intTotal = intTotal + Val(obj)
  Form1.Print obj
Next

Form1.Print
Form1.Print "Total Number of employees: " & intTotal & " in " &
m_colStates.Count & " states"

End Sub
```

What we're doing here is using the For…Each statement to loop through each of the items in the Collection. The last line of code also illustrates how we can use the one and only property of a Collection---the Count property---to display the number of elements in the Collection.

If we now run the program, click on the first command button to load the Collection, and then click on the second command button to display it---the following screenshot will be displayed.



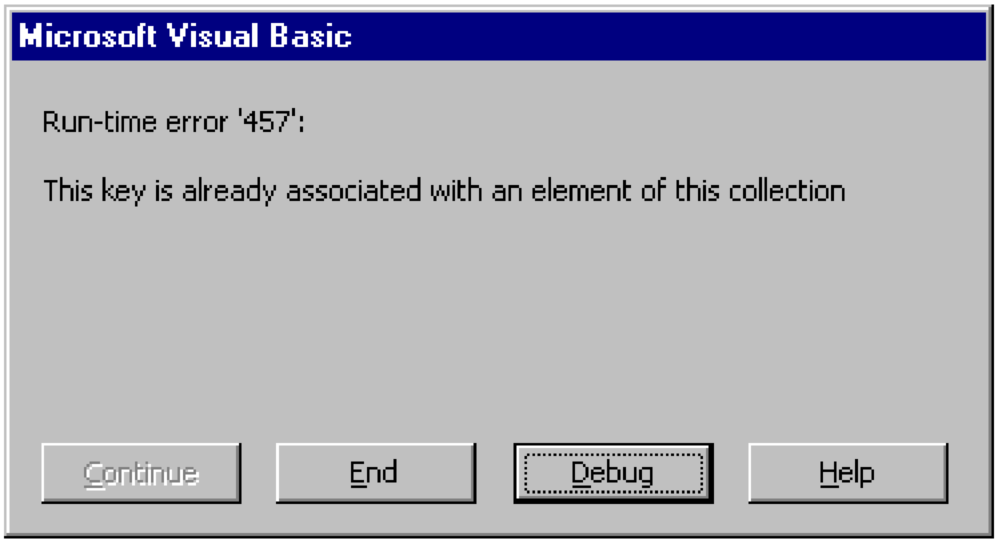
```
123-Connecticut
523-Delaware
12-Georgia
311-Maryland
23-Massachusettes
11-New Hampshire
43-New Jersey
410-New York
56-North Carolina
1012-Pennsylvania
22-Rhode Island
12-South Carolina
83-Virginia

Total Number of employees: 2641 in 13 states
```

This is very much like the Array example earlier. The difference is that using a Collection is much more Object oriented---items are added by executing a method---the Add Method. The count of the number of elements in the Collection is obtained by interrogating a property---the Count property.

Earlier I alluded to the fact that if you try to add an item with a duplicate key, Visual Basic will prevent it. Don't be confused here---Visual Basic doesn't care if you try to add an item with a duplicate value---such as '410-New York'. Visual Basic only cares if you attempt to add an item with a duplicate key---like 'New York'. Let me show you. If we modify the code in the Click Event Procedure of Command1 to look like this….(changed code is highlighted)

```
m_colStates.Add "123-Connecticut", "Connecticut"
|m_colStates.Add "523-Delaware", "Delaware"
m_colStates.Add "12-Georgia", "Georgia"
m_colStates.Add "311-Maryland", "Maryland"
m_colStates.Add "23-Massachusetts", "Massachusetts"
m_colStates.Add "11-New Hampshire", "New Hampshire"
m_colStates.Add "43-New Jersey", "New Jersey"
m_colStates.Add "410-New York", "New York"
m_colStates.Add "56-North Carolina", "North Carolina"
m_colStates.Add "1012-Pennsylvania", "Pennsylvania"
m_colStates.Add "22-Rhode Island", "Rhode Island"
m_colStates.Add "12-South Carolina", "South Carolina"
m_colStates.Add "83-Virginia", "Virginia"
m_colStates.Add "Testing1-2-3", "New York"
```

Notice how the last statement is attempting to add an item with a key value that already exists in the Collection. Watch what happens when we run the program, and click on Command1.


Microsoft Visual Basic

Run-time error '457':

This key is already associated with an element of this collection

This error message is unusually self explanatory for Visual Basic---it's telling us that the key associated with this element already exists in the Collection---something that isn't permitted.

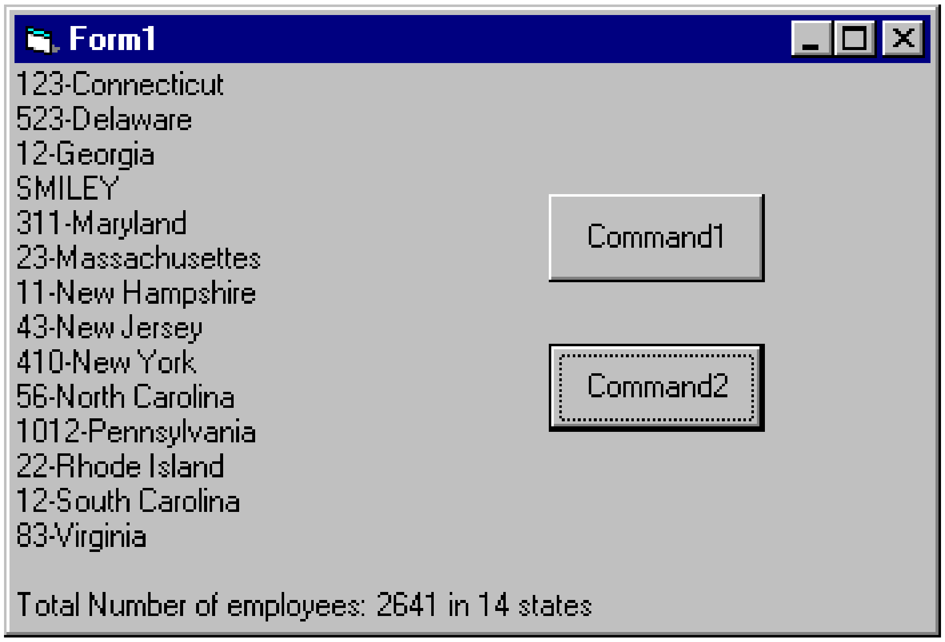
I'd like to take a moment now to discuss the other arguments to the Add Method---Before and After.

The before and after arguments allow you to specify the location of an item in your Collection as you are adding it. For instance, suppose after you have loaded the elements of your Collection, that you decide to add another item to your Collection---and for whatever reason, you want to physically position it before or after another item in the Collection---for instance, after existing item number 3 but before existing item number 4.

No problem! You can specify either the before or after argument of the Add method to do exactly that (the choice is up to you). Furthermore, you can specify as the value for the before or after argument either a numeric index value or an existing key value. For instance, this code…

```
Private Sub Command1_Click()

m_colStates.Add "123-Connecticut", "Connecticut"
m_colStates.Add "523-Delaware", "Delaware"
m_colStates.Add "12-Georgia", "Georgia"
m_colStates.Add "311-Maryland", "Maryland"
m_colStates.Add "23-Massachusetts", "Massachusetts"
m_colStates.Add "11-New Hampshire", "New Hampshire"
m_colStates.Add "43-New Jersey", "New Jersey"
m_colStates.Add "410-New York", "New York"
m_colStates.Add "56-North Carolina", "North Carolina"
m_colStates.Add "1012-Pennsylvania", "Pennsylvania"
m_colStates.Add "22-Rhode Island", "Rhode Island"
m_colStates.Add "12-South Carolina", "South Carolina"
m_colStates.Add "83-Virginia", "Virginia"
m_colStates.Add "SMILEY", "SMILEY", "MARYLAND"

End Sub
```

is instructing Visual Basic to add an item to the Collection with a value of 'SMILEY' and to place that item *before* the item in the array that contains the key value of 'MARYLAND'. If we run this program now, click on the first command button to load the Collection, and then click on the second command button to display it---the following screenshot will be displayed.

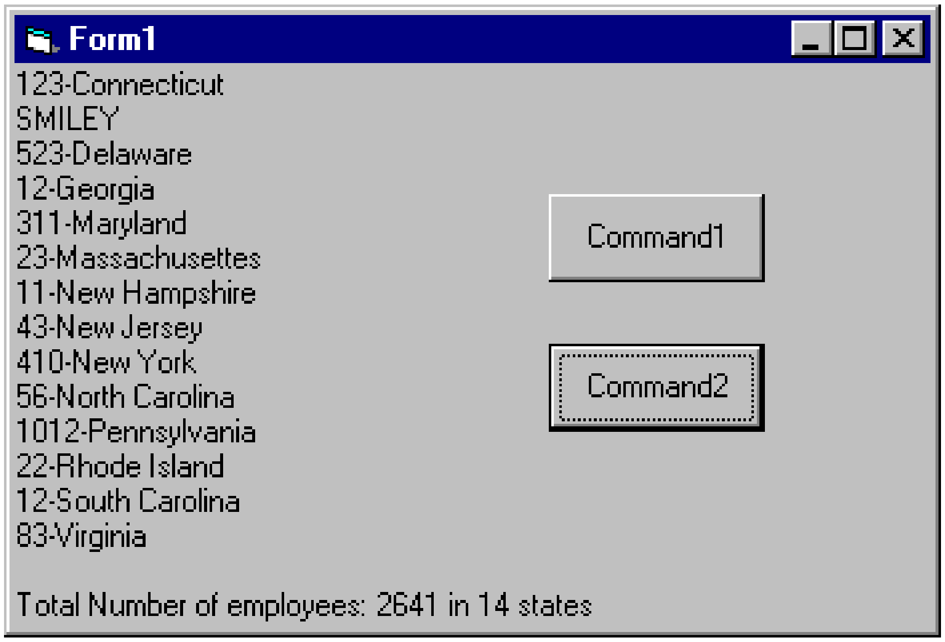


```
123-Connecticut
523-Delaware
12-Georgia
SMILEY
311-Maryland
23-Massachusettes
11-New Hampshire
43-New Jersey
410-New York
56-North Carolina
1012-Pennsylvania
22-Rhode Island
12-South Carolina
83-Virginia

Total Number of employees: 2641 in 14 states
```

Likewise, if we decide we want to place an item *after* an existing item in the Collection, we can specify a value for the after argument. As was the case with the before argument, the value for the after argument may be either a numeric index or the key value of an existing item in the Collection. For instance, if we wanted to place an item after the first item in the Collection, this code will do it…

```
Private Sub Command1_Click()

m_colStates.Add "123-Connecticut", "Connecticut"
m_colStates.Add "523-Delaware", "Delaware"
m_colStates.Add "12-Georgia", "Georgia"
m_colStates.Add "311-Maryland", "Maryland"
m_colStates.Add "23-Massachusetts", "Massachusetts"
m_colStates.Add "11-New Hampshire", "New Hampshire"
m_colStates.Add "43-New Jersey", "New Jersey"
m_colStates.Add "410-New York", "New York"
m_colStates.Add "56-North Carolina", "North Carolina"
m_colStates.Add "1012-Pennsylvania", "Pennsylvania"
m_colStates.Add "22-Rhode Island", "Rhode Island"
m_colStates.Add "12-South Carolina", "South Carolina"
m_colStates.Add "83-Virginia", "Virginia"
m_colStates.Add "SMILEY", "SMILEY", , 1

End Sub
```

If we now run the program, click on the first command button to load the Collection, and then click on the second command button to display it---the following screenshot will be displayed.



```
123-Connecticut
SMILEY
523-Delaware
12-Georgia
311-Maryland
23-Massachusettes
11-New Hampshire
43-New Jersey
410-New York
56-North Carolina
1012-Pennsylvania
22-Rhode Island
12-South Carolina
83-Virginia

Total Number of employees: 2641 in 14 states
```

As promised, the value 'SMILEY' has been added *after* the first item in the Collection.

Some of you may be wondering why this line of code

```
m_colStates.Add "SMILEY", "SMILEY", , 1
```

has an extra comma in the argument list?

I call that a comma placeholder---and it's there because we are passing only one of the two optional arguments to the Add method here---we can't pass both the before and the after argument. Since the after argument comes last in the argument list, we need to specify a comma placeholder to 'mark' its place.

For those of you who find optional arguments and comma placeholders confusing, I'd like to remind you that in most cases you can supply arguments (both required and optional) with Names. When you look up a method, procedure or function in Visual Basic help, if you see the words 'Named Argument' then you know you can pass Named arguments to that Method, Procedure or Function . For instance, we could have written the code to look like this---using Named argument.

```
Private Sub Command1_Click()

m_colStates.Add Item:="123-Connecticut", Key:="Connecticut"
m_colStates.Add Item:="523-Delaware", Key:="Delaware"
m_colStates.Add Item:="12-Georgia", Key:="Georgia"
m_colStates.Add Item:="311-Maryland", Key:="Maryland"
m_colStates.Add Item:="23-Massachusetts", Key:="Massachusetts"
m_colStates.Add Item:="11-New Hampshire", Key:="New Hampshire"
m_colStates.Add Item:="43-New Jersey", Key:="New Jersey"
m_colStates.Add Item:="410-New York", Key:="New York"
m_colStates.Add Item:="56-North Carolina", Key:="North Carolina"
m_colStates.Add Item:="1012-Pennsylvania", Key:="Pennsylvania"
m_colStates.Add Item:="22-Rhode Island", Key:="Rhode Island"
m_colStates.Add Item:="12-South Carolina", Key:="South Carolina"
m_colStates.Add Item:="83-Virginia", Key:="Virginia"
m_colStates.Add Item:="SMILEY", Key:="SMILEY", After:=1

End Sub
```

Hopefully that's a little easier on you.

The Remove Method

Let's not forget to talk about the Remove method of a Collection. Here's the syntax for the Remove Method.

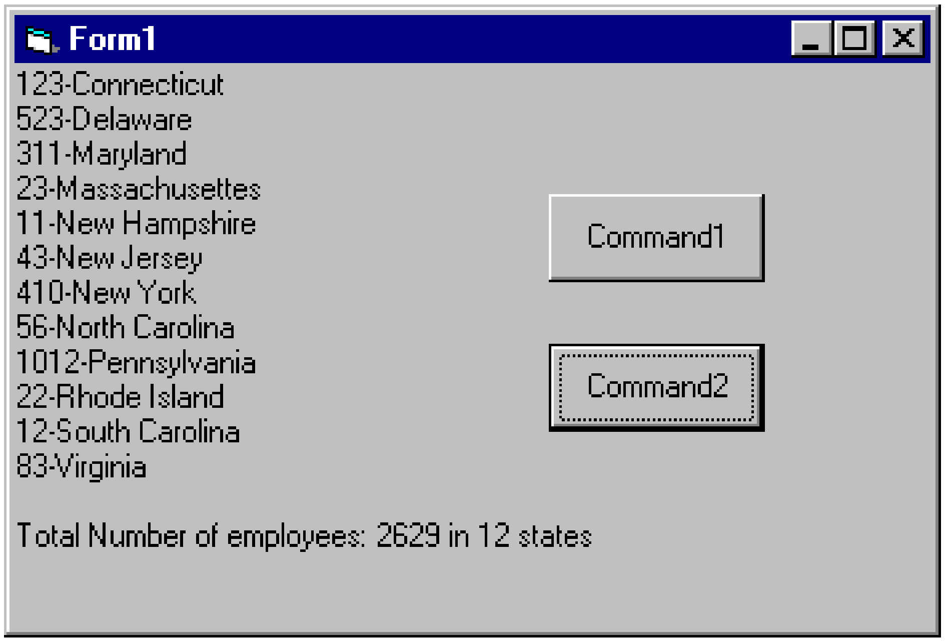
object*.Remove *index

As you can see, there's not much to removing an item from a Collection Object. All you need to do is reference either the item's Index number or key value. Let's again change the Click Event Procedure of Command1 to look like this (changed code is highlighted)

```
Private Sub Command1_Click()
```

```
m_colStates.Add Item:="123-Connecticut", Key:="Connecticut"
m_colStates.Add Item:="523-Delaware", Key:="Delaware"
m_colStates.Add Item:="12-Georgia", Key:="Georgia"
m_colStates.Add Item:="311-Maryland", Key:="Maryland"
m_colStates.Add Item:="23-Massachusetts", Key:="Massachusetts"
m_colStates.Add Item:="11-New Hampshire", Key:="New Hampshire"
m_colStates.Add Item:="43-New Jersey", Key:="New Jersey"
m_colStates.Add Item:="410-New York", Key:="New York"
m_colStates.Add Item:="56-North Carolina", Key:="North Carolina"
m_colStates.Add Item:="1012-Pennsylvania", Key:="Pennsylvania"
m_colStates.Add Item:="22-Rhode Island", Key:="Rhode Island"
m_colStates.Add Item:="12-South Carolina", Key:="South Carolina"
m_colStates.Add Item:="83-Virginia", Key:="Virginia"
m_colStates.Remove 3

End Sub
```

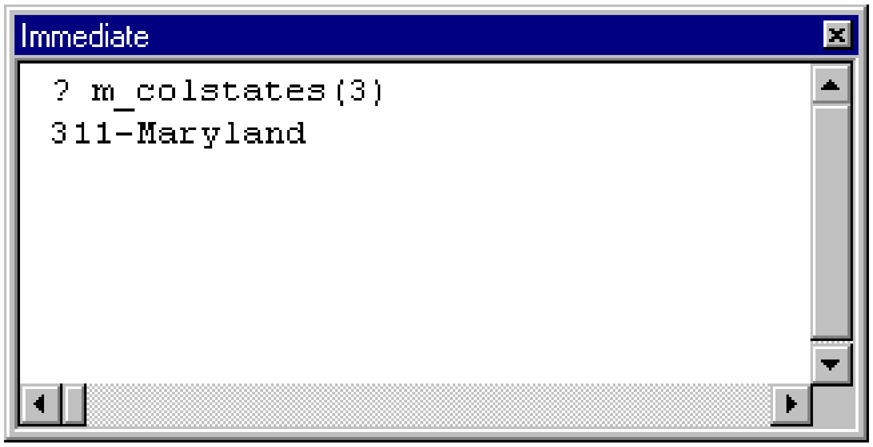
If we now run the program, and click on the first command button to load the Collection---plus remove the third item---and then click on the second command button to display it---the following screenshot will be displayed.



```
123-Connecticut
523-Delaware
311-Maryland
23-Massachusettes
11-New Hampshire
43-New Jersey
410-New York
56-North Carolina
1012-Pennsylvania
22-Rhode Island
12-South Carolina
83-Virginia

Total Number of employees: 2629 in 12 states
```

As you can see, the third item in the Collection, '12-Georgia' has been removed from the Collection. Notice also that the total number of employees and states has changed. And just in case you're wondering, when we removed the third item--Georgia from the Collection, what happened to the third position in the Collection. Is it empty, or did every other item in the Collection move 'up'?

Let's see by typing the following statement into the Immediate Window…



```
? m_colstates(3)
311-Maryland
```

What this means is that with a Collection that has a fair amount of additions and deletions, we shouldn't count on an item being in the same position to which we added it. Items in a Collection will shuffle as additions and deletions are made. That makes adding items to a Collection using the key value even more compelling. The key value can never change, so provided you don't forget it, you can always find an item in a Collection using it. Plus, don't forget that using a Key prevents items being added to the Collection with duplicate keys---an important feature.

Speaking of keys, I want to emphasize that once an item is added to a Collection using a key---try as you might, there is no way to display either the Index or the key value of an item in the Collection. Once added, both the Index and Key are hidden from us.

That also means that once an item is added to a Collection, the item's value cannot be changed---to do that, you would need to remove the item and then add it back with a different value.

Summary

Working with Visual Basic System Collections, and creating your own Collections can provide your program with a tremendous amount of flexibility.