How to…
Write applications using Visual Basic

Last time, I introduced you to some of the properties, methods and events that most VB controls have in common. I'll finish explaining the common events and then we'll put some of what you've learnt into practice by writing our own simple art program.

Common Events continued…

MouseMove
This event occurs whenever the mouse is moved over a control. As with the *MouseDown* event, you are provided with extra information, such as whereabouts the mouse was moved to, which buttons were being held down, and whether or not Shift, Alt or Control were being held down at that time. This event is useful for keeping track of whereabouts the mouse is being moved to – we will use this in our art program later on.

GotFocus
This event occurs whenever a control receives the focus, i.e. whenever a control is ready to receive input from the keyboard. The *GotFocus* event is useful for providing default values, such as when the user navigates to a *TextBox* for example. This event can only occur on controls that have their *Enabled* and *Visible* properties set to *True*, i.e. they are visible and available for interaction with the user. If you explicitly move the focus to a particular control via the *SetFocus* method, that control will receive a *GotFocus* event as expected. Refer to the article in the previous issue for more information about the focus.

LostFocus
As you might have guessed, the *LostFocus* event runs whenever a control loses the focus. This event provides an ideal opportunity to validate the information that a user has typed into a control such as a *TextBox*. Manually setting the focus to another control via the *SetFocus* method causes the control that previously had the focus to raise its *LostFocus* event. As with the GotFocus event, this event can only occur on controls that have their *Enabled* and *Visible* properties set to *True.*

Making a splash

That's enough theory for now, let's put our newly acquired knowledge into action by writing a simple art program. This program will use some of the events we've just learnt about and you'll also encounter some controls that you have not dealt with before.

Start by creating a new VB project (Standard EXE). Then, add the required controls to the default form VB provides so that the form looks like the form in Figure 1. Leave the various properties at their defaults, you'll be changing them in a moment.

Once you've laid out the form and its controls as shown, set the properties of the controls to the values indicated in Figure 2.

Held in containment

Now we're ready to add the *OptionButton* controls. However, we want these controls to reside *inside* our frame control, *fraShape*. Certain VB controls such as the *Frame* and *PictureBox* can act as a holding place for other controls. Controls having this encompassing ability are known as *containers* since they can contain other controls. In order to tell VB that you want one control to belong inside another one, you must create the control inside its container. Therefore, when you create the first *OptionButton*, begin to drag the initial rectangle within the boundaries of the frame control.

Add the first *OptionButton* by dragging its rectangle out inside the frame. Then, resize the control so that it is about two grid units high and about two-thirds as wide as the frame that contains it. Position the newly-created option button in the upper left-hand corner of the frame control since we're going to be adding another three controls below it. Finally, change the *OptionButton's Name* property to *optShape* and it's *Caption* property to *Circle*.

The one who is many

Now we're ready to add the other three. I suppose you're wondering why we didn't call the option button *optCircle* to match its *Caption*. The answer is, I want all four option buttons to be known under the collective name *optShape*. Of course, you can't have four different controls all having identical names. The solution is to make the controls members of something known as a *control array*. This lets you refer to many controls using a common name, but you specify which particular control you want to deal with by specifying its position (index) within the control array. Control arrays do for controls what normal arrays do for variables.

Referring back to the *Units* program we wrote earlier in this series, we used the array *masngMultiplicationFactors* to store the various multiplication factors under a common name - *masngMultiplicationFactors (0)* referred to the first multiplication factor, *masngMultiplicationFactors (1)* referred to the second multiplication factor, and so on. Similarly, we'll be able to refer to the first shape-related *OptionButton* as *optShape(0)*, the second as *optShape(1)* and so on. As with a normal array, the type of item used within the array must be consistent throughout the entire array. That is, you can't have a control array consisting of four *ComandButtons* and three *TextBoxes* – every control in the control array must be the same type of control, in our case, an option button.

To make controls members of a control array, you keep their names identical but assign unique numbers to the *Index* property of each control within the array. Controls that don't have any value at all in their *Index* properties (not even zero) are not regarded as part of any control array. You can manually set the *Index* property of each control, or, if you create a control with the same name as an existing control, VB will offer to do this task for you. Please note that you don't declare control arrays as you do variable arrays,

you just create the controls, setting their *Index* properties and VB accepts that they're part of a control array.

Cloning

We'll get VB to do the boring bit by coaxing it into assigning the *Index* property values for us. Copy *optShape* to the clipboard by clicking on it and then choosing *Copy* from the *Edit* menu. Notice that VB has selected the form for us (look at the handles around its edges). Next, click on *fraShape* and then choose *Paste* from the *Edit* menu. Clicking on the frame immediately before pasting a copy of *optShape* back is very important. Had we not done so, VB would have created the option button on the *form* instead of *inside* the frame since the form was selected at the time we did the paste. VB will inform you that you already have a control called *optShape* and will ask you whether or not you want to create a control array. Choose *Yes* and then look at the *Index* property of both option buttons. Notice that VB has given the original control an index of 0, and the new control an index of 1. Drag the newly-cloned control underneath the first, and change its caption to Square. Click the frame again and paste yet another copy of the control inside the frame. Notice that VB didn't ask you if you wanted a control array to be created since one already exists with the name *optShape*. Place this control below the second and set its *Caption* property to *Horizontal Line*. Once again, click the frame and choose *Paste* from the *Edit* menu. Position the new option button underneath the third and set its *Caption* property to *Vertical Line*. Your form should now look like Figure 3.

Please note that control arrays do not *have* to sit inside container controls such a *Frame*, it just so happens that placing a control array of *OptionButton* controls inside a *Frame* suited this particular application.

Try dragging the frame around the form - notice that the controls it contains follow the frame around since it contains them. Place the frame back where it belongs.

But why a control array?

Why are we using a control array in the first place? The answer is, that every member control of a control array calls the same event handler for a particular type of event. Even though the properties of each control in a control array might be different from one control to the next, all the controls within the array share one *Click* event handler, one *DblClick* event handler, and so on. We will be using a module-level variable to keep track of which option button is chosen at any one time. Rather than writing a separate *Click* event handler for each of the four option buttons, we can just place one piece of code on the centralised *Click* event handler, which will serve all four option buttons in the control array. Doing this saves you the bother of writing four almost identical routines. Event handlers for control arrays receive an extra parameter called *Index*, which represents the *Index* property of the control within the array that raised the event.

Handling the unpredictable

Control arrays have another trick hidden up their sleeve; they let you create controls on-the-fly at run-time. This is useful when you can't predict how many of a particular type of control you will want on a form at design time. For example, let's say that you create a form that lets you choose a particular date from the current month using some option buttons, one for each day of the month. How many option buttons would you place on the form? You could place thirty-one option buttons onto your form and then hide the option buttons that you weren't using, for example in February when only twenty-eight might be required. However, this isn't a particularly nice solution and I can think of more interesting things to do than to manually create and place thirty-one almost identical option buttons. Instead, you could create one option button and make it the only member of a control array by setting its *Index* property to 0. Then, you could write some code on the *Form_Load* event that told VB to create the rest of the controls for you, numbering them as it went. We won't be using this "create on-the-fly" feature of control arrays in this application, but it's worthwhile knowing that control arrays support this useful feature

In Closing

That's all for this month – as usual, you can find the project files that accompany this tutorial on the cover disc. Next month, we'll continue with this application and I'll show you how to use the new controls you've just placed onto your form.

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 "Controls_left_at_defaults.bmp")

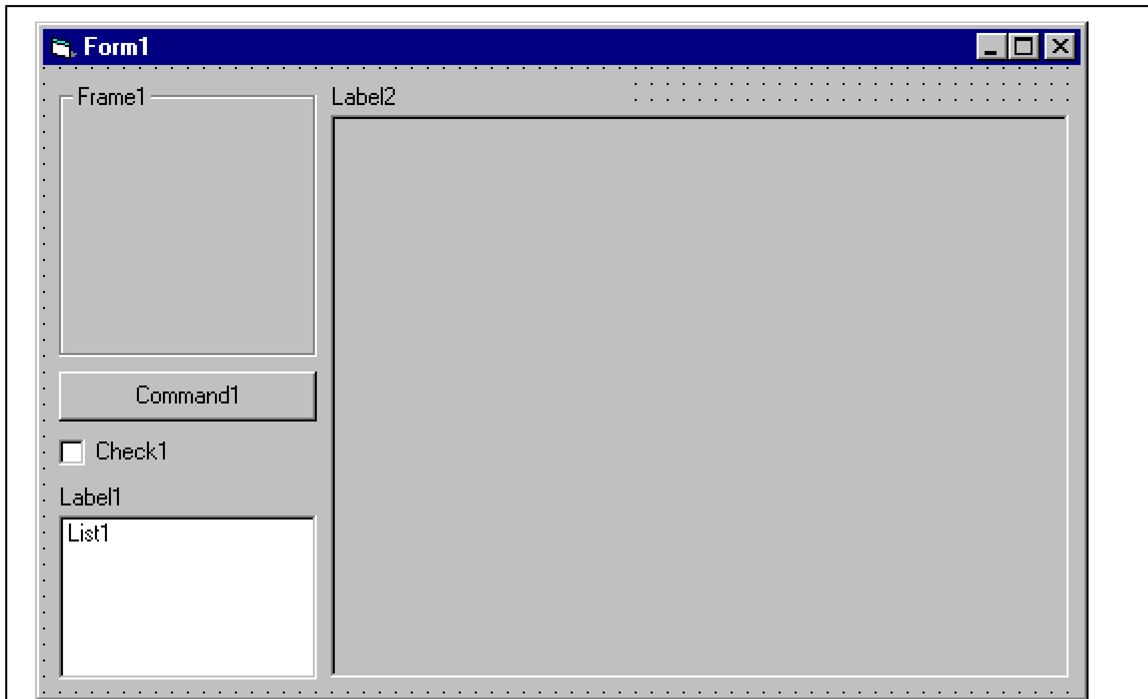

Figure 1 – The form with its controls added. Arrange the controls on your form so that they appear similar to this. It should be obvious as to which types of controls have been used because of their names, with the exception of the large control underneath "Label2" which is a PictureBox control.

Target Object	Property	New Value
Project1	Name	Doodler
Form1	Name	frmDoodler
	BorderStyle	1 – Fixed Single
	Caption	Doodler
Frame1	Name	fraShape
	Caption	Shape
CommandButton1	Name	cmdClear
	Caption	Clear
Check1	Name	chkContinuousDrawing
	Caption	Continuous Drawing
Label1	Caption	Colour
List1	Name	lstColours
Label2	Caption	Drawing Area
Picture1	Name	picDrawingArea
	AutoRedraw	True
	BackColor	&H00FFFFFF&

Figure 2 – Change the properties of the controls to the new values as indicated.

(ED: The filename for this image is “Controls_with_new_properties.bmp”)

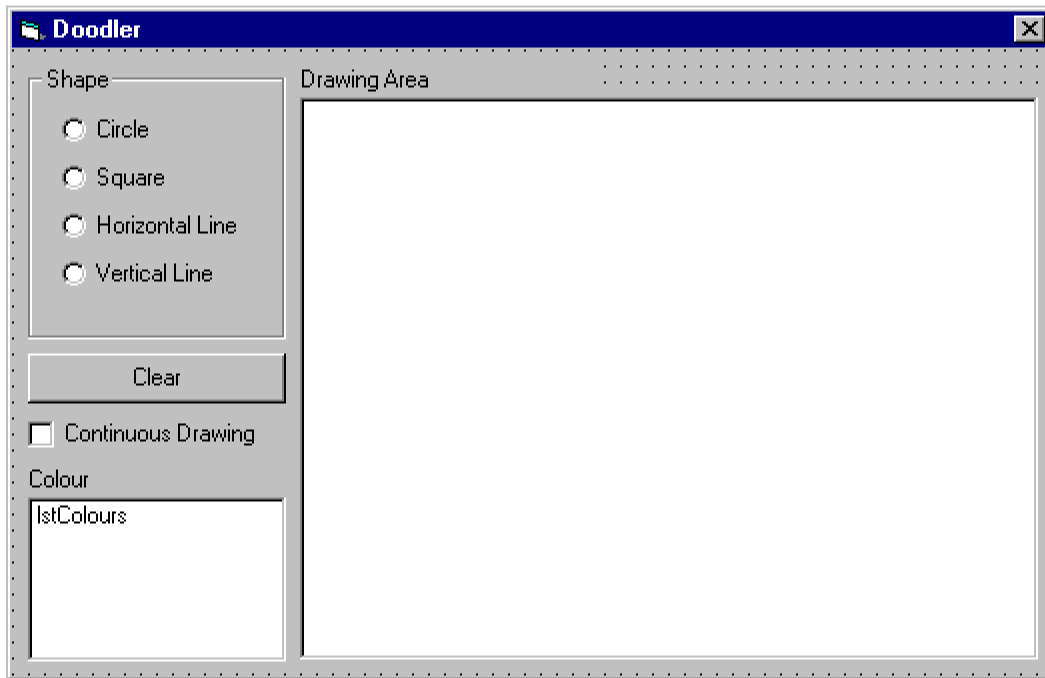

Figure 3 – Your form should end up looking something like this once you’ve changed the properties as indicated. Remember to create those option buttons *inside* the “Shape” frame.

Tip
If you accidentally create a control outside of its intended container, simply dragging the new control over the top of the container won't place it inside the container. Instead, select the misplaced control and choose *Cut* from the *Edit* menu. Next, select the container, and then choose *Paste* from the *Edit* menu. Doing this will place the control inside the container you have just selected. To move a control outside of its container, do the opposite – select the control inside its container, cut it to the clipboard, click the form and then paste it back. The control will then be re-created outside its old container.