How to…
Write applications using Visual Basic

Last month, we finished the Units application by adding the supporting code required to manage the combo box we added. This month, I'll be expanding on the methods, properties and events that you've learnt so far.

Common Features

The various controls in VB have a fair number of properties, methods and events in common. This commonality is present because it reduces the learning curve and makes the process of learning new controls easier. We have already used the *Caption* property of the *CommandButton* and *Label* controls to determine what descriptive text will appear on them. You might not be surprised to learn that the same property serves exactly the same purpose in the *CheckBox*, *Frame* and *OptionButton* controls as well. This familiarity makes Visual Basic relatively easy to use because in many cases, you can begin using an unfamiliar control without needing to learn an entirely new set of properties.

The following properties, methods and events are present in most VB controls:

Common Properties

Name
This is perhaps the simplest property in VB ; as you already know, it represents the name that you use to refer to a control in your program.

BackColor & ForeColor
These two properties refer to the background and foreground colours that Windows will use to draw the control. The foreground colour is the "main" colour of a control; for example, changing the *ForeColor* property of a Label control will change the colour of its text. The background colour normally refers to the blank area around the control and its text. For example, changing the *BackColor* property of a Label control to be red would cause the region covered by the Label to become red, with the exception of the label text itself.

Font
This property reflects the font that will be used to draw the control's text. To change the font of a control, click its *Font* property and then click the three dots (…) that appear by its side. You can then choose an alternative font from the standard Windows font selection dialog.

Enabled
This property determines whether or not the user can interact with a control. It is useful for preventing the user from choosing something inappropriate at a particular time. We used the *Enabled* property in our Units application to stop the user clicking the "Convert" button when no "from" units were given. Controls that have their *Enabled* properties set to *false* have a greyed-out appearance to indicate that they are not available.

Visible
As you might have guessed, setting this property to *true* causes the control to be visible, setting it to *false* makes the control invisible.

Caption
You've met this one before – it determines the descriptive text that is displayed on controls that use this property.

Text
This property represents the text in a control that the user can change, for example, the contents of a TextBox. We used this property to retrieve the user-supplied "from" units in our Units application. A control that has a *Caption* property doesn't have a *Text* property, and vice-versa.

Left, Width, Top & Height
These represent the placement of a control inside its *container*, which in our case, has always been a Form. However, some controls are containers themselves. For example, the *Frame* control is a container, whose sole purpose in life is to act as a holding place for other controls. The *Left* and *Top* properties represent how far a control is from the left-hand and top edges of its container respectively. The *Width* and *Height* properties are self-explanatory. These properties are usually measured in units known as *Twips*. A twip is a display-independent unit of measurement - there are normally fifteen twips to every dot (pixel) on your screen. That is, moving a control right by fifteen twips will move the control one pixel to the right. If you don't like referring to dimensions in terms of twips, you can change the unit of measurement to something more meaningful like pixels or centimetres. This is achieved by changing a form's *ScaleMode* property, which will be explained in a future tutorial.

Tag
This property has no meaning to Visual Basic – it is provided for you to use as you see fit. It provides a convenient place in which to place your own information about a control.

Common Methods

SetFocus
As explained in previous tutorials, your keypresses are directed to only one control at a time. The control that receives the keyboard input is said to have the *focus*. You can tell which control has the focus by looking at it - if a CommandButton has the focus, a dotted

box is drawn around its caption. If a TextBox has the focus, the caret (the vertical flashing bar) is displayed within it, and so on. The focus is normally moved between controls either by tabbing between them using tab and shift-tab, or by clicking on them. However, you can manually set the focus in a VB program by using the *SetFocus* method on the control you want to receive the focus. This is useful if the user enters some inappropriate value into a textbox for example, since you can direct them back to the control using *SetFocus* to allow the user to correct the mistake.

Move

This is a quick way of repositioning or resizing a control without setting its Left, Top, Width and Height properties individually. To use this, you use

```
<Control>.Move <Left>, [Top], [Width], [Height]
```

where `<Control>` is the control you want to move, `<Left>` is the new value for its *Left* property and the rest are optional arguments which will be assigned to their respective properties. For example:

```
txtName.Left = 100
txtName.Top = 150
```

has the same effect as

```
txName.Move 100,150
```

Refresh

This method forces VB to redraw the control to which you apply this method. Under normal circumstances, VB only redraws controls when it is *not* busy running some code, i.e. when it is waiting for the user to move the mouse or click a button, and so on. This means that if you change the appearance of a control in some way and then run some code immediately afterwards, you won't see the change in the control's appearance until the code is finished.

Not refreshing enough?

To see this in action, start a new project and place a *CommandButton* and a *Label* on the default form that VB provides. Don't worry about their names for now. Double-click on the *CommandButton* and enter the following code:

```
Dim intCount As Integer
For intCount = 1 to 1500
   Label1 = intCount
Next
```

Please note that the *Caption* property of a Label is its default property and therefore, I used `Label1 = intCount` rather than `Label1.Caption = intCount`.

Okay, before you run the program, what do you think will happen when *CommandButton1* is clicked? You'd expect to see *Label1* count upwards from 1 to 1500 rather quickly (the exact speed depends on how fast your PC is). Try running the program and seeing what happens. Didn't work as you expected, did it? VB just paused for a brief moment and then displayed "1500" on the label control.

Does VB cheat?

Is VB cheating since it already knows what the result will be? Well, no - in fact, VB isn't redrawing the label control until the For..Next loop finishes because it is busy running the code in the loop. Once the loop finishes, VB gets some free time whilst it waits for you to do something, and therefore, it redraws the label control. Since the *Caption* property at that time is set to 1500, that's all you see.

We can force VB to refresh (update) the control by placing the line

```
Label1.Refresh
```

immediately before the *Next* statement. This explicitly tells VB that you want the control to be redrawn right away, rather than waiting until it gets some free time on its hands. If you add the line and run the program again, you'll see that the label control counts upwards as expected.

Please note that any new values you assign to properties *do* make it to their controls immediately, it's just that the display won't be updated unless VB gets some free time, or you explicitly ask for a control to be refreshed via the *Refresh* method.

Common Events

Click

We've already encountered the *Click* event in our Units application. We wrote an event handler for the Click event on the "Convert" *CommandButton* to perform the conversion whenever the user clicked on the button. As you'll have already guessed, the Click event occurs whenever the user clicks on a control. More precisely defined, a Click event occurs on a control when the user presses and then releases a mouse button whilst the mouse is still over that same control.

DblClick

This event occurs whenever the user double-clicks on a control, i.e. presses and releases the mouse twice in quick succession whilst it is over a control.

MouseDown

The *MouseDown* event occurs as soon as the user presses a mouse button whilst the mouse is over a control. Unlike the *Click* event, this event runs as soon as the button is pressed, i.e. it doesn't wait for the button to be released. VB provides you with additional information when using the *MouseDown* event, such as which mouse button was pressed,

whereabouts it was pressed and whether or not Shift, Alt or Control were being held down at that time.

MouseUp
This is similar to the *MouseDown* event, except that it occurs when the mouse is released instead. A control that receives a *MouseDown* event will also receive the corresponding *MouseUp* event, even if the mouse isn't over the control that received the initial *MouseDown* event in the first place. For example, if you press a mouse button whilst over a *Label* control and then keeping the button held down, move the mouse over a *TextBox* control and then release the mouse button, the *TextBox* control will *not* receive a *MouseUp* event. The *Label* control receives it instead since it also received the initial *MouseDown* event.

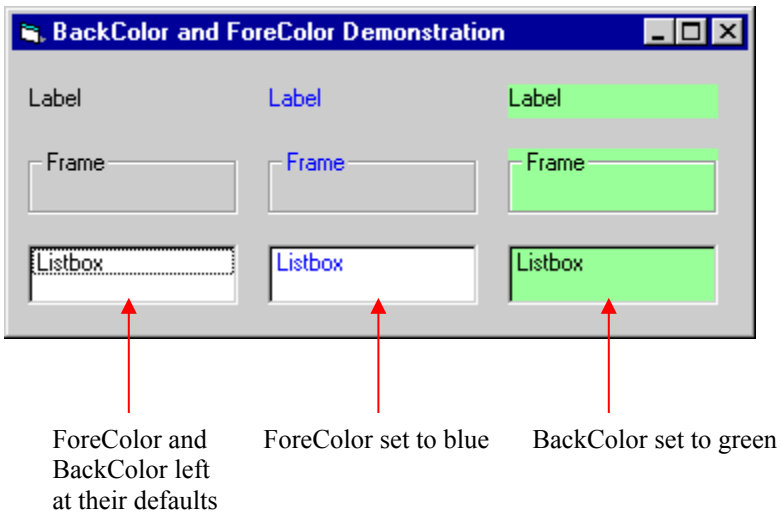
In Closing

Well, that's all for now. As usual, if you want to know what so-and-so does, then try it! That's the beauty of VB; many things are self-explanatory. For example, I haven't explained what causes the *MouseMove* event to occur yet, but I would think you've already guessed correctly. Until next time,

Cheers,
Nick.

Nicholas Scott is a freelance columnist who currently works for MIS Computer Services in Northwich. Nick can be contacted via email at nicks@miscs.com.

(ED: The filename for this image is Colours.bmp)



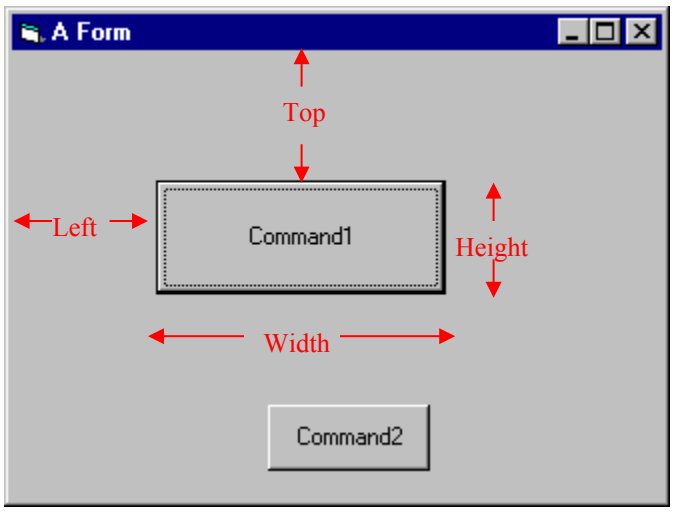
ForeColor and BackColor left at their defaults

ForeColor set to blue

BackColor set to green

The effects of changing the *ForeColor* and *BackColor* properties.

(ED: The filename for this image is Sizes.bmp)



The *Left*, *Top*, *Width* and *Height* properties, and the dimensions that they represent. Notice that *Command1* has got the *focus* in this case, hence the dotted rectangle that surrounds its caption.