Moving From VB to Delphi:

A technical discussion

Introduction

In recent years, the concept of visual development in Windows has been popularised by the availability of tools that support Rapid Application Development (RAD) . However, professional developers intending to exploit the benefits of visual design very rapidly run up against the inherent limitations of first-generation interpreter-based products such as Visual Basic Pro, PowerBuilder and SQLWindows.

This document introduces the next generation approach to visual development that has been pioneered by Borland's Delphi and Delphi Client/Server, and the advantages it offers in comparison to Visual Basic Pro. Delphi is unique among development tools in that it combines the benefits of a high-performance optimising native code compiler, visual two-way rapid application development (RAD) tools and fully scalable database access. It is Delphi's underlying compiler technology, built upon more than ten years of compiler development expertise at Borland, that is the key to its significant performance lead over Visual Basic Pro.

Specifically, this document is aimed at programmers with at least intermediate knowledge and experience in Visual Basic. It assumes an awareness of GUI design concepts, component properties and the event-driven programming model. While it does not set out to provide a comprehensive tutorial on Delphi or its underlying Object Pascal language, this document does cover some of the key differences between Delphi and Visual Basic, so that Visual Basic users may more quickly get up to speed with Delphi programming.

For many concepts in Visual Basic, there are parallel facilities offered by Delphi. However, there are several significant features of Delphi such as creating reusable components or Dynamic Link Libraries (DLLs) that are simply impossible within the Visual Basic environment. This documents outlines some of the key similarities and differences in visual development between the two products.

The development environment

Both Delphi and VB feature visually oriented development environments. As in VB, you see a main window, a property inspector and a crisp new form called *Form1*. On the left side of the main window, you have a toolbar with similar functionality to the one in VB. To the right, however, you find the Component palette, to provide a better organization, and easier access, than the VB Toolbox.
The components are grouped on the palette by functionality, on a series of tabbed pages. You will discover as you delve deeper into Delphi that you can customize these pages but for now it is just important to know that you use this Component palette the same way you use the floating toolbox in VB. One difference that you will appreciate is that the Delphi palette has Help hints for the controls so that you know right away what they are.

Adding Controls to Forms

You add controls to a form in a manner similar to that of Visual Basic. You can add a control in each of the following ways:

- Click the tool, draw the control
- Click the tool, click the form
- Double-click the tool
- Shift-click the tool to create multiple controls of the same type

You can move controls around the form and resize them. You can also select multiple components via a rubber-band box or by Shift-clicking the components you want to select. On the Edit menu, you will see familiar commands to manipulate the Z-Order of the controls. In addition, there are some new component-manipulation tools such as alignment, sizing, scaling and tab order. Delphi also features an intuitive and easy-to-use Tab Order dialog box, available from the Edit menu but, if you are more familiar with VB's TabIndex property, you can also use the equivalent TabOrder property of Delphi.

Setting Properties

Once you place a control on a form, you can set various properties in Delphi's property window, called the Object Inspector. By default, this window is placed on the left hand side of the development environment. At the top of the window, there is a combo box that displays, and provides a means for you to choose, the component whose properties you are editing. The component name is on the left and the component type (or class) is on the right. Below is a list of properties available for that control at design time. Delphi's Object Inspector allows direct selection and editing of properties, rather than only at the top of the window. There are several property editors that appear in the Object Inspector. For properties such as *Caption*, for which direct user input is expected and there is no assistance available, the only property editor is a text box. For others, more sophisticated alternatives are available. For example, the *Visible* property provides a choice between True and False. The *BorderStyle* property of a form provides for a selection of valid settings. As in VB, you can double-click a property value to iterate through the available choices. Many Delphi properties, such as *Color*, *Picture*, and *Icon*, have dialog-based property editors.
One significant improvement over Visual Basic is that properties with enumerated choices, such as *BorderStyle*, display actual built-in constant values that you can use within your code. In VB, *BorderStyle* is an enumerated property from 0 to 3, the constants for which are in CONSTANT.TXT. In Delphi, you see the actual constant values in the Object Inspector (i.e. *bsDialog*, *bsNone*, *bsSingle* and *bsSizeable*) and these are always available within your code.

Properties can be objects

An important distinction between Delphi and VB is that in Delphi there are many properties that are themselves objects. An example of this in Visual Basic is the *RecordSet* property of a data control. The name of the property is *RecordSet* but the datatype, rather than being a simple type like **string** or **integer** is an object, *Dynaset*. This leads to a syntax like this in VB, where *MoveNext* is a method supported by all *Dynaset* objects:

```
Data1.RecordSet.MoveNext
```

However, VB does not support the creation of objects, so support for this type of object organization and syntax is limited. Because Delphi is a completely object-oriented environment, you can easily create your own objects. Not surprisingly, then, Delphi has a wide variety of properties that are also objects For example, the *Font* property of an object is itself an object of type *Font*. A *Font* object has its own properties: *Color*, *Height*, *Name*, *Pitch*, *Size* and *Style*. Therefore, instead of using *ForeColor*, in Delphi, you would change the color of the caption of a label like this:

```
Label1.Font.Color := clRed;
```

To accommodate this, the Object Inspector functions as an outliner. Properties with a + character in front of them can be expanded to reveal the properties of that property. Double-click on the property name and it will be expanded for a result like this:

-Font
  Color
  Height
  Name
  Pitch
  Size
  +Style

There is another data type in Delphi called a *set*, discussed in more detail below, which is a collection of Boolean values. The *Style* property of a *Font* object is a set. When expanded, the *Style* property provides a

set of choices for **bold**, *italic*, underline and strikethrough. You can always double-click a property value to see if there is a dialog associated with it. The *Font* property is a good example where the Font common dialog is used to set all of the relevant font properties of an object.

Creating Event Handlers

In Delphi, as with Visual Basic, once you add controls to a form, you create event handlers "behind" those controls, which represent the logic of your application. To edit the handler for the default event of a component, simply double-click the component. For example, double-clicking on a button component will place you in the *OnClick* event handler and double-clicking on an edit box component will place you in the *OnChange* event handler.

As a simple example, create a new Delphi project and place an edit control on the form. The edit control can be found on the "Standard" page of the Component palette. Now, double-click the Edit control and enter the single line of code as shown below:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
    Form1.Caption := Edit1.Text;    {Add this line of code}
end;
```

As you can see, with the exception of the colon before the = sign and the semicolon at the end of the line, this code is very similar to the corresponding code in Visual Basic. Now, press *F9* to run your program. As you would expect, the caption of the form changes to reflect the text typed into the edit control. Double-click the system menu to exit the application and return to design mode. One minor difference you will see is that no underscore character is added between the object name and the event name in the creation of the event procedure name.

A more significant difference you will notice is that the procedure code appears in a single full-featured code-editing window, not in a separate code window. As you create more event handlers, you will see that they are all available in this same code-editing window. Usability testing has shown that people are more comfortable with less segmented code.

The most significant difference in event handlers between Delphi and VB is that Delphi treats events in a manner very consistent with its treatment of properties. Notice that the Object Inspector has a separate page labeled "Events." It is here that you create event handlers for the selected object by switching to the Events page and selecting the event you wish to handle. At this point, you may either type in the name you would like the event handler to have or simply double-click to generate a default procedure name. The ability to choose the event procedure for an object will prove very powerful, as you'll see later.

Running your Application

The function keys for running your application and setting a breakpoint in your code are different in Delphi. Press *F9* to run your application. In addition, you can use the Run selection from the Run menu or the toolbar icon that looks like a VCR's "Play" button (). This process actually creates a stand-alone, native code .EXE file and then launches it. In other words, once you have tested your application, there is an .EXE in the project directory that you can launch from File Manager or Program Manager. This .EXE runs without any run-time interpreter DLL, and runs up to 10-20 times faster than interpreted p-code.

Exiting your application

To exit your application, you can double-click the system menu of the main form or select Program Reset from the Run menu. The function key equivalent for this is *Ctrl-F2*. To create a simple event handler that closes the main form, you could do the following in the *OnClick* event handler of a button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Form1.Close;
end;
```

Saving your Work

When you select "Save Project" from the File Menu, you are first prompted to save the form. If you enter the name you would like the form file to have (without an extension), the appropriate files will be saved for you using the .PAS file extension; the default name is UNIT1.PAS. Next, you will be prompted to name the project itself. Because Delphi projects themselves contain executable code, Delphi does not permit you to give the same name to project and one of its constituent forms. You might want to prefix your form names with “F” or a similar convention that will help you quickly and easily identify them as forms.
When you save a form, two files are created: the DFM file and the PAS file. The DFM is a binary file which contains the form layout information. The PAS file contains the code associated with that form. Any time you add a form to a project, you select it via the DFM file but both files are actually added to the project. The project is saved with the extension DPR.
From the View menu, select Project Manager to see the Project Manager window. From this window, you can choose to see the form or associated code of any form in the project. Selecting "Save Project" from the File menu when all items have been saved once will simply save all of the files in the project.

Beginning to Code

In this section, you’ll begin to look at Delphi code—just enough to get you started. Of course, there is really no substitute for learning as much as you can about the Object Pascal language and it is beyond the scope of this paper to describe the entire language. However, there are enough similarities between Object Pascal and Visual Basic that a few notes should be enough to get you started. More language elements are covered later in this chapter.

Variables

While the use of so-called "automatic variables" in BASIC is considered risky and unsafe, all Delphi variables must be explicitly declared. This is as if Option Explicit were used in VB.
To create local variables in a procedure, you need to create a **var** clause in your procedure in which to declare them. Declaration of variables takes the form of

```
Name:Type;
```

where `Name` is the name of the variable and `Type` is the variable type. The colon takes the place of the *As* keyword in VB. The result in a procedure might look something like this:

```
procedure TfrmMain.Button2Click(Sender: TObject);
var
  i:Integer;
  s,z:string;
begin

end;
```

As you can see, it is possible to create a number of variables of the same type by simply separating their names with commas.
With the exception of variant and currency, all of the Visual Basic data types are available in Delphi as this table shows:

VB	**Delphi**
Integer	*Integer*
Long	*LongInt*
Single	*Single*
Double	*Double*
String	*String*

Variables declared in the **var** section of a procedure are local to that procedure. There are, of course, several

data types available in Delphi, not found in VB such a boolean, char and byte.

Code Blocks

One thing that you will notice about Object Pascal code right away is that all code blocks are surrounded by **begin** and **end**. That is why the event handler procedures that are generated by Delphi all have a **begin..end** section. This is the main executable code of your procedure. If there is a **var** section, it must come before the **begin**.

In addition to the **begin** and **end** which surround the code of a procedure, there might be several code blocks *within* the procedure which require their own **begin** and **end** pairs. A good example of this is an **if** statement. Suppose you create a form with several buttons and a checkbox that determines the visibility of those buttons. The *OnClick* event of the checkbox might look something like this:

```
procedure TForm1.chkShowClick(Sender: TObject);
begin
  if chkShow.State = cbChecked then
    begin
       Button1.Visible := True;
       Button2.Visible := True;
       Button3.Visible := True;
    end
    else
    begin
       Button1.Visible := False;
       Button2.Visible := False;
       Button3.Visible := False;
    end;
end;
```

Don't worry too much about the structure of the **if** statement at this point but notice that there are two code blocks, each of which is surrounded by a **begin** and **end** pair.

You will also notice that Delphi uses the semicolon to find the end of a statement, rather than the actual end of a line. Therefore, you may break up a line in any way you find most readable. For example, coming from Visual Basic, you might be most comfortable looking at an **if** statement like this:

```
if chkShow.State = cbChecked then begin
 Button1.Visible := True;
 Button2.Visible := True;
 Button3.Visible := True;
end;
```

As you might imagine, a code block is defined as more than one line of code together. Therefore, a single line of code in an **if** statement doesn't require a **begin..end** block. In other words, the following code is perfectly valid, because only one line of code needs to be executed:

```
if chkShow.State = cbChecked then Button1.Visible := True;
```

Remembering to encase code blocks in **begin..end** pairs will be one of the most challenging "mental blocks" when moving from VB to Delphi.

Assignment

As you will have noticed, the assignment operator in Delphi is the := sign. This is distinguished from the comparison operator which is simply the = sign. This is like the distinction between = and == in C. Accordingly, the code to toggle the buttons above might look something like this instead:

```
procedure TForm1.chkShowClick(Sender: TObject);
begin
  Button1.Visible := chkShow.State = cbChecked;
  Button2.Visible := chkShow.State = cbChecked;
  Button3.Visible := chkShow.State = cbChecked;
end;
```

where `chkShow.State = cbChecked` is a Boolean expression, the result of which is assigned to the

Boolean property *Visible*.

Note In Delphi, you enclose string literals in single quotes, not double quotes as in VB. Therefore, to make an assignment to the caption of a form, you would do something like this:

```
Form1.Caption := 'Hello World!';
```

Commenting your code

It is appropriate to mention the topic of comments early in the discussion of any development environment. In Delphi, you create a comment by surrounding text in curly braces. This is an example of a comment:

```
{This is a comment}

{This is an example of
a multi-line comment}
```

There is no equivalent to the ' comment in VB. A comment begins with a brace and continues until the next closing brace.

Components in Delphi

There are several component types in Delphi which can be used in the design environment. You will notice when you place a component on a form that the class name for the component (which shows up in the Object Inspector) begins with a "T" for "type." This is a convention that is used in Object Pascal frequently. Therefore, the class name of the button control is *TButton*, an edit control *TEdit* and so on. The time it is most important to know this is when you are trying to get help on a component by searching for it by name. The following table is a list of Visual Basic controls and the corresponding components found in Delphi:

This VB control.....	**corresponds to this Delphi component.....**	**found on this page of the Delphi Component palette**
Image	*Timage*	Additional
Label	*Tlabel*	Standard
TextBox	*Tedit*	Standard
Frame	*TGroupBox*	Standard
CommandButton	*Tbutton*	Standard
CheckBox	*TCheckBox*	Standard
OptionButton	*TRadioButton*	Standard
ComboBox	*TComboBox*	Standard
ListBox	*TListBox*	Standard
HScrollBar	*TScrollBar*	Standard
VScrollBar	*TScrollBar*	Standard
Timer	*Ttimer*	System
DriveListBox	*TDriveComboBox*	System
DirListBox	*TDirectoryListBox*	System
FileListBox	*TFileListBox*	System
Shape	*Tshape*	Additional
OleControl	*TOleContainer*	System
Grid	*TStringGrid*	Additional
CommonDialog	*TOpenDialog*	Dialog
	TSaveDialog	Dialog
	TFontDialog	Dialog
	TColorDialog	Dialog
	TPrintDialog	Dialog
	TPrinterSetupDialog	Dialog
	TFindDialog	Dialog
	TReplaceDialog	Dialog
Gauge	*Tgauge*	Samples
Graph	*Tchart*	VBX

MMControl	*TMediaPlayer*	Additional
MaskEdBox	*TMaskEdit*	Additional
Outline	*Toutline*	Additional
SpinButton	*TSpinButton*	Samples
SSCommand	*TBitBtn*	Additional

In addition, there are several controls included with Delphi that you would need to purchase separately with Visual Basic: the SpeedButton, TabSet, Notebook, Header, Scrollbox, TabbedNotebook and Calendar, and Grid components.
What follows is a discussion of the more prominent control types and how they compare to their VB equivalents.

Forms

Forms in Delphi are very similar in function and operation to forms in Visual Basic. They both act as the center of an application and as containers for controls. The following is a list of form properties in VB and their equivalents in Delphi:

Visual Basic	**Delphi**
ActiveControl	*ActiveControl*
ActiveForm	*ActiveMDIChild*
BackColor	*Color*
BorderStyle	*BorderStyle*
Caption	
Caption	
Enabled	*Enabled*
FontBold	*Font.Style*
FontItalic	*Font.Style*
FontName	*Font.Name*
FontSize	*Font.Size*
FontStrikThru	*Font.Style*
FontUnderline	*Font.Style*
ForeColor	*Font.Color*
HDC	*Canvas**
Height	*Height*
HelpContextID	*HelpContext*
Hwnd	*Handle*
Icon	*Icon*
KeyPreview	*KeyPreview*
Left	*Left*
MDIChild	*FormStyle*
MousePointer	*Cursor*
Name	*Name*
Picture	*Picture*
ScaleHeight	*ClientHeight*
ScaleWidth	*ClientWidth*
Tag	*Tag*
Top	*Top*
Visible	*Visible*
Width	*Width*
WindowState	*WindowState*

Where there isn't a direct property relationship, there is alternative functionality such as in the case of DDE

and scaling. In addition, there are several methods that the forms have in common, as the following table shows:

Visual Basic	**Delphi**
Circle	*Canvas.Elipse, Canvas.Arc*
Hide	*Hide*
Line	*Canvas.LineTo*
Move	*SetBounds*
Point	
PrintForm	*Print*
Print	*Canvas.TextOut*
Refresh	*Refresh*
SetFocus	*SetFocus*
Show	*Show*
TextHeight	*Canvas.TextHeight*
TextWidth	*Canvas.TextHeight*
Zorder	*BringToFront, SendToBack*
(Load)	*Create*
(Unload)	*Destroy*

As you can see, there is a property of a form which is its canvas. It is on the canvas that you do your drawing. As you will see below, there is a canvas associated with every object in Delphi on which you are able to draw.
Finally, there is more overlap in the events available from a Delphi form and those available in VB.

Visual Basic	**Delphi**
Activate	*OnActivate*
Click	*OnClick*
DblClick	*OnDblClick*
Deactivate	*OnDeactivate*
DragDrop	*OnDragDrop*
DragOver	*OnDragOver*
GotFocus	*OnGotFocus*
KeyDown	*OnKeyDown*
KeyPress	*OnKeyPress*
KeyUp	*OnKeyUp*
Load	*OnCreate*
LostFocus	*OnLostFocus*
MouseDown	*OnMouseDown*
MouseMove	*OnMouseMove*
MouseUp	*OnMouseUp*
Paint	*OnPaint*
QueryUnload	*OnQueryClose*
Resize	*OnResize*
Unload	*OnDestroy*

The equivalent of Me is Self in Delphi. You can access the object that triggered an event at any point by using the *Sender* parameter. To act on the form, you would use the syntax:

```
TForm(Self).Caption := 'Hello World!';
```

Because ObjectPascal is such a strongly typed language, you need to cast a generic object variable to a specific type. In the example above, this is done by using the type as if it's a function call—for example, `TForm(Self)`. This allows access to all of the methods and properties of that object.

CommandButtons

The *TButton* component in Delphi is the equivalent of the CommandButton control type in VB. It is nearly identical in operation to the CommandButton. The CommandButton properties (which are not shared with a form) are below:

Visual Basic	**Delphi**
Cancel	*Cancel*
Default	*Default*
DragIcon	*DragCursor*
DragMode	*DragMode*
Parent	*Parent*
TabIndex	*TabOrder*
TabStop	*TabStop*

The only new method for the CommandButton is the Drag method which has an equivalent in the BeginDrag method of the *TButton* component. All of the events of the CommandButton are covered in the table for the form object.

Text Boxes

The equivalent of the VB TextBox in Delphi is the *TEdit* component. As with the button objects, the TextBox and *TEdit* have most properties and methods in common. The really significant difference is that in Delphi there are two controls which correspond to the TextBox in VB: the *TEdit* and the *TMemo* components. Here are the equivalent properties of the TextBox, as reflected in Delphi's *TEdit* and *TMemo* components:

Visual Basic	**Delphi**
Alignment	*Alignment (TMemo)*
HideSelection	*HideSelection*
MaxLength	*MaxLength*
PasswordChar	*PasswordChar*
Scrollbars	*Scrollbars (TMemo)*
SelLength	*SelLength*
SelStart	*SelStart*
SelText	*SelText*
Text	*Text*
	Lines (Tmemo)

The *Lines* property of the *TMemo* component is similar to the List property of a Listbox and allows line by line access to the contents of the control. This property is a *TStringList* object which exists in several places in Delphi and is discussed in conjunction with the *ListBox* component. There are no new methods in the TextBox and the new event, *OnChange*, is one the TextBox has in common with the *TEdit* component.

The Font.Style property is an Object Pascal "set" which is a bit flag variable but with a cleaner syntax than is available in Visual Basic. For example, the code behind the *chkBold* checkbox looks like this:

```
procedure TForm1.chkBoldClick(Sender: TObject);
begin
 if TCheckBox(Sender).State = cbChecked then
   txtDisplay.Font.Style := txtDisplay.Font.Style + [fsBold]
 else
   txtDisplay.Font.Style := txtDisplay.Font.Style - [fsBold];
end;
```

Simply by adding the appropriate constant in brackets, you add that attribute to the style and you remove it

by subtracting it.

ListBoxes

Once again, the *TListBox* component in Delphi is very similar to the ListBox control in Visual Basic. Here are the properties of a ListBox:

Visual Basic	**Delphi**
Columns	*Columns*
ItemData	*Items.Objects**
List	*Items*
ListCount	*Items.Count*
ListIndex	*ItemIndex*
MultiSelect	*MultiSelect**
NewIndex	*(Items.Add)*
Selected	*Selected**
Sorted	*Sorted*
TopIndex	*TopIndex*

* Not a direct equivalent

The primary methods a little different because they are the methods of the Items collection but the methods for the ListBox are as follows:

Visual Basic	**Delphi**
AddItem	*Items.Add*
	Items.Insert
Clear	*Clear*
RemoveItem	Items.Delete

The contents of a ListBox are in the *Items* property, which is much like the List property in VB except it is more than an array; it is also a class of type *TStringList*. This object has many interesting features but fundamentally, manipulation of the list takes place via methods of the *Items* property.
A VB example of Listbox functionality can be found in the LISTBOX.FRM example in the SAMPLES\CONTROLS\CONTROLS.MAK sample Visual Basic project. This is a good demonstration of the differences in using the *TListbox* and the VB ListBox. In the VB sample, the Add button executes the following code:

```
Sub cmdAdd_Click ()
   lstClient.AddItem txtName.Text
   txtName.Text = ""
   txtName.SetFocus
   lblDisplay.Caption = lstClient.ListCount
End Sub
```

The equivalent code in Delphi looks something like this:

```
procedure TfrmListBox.cmdAddClick(Sender: TObject);
begin
    lstClient.Items.Add (txtName.Text);
    txtName.Text := '';
    txtName.SetFocus;
    lblDisplay.Caption := IntToStr (lstClient.Items.Count);
end;
```

The first thing you should notice is that the *Add* method is on the *Items* property, not the listbox itself. A brief aside is that you must surround the parameters of both subroutines and functions in parentheses, unlike in Visual Basic where subroutines (and methods) don't use parentheses. Second, you should see that the absence of variants means that you need to explicitly convert data types for assignment. Therefore, in order to display the *Items.Count* property in the label *lblDisplay*, use the *IntToStr* function which is similar

in functionality to STR in VB.
The code behind the VB Remove button looks like this:

```
lstClient.RemoveItem lstClient.ListIndex
```

while the equivalent Delphi code looks like this:

```
lstClient.Items.Delete(lstClient.ItemIndex);
```

where again the *Delete* method is supported by the *Items* property. In both VB and Delphi, the *Clear* method is a method of the listbox object itself so the code looks identical except for the semicolon on the end of the line of Delphi code.
A major advantage of the TStringList property type is that it is compatible with lots of other properties so the following represent working statements in Delphi:

```
ListBox1.Items := Memo1.Lines;
ListBox2.Items := Screen.Fonts;
ListBox3.Items.LoadFromFile('mylist.txt');
```

PictureBoxes/Images

The *TImage* component of Delphi is equivalent to the Image control in Visual Basic except that it has the equivalent drawing functionality of a PictureBox, thereby allowing it to serve the same purpose as both controls in VB. It is a lightweight control just like the Image control and shares a number of properties in common with the Image and the PictureBox. The core display property is the Picture property which operates in much the same way in both environments.
An example of pictureboxes in VB is the SAMPLES\FIRSTAPP\BUTTERF.MAK demo shipped with Visual Basic. The timer code in Visual Basic looks like this:

```
Sub Timer1_Timer ()
    Static PickBmp As Integer
    Main.Move (Main.Left + 20) Mod ScaleWidth,_
              (Main.Top - 5 + ScaleHeight) Mod ScaleHeight
    If PickBmp Then
       Main.Picture = OpenWings.Picture
    Else
       Main.Picture = CloseWings.Picture
    End If
    PickBmp = Not PickBmp
End Sub
```

whereas the equivalent code in Delphi looks pretty much the same:

```
procedure TForm1.Timer1Timer(Sender: TObject);
const
    PickBmp:Boolean = False;
begin
    Main.SetBounds ((Main.Left + 20) Mod ClientWidth,
                    (Main.Top - 5 + ClientHeight) Mod
                     ClientHeight,Main.Width, Main.Height);
    if PickBmp = True then
        Main.Picture := OpenWings.Picture
    else
        Main.Picture := CloseWings.Picture;
    PickBmp := Not PickBmp;
end;
```

There are a couple of differences worth noting. First, the *SetBounds* method, unlike the Move method in VB has no optional parameters so you need to supply the *Width* and *Height* values. Second, there is no so-called **static** variable type in Delphi. Instead you may use a **typed consant** in its place. The point is that the assignment to the *Picture* property works just like it does in VB.
Another key difference is that instead of a *LoadPicture* function that returns a picture, the *Picture* property of a *TImage* has its own *LoadFromFile* method as demonstrated in the next section.

File Controls

Just as in Visual Basic, there are a set of file-oriented components in Delphi for the construction of browsers and customized file dialogs. These components are the *TDirectoryListBox*, *TFileListBox*, *TDriveComboBox* and *TFilterComboBox*. There is no equivalent to the FilterComboBox in VB but you have seen similar functionality in the *Filter* property of the common dialog control.

The *TDriveComboBox* is the counterpart to the DriveListBox in VB. The relevant property in both is the Drive property. The primary difference is that the *DirectoryListBox* in Delphi also has a *Drive* property for direct assignment from the *DriveComboBox*. Therefore, in place of the code

```
Sub Drive1_Change ()
   Dir1.Path = Drive1.Drive
End Sub
```

in Visual Basic, the Delphi equivalent is:

```
procedure TForm1.Drive1Change(Sender: TObject);
begin
    Dir1.Drive := Drive1.Drive;
end;
```

The *TDirectoryListBox* and *DirListBox* are the Delphi and Visual Basic components to represent a directory. Both display a similar hierarchical structure. The core property of the *TDirectoryListBox* is Directory which is the equivalent of the Path property in VB's DirListBox. So the following code in VB

```
Sub Dir1_Change ()
   file1.Path = Dir1.Path
End Sub
```

translates to the following in Delphi:

```
procedure TForm1.Dir1Change(Sender: TObject);
begin
    File1.Directory := Dir1.Directory;
end;
```

Certainly, you can see that the operation of these controls is very similar in Visual Basic and Delphi. Most of the differences you encounter are subtle changes to the object design.
Finally, there comes the *TFileListBox* control which is the counterpart to the FileListBox in Visual Basic. Again, the operation of these controls is similar but there are enough differences that they bear closer scrutiny. Here are the relevant properties of each:

Visual Basic	**Delphi**
Archive	*FileType*
FileName	*FileName*
Hidden	*FileType*
Normal	*FileType*
Path	*Directory*
Pattern	*Mask*
ReadOnly	*FileType*
System	*FileType*

As you can see, the biggest difference between the two controls is the selection of file types to display. In VB, this is a set of Boolean properties whereas in Delphi it is a set property type like the *Style* property of a *TFont* object. In the Object Inspector, you simply double-click on *FileType* to expand the component choices. In addition to the choices in VB, there is also the ability in the *TFileList* to include the directories and volume id. To change these values programmatically, you simple add or subtract the constants from the property. In other words:

```
File1.FileType := File1.FileType + [fsDirectory] - [fsHidden];
File1.FileType := File1.FileType + [fsNormal] + [fsSystem];
```

would be valid operations with the *FileType* property. If you want to check membership in a set, you simply use the **in** operator. For example,

```
if fsHidden in File1.FileType then ...
```

is the code you would use to test whether the *TFileListBox* was displaying hidden files.
The operation of the two file lists is similar so that the following code from PICVIEW.MAK

```
Sub File1_DblClick ()
' When at the root level (for example, C:\) the Path property
' has a backslash (\) at the end.  When at any other level,
' there is no final \.  This code handles either case to build
' the complete path and filename of the selected file.
     If Right(file1.Path, 1) <> "\" Then
       label1.Caption = file1.Path & "\" & file1.FileName
     Else
       label1.Caption = file1.Path & file1.FileName
     End If
' Load the selected picture file.
     Form1.open.Picture = LoadPicture(label1.Caption)
End Sub
```

would look like this in Delphi

```
procedure TForm1.File1DblClick(Sender: TObject);
begin
    if Length(File1.Directory) = 3 then
        Form1.Caption := File1.Directory + File1.FileName
    else
        Form1.Caption := File1.Directory + '\' + File1.FileName;
    Image1.Picture.LoadFromFile (Form1.Caption);
end;
```

The only difference here is the result of no *Right* function in Delphi. You could code it with *Copy* command in Delphi (which is like MID, see string handling below) but it was just as simple to check the length instead.

Menus

Menus are handled a little differently in Delphi than in VB and allow for much greater flexibility. However, the common functionality between them is fairly straightforward.
To create a menu for a form in Delphi, place a *TMainMenu* component on the form. By default, this is the first control in the Standard page of the Component palette. This component encapsulates all the functionality of the menu. To get to the Menu Designer, simply double-click the main menu component.
The Delphi Menu Designer looks very much like a menu. When it first appears, there is one blank menu item. Simply type in a caption. You will notice that the Object Inspector will record what you type in the *Caption* property and, when you press *Enter*, will use the menu caption to generate a default name for this menu item.
Once you press *Enter*, you can start typing the caption of the first item in this menu and so on. When a particular menu item is selected, it appears in the Object Inspector. Menu items in Delphi have similar properties to menu items in Visual Basic.

Visual Basic	**Delphi**
Caption	*Caption*
Checked	*Checked*
Enabled	*Enabled*
ShortCut	*ShortCut*
Tag	*Tag*

Menu items are used the same way for the most part. An “&” in the caption creates an accelerator. Using a hyphen "-" as the caption creates a separator bar in the menu.
Unlike in the menu designer in VB, you are directly manipulating menu items. Therefore, to move a menu item, you can simply drag it from one place in the menu to another, including to another menu. For example, you can drag an item from the File menu to the Edit menu. To create a sub-menu, right-click a

menu item and choose Create Sub-menu or use *Ctrl-Right Arrow* to create it directly.
To associate code with a menu item, simply double-click the item in the menu designer to get to the *OnClick* event of the menu item. Bear in mind that the only way to see these menu items is to open the Menu Designer by double-clicking on the *TMainMenu* component.
This section doesn't really do justice to the menu functionality of Delphi. Be sure to read the documentation on merging menus and on creating them dynamically. While there are no control arrays as such in Delphi, the ability to dynamically create objects provides even more flexibility and power. A method of emulating control arrays is discussed below.

VBX Support

Delphi provides robust support for Visual Basic version 1 custom controls (VBXs). The great majority of commercial VBXs have been written to detect the version of VB and respond accordingly. The new aspects of VBXs in version 2 and 3 dealt with support for graphical controls and data binding; not many of the former appeared, and the latter are unnecessary, given the database functionality of Delphi.

To add a VBX control to the component library,

1 Open the Install Components dialog box.
2 Choose VBX to open the Install VBX File dialog box.
3 Navigate until you locate the .VBX file you want to add, then choose OK.
4 Once you have added all the VBX controls and other modules you want, choose OK to close the Install Components dialog box and rebuild the component library.

Delphi will then build a wrapper VCL component around the VBX control, thereby integrating it into the Delphi environment. The VBX is *not* directly linked into your application in any way. Instead it remains as a separate and shareable resource. As such it is important to remember that whatever license file was required to use it in Visual Basic will continue to be required to use it in the Delphi development environment. Note that some VBX controls will only operate properly if they are in your path. If you're unsure, you may wish to place the VBX in the \Windows\system subdirectory.

The Component Palette

Now is a good time to mention the differences in the component palette and the toolbox in Visual Basic. In Visual Basic, you define a set of custom controls to be associated with a particular project. By contrast in Delphi, custom components become part of the development environment itself when installed. Of course, in the case of VBXs, this is only a wrapper but the VBX remains "installed", if not loaded into memory beyond the current project.
This is a sensible and beneficial approach, because the overhead associated with Delphi components is *much* less than that of VBXs in VB. In Visual Basic, each VBX is a separate dynamic-link library (though it may contain multiple controls) and as such consumes memory and resources. Therefore, in VB it is very important to limit the number of VBXs you keep loaded during development, because you will quite literally run out of resources.
By contrast, there are three reasons this situation is far better and more efficient in Delphi. First, components are typically smaller because they are all inherited from other components you already have installed. For example, the *TFileList* component is based on the *TListbox* component which is already loaded. Second, the code behind each component is compiled down to native code when installed in the IDE. For example, the DLL containing all of the standard controls is just over 1 Mb in size. Third, all of the installed components are added to a single dynamic-link library for greater memory and resource efficiency.
That said, it *is* possible to control which custom components are loaded at a particular point by changing the dynamic-link library currently installed in the IDE. At any point, you can make a copy of the current COMPLIB.DCL under a new name. You can then load that library at some future point. Bear in mind that,

like in Visual Basic, it is essential you have all of the required controls installed into the IDE when using a project that requires them. This is not the case with distribution of your completed applications, as Delphi Visual Component Library (VCL) components are linked directly into your EXE file.

Advanced Code

The following sections describe some of the more advanced programming concepts of both Visual Basic and Delphi and how they compare.

Units

While there are many differences, the Unit in Delphi is the functional equivalent of the module in Visual Basic. It represents the fundamental unit of code. In VB, there is an implied module associated with each form and the ability to "Add" further modules to your project containing, procedures, DLL declarations as well as global constants and variables.

In Delphi there is an explicit unit associated with each form that contains all of the code associated with a form, including its class definition. That is why when you save a project in Delphi, you are prompted for a unit name (*.PAS). The form is simply saved with the same name and different extension (DFM). In addition, it is possible to make use of any addition units you desire by way of the Uses clause. Unlike Visual Basic, where modules must be loaded into a project and then are available throughout the application, the actual availability of unit resources in Delphi has nothing to do with project membership—a very flexible arrangement!

You gain access to one unit from another by listing that unit in the **uses** clause of your current unit. As you look at the form in a new project, you can see a tab beneath it labeled "Unit1" as this is the default name for the unit. If you click that tab, you will see the code which already exists in the unit. On the fifth line there is a **uses** clause which lists a series of units separated by commas:

```
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;
```

These are the default units which are available to all new forms. You don't need to add any of them to your project; Delphi does this for you. Their presence in the **uses** clause makes their functionality available from within the form. Some of these are significant. For example, `WinProcs` and `WinTypes` add all of the functions of the core Windows API. Some of these such as `Classes, Graphics, Controls` and `Forms` are required by Delphi to make forms work within your application. You should remove units from this default list with care. In most cases, you won't touch this list because if no functionality from a particular unit is exploited, the code for that unit is not linked in. If you would like to exploit the functionality of *Unit1* inside of *Unit2*, you need to include *Unit1* in a **uses** clause within *Unit2*.

Like modules, units may contain procedures, DLL declarations, user-defined types and global variables and constants. There are two sections to a unit, the **interface** and the **implementation**. The interface is the "public" area of the unit and functionality defined in that area is available anywhere in the application where that unit is listed in a **uses** clause. The **implementation** portion of a unit defines the "private" area of that unit so variables, constants and functions defined *only* there are available only within that unit. This is discussed further in the discussion of variable scope and procedures below.

So, remember that adding a unit to a project is just a convenience for editing and compiling. If you want to access the functionality of that unit, it must find its way into a **uses** clause in your code.

Variable Scope

Delphi supports nearly all of the levels of scope found in Visual Basic and more. The following table represents the different levels of scope in Visual Basic and their corresponding scopes in Delphi.

Visual Basic	**Delphi**
Local	*Local*
Static	*typed constant*
Module Level	*Unit Level*
Global	*Global*
n/a	*Object Level*

In Visual Basic, if you use the Dim keyword inside a procedure definition, you are defining a variable which is local to that procedure. Likewise if you add a variable to the **var** section of a procedure in Delphi, you are defining a local variable. There is no equivalent to a static variable in Delphi, but a static variable is really just a module level variable which is used in only one procedure. It is a naming convenience more than anything else.
In Visual Basic, if you use the Dim keyword in the declarations section of a module or form, you are creating a module level variable. Likewise, if you place a variable in the **Var** clause of the Implementation section of a unit, you are defining a unit level variable (i.e. one which can only be seen from within that unit).
In Visual Basic, if you use the Global keyword in the declarations section of a module, you are defining a global variable. In Delphi, you simply put a variable in the **var** clause of the **interface** section of a unit and that variable is available to any other unit which includes the variable's unit in its **uses** clause. The following code segment helps to clarify:

```
unit MyUnit;

interface

uses
  WinProcs, WinTypes;  {these are the units used by this unit}

var
  globalInt:integer;
  globalStr:string;

implementation

var
  unitInt:Integer;
  unitStr:String;

procedure unitProc;
const
  staticInt:Integer = 0;
  staticStr:String = '';
var
  localInt:Integer;
  localStr:String;
begin
  {code goes here}
end;

end.
```

The preceding was a complete unit with an **interface** and an **implementation** section. The variables defined in the **interface** section are global. The variables defined in **var** clause of the unit are unit level and finally those defined in the **var** section of the procedure are local.
Be aware that unit-level variables created inside a form are available across multiple instances of that form. In other words, unlike in Visual Basic, those variables exist only once per application so even though you might have multiple instances of a form, unit level variables are shared.
The way around this is to define a set of variables as part of the new form class you are creating. A typical form definition might look something like this:

```
type
 TForm1 = class(TForm)
   Button1: TButton;
   Button2: TButton;
 private
   { Private declarations }
 public
   { Public declarations }
 end;
```

You can create the equivalent of VB's "form level" variables by adding them to the **private** section of the form definition. So the change might look something like this:

```
type
 TForm1 = class(TForm)
   Button1: TButton;
   Button2: TButton;
 private
   { Private declarations }
    formInt:Integer;
    formStr:String;
 public
   { Public declarations }
 end;
```

If you don't create multiple instances of your forms in your applications, you'll never know the difference between unit level and form level variables. However, a great many MDI applications begin as single form applications so it is best to use the most restrictive level of scoping you can for flexibility in the future.

Conditional Execution

There are two types of conditional execution in Delphi as in Visual Basic, **if..then** and **case**. These constructs are similar in both environments.

If...Then...Else

Nearly every language has some form of the **if** statement. In its simplest form in Visual Basic, it takes the form:

```
If <condition> Then <action> Else <action>
```

which is exactly like the simple form in Delphi. Therefore, the statement,

```
If chkShow.Value = True Then Text1.Visible =True Else Text1.Visible = False
```

in Visual Basic, looks like the following in Delphi:

```
If chkShow.State = cbChecked Then Edit1.Visible := True Else Edit1.Visible :=
False;
```

The difference appears when you need to execute multiple lines of code in a condition. As stated above, Object Pascal requires that blocks of code be enclosed in a **begin..end** pair. So the following would be the Delphi code to set the *Visible* property of multiple controls to *True* given a particular condition:

```
if chkShow.State = cbChecked then
                    begin
                       Edit1.Visible := True;
                       Edit2.Visible := True;
                    end;
```

Now take a look at an example involving **else**. You will notice a small anomaly in syntax in this instance:

```
if chkShow.State = cbChecked then
                    begin
                       Edit1.Visible := True;
                       Edit2.Visible := True;
                    end
else
                    begin
                       Edit1.Visible := False;
```

```
                    Edit2.Visible := False;
                end;
```

The one exception to the rule about ending each line with a semicolon is with lines immediately preceding an **else**. Accordingly, you will notice above that the first **end** statement doesn't have a semicolon because it is immediately followed by **else**. These **begin..end** pairs should not be confused with VB's requirement to complete a multi-line **if** block with an **EndIf**. There is no EndIf equivalent in Delphi. You can, of course, nest **if** statements as long as each block of code is wrapped in **begin..end**.

Case

The Select Case statement in Visual Basic is designed to provide a more elegant structure than a series of multiple **if** clauses. The **case** statement in Delphi provides similar functionality. The following code in Visual Basic:

```
Select Case MyVar
  Case 0
                    J = "Hello"
  Case MY_CONST
                    j = "Goodbye"
                    Flag = False
  Case 2, 3, 5 To 10
                    j = "How are you?"
End Select
```

would translate into the following in Delphi's Object Pascal:

```
case MyVar of
  0:
       J := 'Hello';
  MY_CONST:
                    begin
                       j := 'Goodbye';
                       Flag := False;
                    end;
          2, 3, 5..10:
                    j := 'How are you?';
```

Two things should be clear. First, the code block after MY_CONST required a **begin..end** pair and second, there is nothing like End Select. The **case** construct simply ends with the last line of code—which is the case with most constructs in Object Pascal, including loops, discussed in the next section.

As in Visual Basic, you can combine multiple cases by separating them with a comma. A range is denoted by `<LowerVal..HigherVal>` rather than the **to** keyword. There is no equivalent for "Is < 500" Instead, you need to use a range.

When using the **case** statement in Delphi, remember that you cannot use strings, only numbers for comparison. If you need to compare a series of strings, you will need to create a set of **if..then** statements. There is, however, a data type in Delphi called **char** which is a single letter. Because this can be represented as a number, it is possible to use the **case** statement to compare **char** values. In other words, the following example would be valid:

```
var
  C:Char;
begin
  Case C of
                    'A':
                        DoSomething;
                    'B', 'D'..'G':
                        DoSomethingElse;
end;
```

If you need to compare a series of strings, you will need to do that with a set of **if..then** statements rather than with **case**.

Loops

As in any language, there are two different kinds of loops, determinate and indeterminate. The difference is simply whether you know the number of times you want the loops to be executed before code execution proceeds. The determinate loop in Visual Basic is the For..Next loop and there is a similar construct in Delphi called the **for** loop. For indeterminate loops in VB you have the While..Wend and Do..Loop loops. In Delphi, you have the **while** and **repeat..until** looping constructs.

The For Loop

As in Visual Basic, the **for** loop in Delphi allows you to execute a block of code a predetermined number of times. The syntax is very similar. In VB, you might have something that looks like this:

```
For X = 1 To 10
                    A  = A + X
Next
```

whereas in Delphi the same thing would be represented like this:

```
for X := 1 To 10 do
                    A := A + X;
```

The two differences are the **do** keyword at the end of the **for** statement and the missing Next which is implied by the end of the code block. In other words if you had more than one operation to perform it might look like this in Delphi:

```
for X := 1 To 10 do
                    begin
                       A := A + X;
                       Caption := IntToStr(X);
                    end;
```

where the **begin..end** block defined the lines of the loop.
In Visual Basic, you can use the Step keyword to specify the increment by which the variable (i.e. X) should be changed. This is used to count by more than one (i.e. Step 5) or to create a decrementing counter (i.e. Step -1). While there is no way to create an increment larger than one in Delphi, it is possible to create a decrementing counter with the **downto** keyword. It is used in the following manner:

```
for X := 10 downto 1 do
                    begin
                       A := A + X;
                       Caption := IntToStr(X);
                    end;
```

where X begins at 10 and is decremented to 1. As in Visual Basic, loops in Delphi can be nested.

The Do Loop

The functionality of Visual Basic's Do..Loop construct is provided by Delphi's **repeat..until** style loop. Therefore, the following loop in Visual Basic

```
Do
 K = I Mod J
 I = J
 J = K
Loop Until J = 0
```

would look like the following in Delphi

```
repeat
 K := I Mod J;
 I := J;
 J := K;
until J = 0;
```

In this example, it is important to remember that the comparison operator in Delphi is a simple = sign, not the := used for assignment. The other thing you will notice for the first time is the conspicuous absence of **begin..end**. This is because the **repeat..until** loop takes the place of the **begin** and **end** statements. It is the only loop that is contained in this way. All of the other loops are defined only by their first line and therefore require the **begin..end** pair for multiple lines of code.

The While Loop

The **repeat** loop, like the Do loop, does its testing at the bottom so the code is executed at least once. Sometimes you want to do your test at the top because there are certain instances in which the code block shouldn't be executed at all. In Delphi, as in Visual Basic, this is accomplished with the **while** loop. The following example of Visual Basic code:

```
While CanDraw = True
  A = A + 1
Wend
```

would look like this in Delphi

```
while CanDraw = True do
  A := A + 1;
```

where, as with the **for** loop, there is no terminator necessary. Instead, if you want to execute a block of code, you use the **begin..end** construct. In other words, this Visual Basic code:

```
While Not Eof(1)
  Line Input #1, Text
  Process Text
Wend
```

might look something like this in Delphi:

```
while not Eof(InFile) do
 begin
  ReadLn (InFile, Text);
  Process (Text);
 end;
```

As you can see, there are structures common to both languages making it easy to make the transition between them.

String Handling

While Delphi has a string variable type, it does carry some restrictions of which you should be aware. First, string length is limited to 255 characters and is not used directly when calling API functions requiring an LPSTR.

Other than these limitations, a Delphi string can be used just like a string in Visual Basic in terms of assignment (literals enclosed in single quotes, however), concatenation and comparison. There are a number of string manipulation routines in Delphi just as there are in VB:

Visual Basic	**Delphi**
Str	*IntToStr, Str*
Val	*StrToInt, Val*
Len	*Length*
Instr	*Pos*
Mid	*Copy*
Ucase	*UpperCase*
Lcase	*LowerCase*
Format	*Format*

In addition, a Delphi string can be treated as an array of characters making certain search and replace operations easier. In Delphi use the syntax:

```
OpFlag := TButton(Sender).Caption[1];
```

to assign the first character of a string to a variable of type **char**. Remember from above that you cannot use the **case** statement on a string but you can on a single character. This statement allows you to retrieve the first character of a string as its ASCII value for use in a **case** statement.

Likewise, if you need to create an LPSTR to pass to an API function, you can construct one in a string and pass its address like this:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  S:String;
begin
  S := 'Hello World'#0;
  MessageBox (0, @S[1], 'Test', 0);
end;
```

The actual Delphi data type that corresponds to the LPSTR in C is the *PChar* which is a pointer to a character array. There are a series of functions designed to manipulate *PChar*-style strings for such things as assignment, concatenation and comparison. Search for "String-handling routines (null-terminated)" in online Help for a list of the relevant functions. The above code implemented using *PChars* would look something like the following:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  P:PChar;
begin
  P := StrAlloc (256);
  StrPCopy (P, 'Hello World!');
  MessageBox (0, P, 'Test', 0);
  StrDispose (P);
end;
```

A few important things to notice are that you don't have to explicitly include the null character in the assignment, you can use *StrAlloc* to create a string much bigger than 255 characters and finally you don't have the use the @ symbol when passing a *PChar* to an API function because the *PChar* is already defined as a pointer.

Arrays

Arrays are used in Delphi much the same way they are in Visual Basic. The VB code to create an array looks like:

```
Dim MyArr (10, 1 To 5) As Integer
```

and the same definition in Delphi would look something like:

```
MyArr: array [0..10, 1..5] of Integer;
```

In Delphi you must pre-define the bounds of the array.

It is, however, possible to pass an array to a procedure that doesn't know how many elements are in the array much like the open parentheses in Visual Basic. So the following code in Visual Basic:

```
Dim MyArr (1 to 10) As Integer

Sub Set2Zero (A() As Integer)
  Dim i As Integer
  For i = LBound (A) to UBound (A)
                        A(i) = 0
  Next
End Sub

Sub Command1_Click ()
  Set2Zero MyArr()
End Sub
```

might look something like this in Delphi:

```
var
  MyArr: array [1..10] of Integer;

procedure Set2Zero (A:array of Integer);
var
  i:Integer;
begin
  for i := Low(A) to Hight(A) do
```

```
                    A[i] := 0;
end;

procedure TForm1.Command1Click (Sender: TObject);
begin
  Set2Zero (MyArr);
end;
```

This syntaxl facilitates greater flexibility in the construction and calling of generic procedures.

Procedures and functions

In Delphi, as in Visual Basic, you are creating procedures before you realize it in the form of event handlers. You have been using subroutines (known as *procedures* in Delphi) every time you double-click on an object to create an event handler. The general syntax for a procedure in Delphi is:

```
procedure MyProc (P1:Type; P2:Type);
var {optional}

begin
  {code}
end;
```

which looks fairly similar to:

```
Sub MyProc (P1 As Type, P2 As Type)
  Dim...    'optional
  {code}
End Sub
```

The key difference is that parameters in a procedure definition are separated by semicolons rather than commas in Delphi. In addition, keep in mind that to define local variables requires the creation of a Var clause within the procedure.

Delphi functions are also similar to their VB counterparts. The following declaration in VB:

```
Function MyFunc (P1 As Integer, P2 As Long) As Integer
  Dim...
  'Code
  MyFunc = ...
End Function
```

is nearly identical to the Delphi equivalent:

```
function MyFunc (P1:Integer; P2:LongInt):Integer;
var

begin
  {Code}
  MyFunc := ...
end;
```

One fundamental difference is that Delphi requires the declaration of a procedure before its use, unlike Visual Basic. You can either implement the procedure before you use it or declare it with a forward reference like so:

```
procedure OtherProc (P1:Integer);forward;

procedure MyProc;
begin
  OtherProc (37);
end;

procedure OtherProc (P1:Integer);
begin
  DoSomethingTo (P1);
end;
```

Another important difference is that parameters are, by default, passed by value to Delphi procedures and functions whereas they are passed by reference in Visual Basic. Delphi gives you the flexibility to choose

whichever approach best suits your needs. To define a parameter as being a reference, use the var keyword when defining the procedure, as in the following example:

```
procedure MyProc(var P1 as Integer);
begin
end;
```

Scope of procedures and functions is similar to scope of variables. Those procedures defined in the **implementation** section of a unit are local to that unit. Those which are declared in the **interface** section (they are all *implemented* in the **implementation** section) are available to others which include this unit in their **uses** clause.

The following is the complete source for a unit called VBFUNC which contains a QBColor function designed to mimic the functionality of the like-named VB function.

```
unit VBFUNC;

interface

uses Graphics; {location of TColor def}

function QBColor (n:Integer):TColor; {mentioning here makes it public}

implementation

function QBColor (n:Integer):TColor;
var
  C:TColor;
begin
    case n of
         0: C := 0;
         1: C := 8388608;
         2: C := 32768;
         3: C := 8421376;
         4: C := 128;
         5: C := 8388736;
         6: C := 32896;
         7: C := 12632256;
         8: C := 8421504;
         9: C := 16711680;
         10: C := 65280;
         11: C := 16776960;
         12: C := 255;
         13: C := 16711935;
         14: C := 65535;
         15: C := 16777215;
    end;
    QBColor := C;
end;

end.
```

As you can see, there's no “witchcraft” here. They key element is if you want this to be a public function, you need to declare the function in the interface section of the unit. You then need to include VBFUNC in the uses clause of any unit in which you would like access to this function, as was done in the implementation section of the FARRAY.PAS form unit.

Control Arrays

While the concept of a control array doesn't exist as such in Delphi, the functionality (and much more) is certainly available to you. There are two reasons to create a control array in Visual Basic. The first is so that multiple controls might share the same code. This can be accomplished by simply assigning the same event

handler to say the OnClick event handler of multiple components on a Delphi form. The second is that you want to create controls on the fly. This too is more than possible in Delphi.
You might, for example, create a form with three buttons. In Visual Basic, they would all have the name B and different indexes. You could then write code like this:

```
Sub B_Click (Index As Integer)
  Caption = B(i).Caption
End Sub
```

If you want to accomplish the same thing in Delphi, you would create three buttons and give them descriptive names such as B_1, B_2 and B_3. You would then create an event handler for one of them called B and then assign that event handler to the other two. That event handler would look something like this:

```
procedure TForm1.BClick(Sender:TObject);
begin
  Caption := TButton(Sender).Caption;
end;
```

It should now be clear the purpose of that "Sender" parameter that's been ignored up until now. It refers to the component which activated the event handler. Just as you use the Index parameter in Visual Basic to differentiate between the controls which might call the event handler, you use *Sender* to determine the source of the event in Delphi.
The second thing of note is that you must "cast" the *Sender* parameter to a particular object type. The data type of *Sender* is simply *TObject* which could be just about anything in Delphi. Therefore, a button, a text box and a check box could all share the same event handler in Delphi, something which would not be possible in Visual Basic. The only price for this flexibility is that you must tell the compiler how to treat this *TObject* by casting it to an object type a little lower down in the hierarchy. The syntax for casting is much like that of a function where you simply use the class name as the function and the *TObject* as the single parameter. This tells Delphi to temporarily treat this *TObject* as a *TButton*. You would need to continue to cast this variable as you needed it. If you wanted to make your code a little more readable, you could make an assignment to a local variable of type *TButton* like this:

```
procedure TForm1.BClick (Sender:TObject);
var
  Source:TButton;
begin
  Source := TButton(Sender)
  Caption := Source.Caption;
end;
```

Dynamic Control Arrays

The other reason to use a control array in Visual Basic is that you want the ability to create controls "on the fly" based on run-time information. The mechanism in VB is that you must create a template and then build copies of that template. You have no such limitations in Delphi. You can create any object on the fly and dynamically assign event handlers to it.
In the VB sample, ARRAY.FRM form in the \vb\samples\controls\controls.mak, there is an option button on the form called optButton with an index of 0 which gets used as a template in the cmdAdd_Click procedure:

```
Sub cmdAdd_Click ()
   If MaxId = 0 Then MaxId = 1
   If MaxId > 8 Then Exit Sub
   MaxId = MaxId + 1
   Load OptButton(MaxId)
   OptButton(MaxId).Top = OptButton(MaxId - 1).Top + 400
   OptButton(MaxId).Visible = True    ' Display new button.
   OptButton(MaxId).Caption = "Option" & MaxId + 1
End Sub
```

A comparable routine in Delphi might look something like this:

```
procedure TForm1.cmdAddClick(Sender: TObject);
var
```

```
  rbOld, rbNew: TRadioButton;
begin
  if nCtl = 0 then
     nCtl := 1
  else
     Inc(nCtl);
  if nCtl < 16 then begin
     rbOld := TRadioButton(FindComponent ('optButton_' + IntToStr (nCtl
-1)));
     rbNew := TRadioButton.Create(Self);
     rbNew.Parent := Self;
     rbNew.SetBounds (rbOld.Left, rbOld.Top + rbOld.Height * 2,
                  rbOld.Width, rbOld.Height);
     rbNew.Caption := 'Option' + IntToStr(nCtl);
     rbNew.Name := 'optButton_' + IntToStr(nCtl);
     rbNew.OnClick := optButton_0Click;
  end;
end;
```

In this code the beginning looks just like the Visual Basic example, checking for valid data. Then the following occurs:

1 Set *rbOld* to last component created.
This is accomplished with a very useful function called FindComponent which takes a string value. Try *that* in VB!

1 *rbNew* is created
2 *rbNew* is placed on the form
3 *rbNew* is moved to a new location
4 *rbNew*'s caption is set
5 *rbNew*'s name is set!
6 *rbNew*'s *OnClick* handler is dynamically assigned to the *optButton_0Click* handler!

The net result is exactly the same as with the Visual Basic code above. The *optButton_Click* handler in VB looks like this:

```
Sub optButton_Click (Index As Integer)
   picDisplay.BackColor = QBColor(Index + 1)
End Sub
```

and looks very similar in Delphi:

```
procedure TForm1.optButton_0Click(Sender: TObject);
var
  i:Integer;
  rb:TRadioButton;
begin
  rb := TRadioButton(Sender);
  Caption := rb.Name;
  i := StrToInt (Copy (rb.Name, 11, 2));
  shpDisplay.Brush.Color := QBColor (i);
end;
```

The big difference is that in the absence of an *Index* parameter, you're extracting the last character of the caption to get the index value. This is just one technique to handle control arrays. For example, the *Tag* property in Delphi is a long integer rather than a string. So the two routines could look like this:

```
procedure TForm1.cmdAddClick(Sender: TObject);
var
  rbOld, rbNew: TRadioButton;
begin
  if nCtl = 0 then
     nCtl := 1
  else
     Inc(nCtl);
  if nCtl < 16 then begin
```

```
      rbOld := TRadioButton(FindComponent ('optButton_' + IntToStr (nCtl
-1)));
      rbNew := TRadioButton.Create(Self);
      rbNew.Parent := Self;
      rbNew.SetBounds (rbOld.Left, rbOld.Top + rbOld.Height * 2,
                       rbOld.Width, rbOld.Height);
      rbNew.Caption := 'Option' + IntToStr(nCtl);
      rbNew.Name := 'optButton_' + IntToStr(nCtl);
                       rbNew.Tag := QBColor (nCtl);            {this is new}
      rbNew.OnClick := optButton_0Click;
    end;
  end;
  procedure TForm1.optButton_0Click(Sender: TObject);
  begin
    shpDisplay.Brush.Color := TRadionButton(Sender).Tag;
  end;
```

In the example above, the actual color value is stored in the Tag property so that all the optButton_0Click routine needs to do is set the color equal to the Tag of the Sender.

Finally, the elements of the control array are removed in a manner similar to that in Visual Basic:

```
Sub cmdDelete_Click ()
   If MaxId = 1 Then Exit Sub      ' Keep first two buttons.
   Unload OptButton(MaxId)         ' Delete last button.
   MaxId = MaxId - 1               ' Decrement button count.
   OptButton(0).SetFocus           ' Reset button selection.
End Sub
```

becomes this in Delphi:

```
  procedure TForm1.cmdDeleteClick(Sender: TObject);
  var
    rb:TRadioButton;
  begin
      if nCtl > 0 then
       begin
         rb := TRadioButton (FindComponent ('optButton_' + IntToStr(nCtl)));
         rb.Destroy;
         Dec(nCtl);
       end;
  end;
```

Another example of control arrays can be found in the CALC.DPR sample in the [VBSAMPL] directory.

Object Variables

Object variables are a rather new concept to Visual Basic although you have seen them as parameters to procedures in both VB and Delphi. Another way to manipulate a set of controls as an array in Visual Basic is to create an array of type control and populate it at Form_Load like so:

```
Dim T (1 To 10) As TextBox

Sub Form_Load ()
   Dim c As Integer
   Dim i As Integer
   i = 1
   For c = 0 to Form1.Controls.Count -1
      If TypeOf Form1.Controls (c) is TextBox then
        Set T(i) = Form1.Controls(c)
        i = i + 1
        end if
   Next
End Sub
```

You could then walk through this array and set a property of all of the member controls to a certain value,

just as with a control array. The same example in Delphi might look something like this:

```
var
  i, c:Integer;
begin
  i := 1;
  for c := 0 to form1.componentcount -1 do
      if Form1.Components[c] Is TEdit then
         begin
         T[i] := TEdit(Form1.Components[c]);
         Inc (i);
         end;

end;
```

Object variables are also used in Visual Basic to create multiple instances of forms using the New keyword. The \VB\SAMPLES\OBJECTS\OBJECTS.MAK project in VB demonstrates this functionality. The Visual Basic sample has code that looks like the following:

```
Sub cmdNextInstance_Click ()
   Dim NextInstance As New frmMain
   NextFormNum = NextFormNum + 1
   NextInstance.Caption = "Instance # " & NextFormNum
   NextInstance.Left = Left + (Width \ 10)
   NextInstance.Top = Top + (Height \ 10)
   NextInstance.Show
End Sub
```

Similar functionality can be achieved in Delphi with a procedure that looks like this:

```
procedure TfrmMain.cmdNewInstanceClick(Sender: TObject);
var
  F:TfrmMain;
begin
  F := TfrmMain.Create(Application);
  Inc(NextFormNum);
  F.Caption := 'Instance #' + IntToStr(NextFormNum);
  F.Left := Left + (Width div 10);
  F.Top := Top + (Height div 10);
  F.Visible := True;
end;
```

With the exception of the .Create method, this code is practically identical to the code you are used to in Visual Basic.

Graphics

Graphics in Delphi is a larger topic than this chapter allows but it is important to touch on a few of the fundamental graphics concepts in Delphi and how they compare to Visual Basic.

Canvases

The *TCanvas* object is an abstraction that Delphi provides as a drawing surface in a number of components including the form, image, printer and listbox! Essentially a wrapper around a device context, this object allows you to write routines which operate on any of a number of different component types rather than just one as in VB.

All of the graphic methods you have for a Picturebox or Form in Visual Basic have equivalent methods in a Delphi *TCanvas*:

Visual Basic	**Delphi**
Circle	*Elipse, Arc, Pie*
Line	*LineTo, Rectangle*

In addition, the *TCanvas* represents a fairly complete wrapper around the graphics API of Windows so there are methods to support functionality that would otherwise require the use of the Windows API in VB. Examples are: CopyRect, FloodFill, and Polygon. The key portion of VB\SAMPLES\CALLDLLS\CALLDLLS.MAK code in VB looks like this:

```
temp = BitBlt(hDC, X, Y, PicWidth, PicHeight, picCopy.hDC, 0, 0, SRCCOPY)
temp = BitBlt(picCopy.hDC, 0, 0, PicWidth, PicHeight, hDC, NewX, NewY,
SRCCOPY)
temp = BitBlt(hDC, NewX, NewY, PicWidth, PicHeight, picMask.hDC, 0, 0,
SRCAND)
temp = BitBlt(hDC, NewX, NewY, PicWidth, PicHeight, picSprite.hDC, 0, 0,
SRCINVERT)
```

the same functionality in the Delphi example looks like this:

```
C.CopyMode := cmSrcAnd;
  C.CopyRect (Dest, imgMask.Canvas, Src);
  C.CopyMode := cmSrcInvert;
  C.CopyRect (Dest, imgSprite.Canvas, Src);
  C.CopyMode := cmSrcCopy;
```

where C refers to the canvas property of an image control. Furthermore, it is possible to create a canvas and assign an hDC into that canvas, thereby minimizing the amount of Windows API functionality required. In the Sprite sample, the code checks whether to act on the form or the desktop and assigns a different value to the local canvas variable:

```
if chkDesk.State = cbChecked then begin
     C := TCanvas.Create;
     C.Handle := GetDC(GetDesktopWindow);
     RightEdge := GetSystemMetrics(SM_CXSCREEN);
     BottomEdge := GetSystemMetrics(SM_CYSCREEN);
     end
  else begin
     C := Canvas; {the Canvas property of the form}
     RightEdge := ClientWidth;
     BottomEdge := ClientHeight;
  end;
```

Therefore, the canvas is an extremely important aspect of graphics programming in Delphi, especially from the perspective of hiding the complexity of manipulating the drawing surface of different objects.

Image Manipulation

Similarly, image manipulation in Delphi is designed around the concept of a picture object such as *TBitmap*. This object too has a canvas which can be manipulated the same way that of a picture box or form can be.

Suppose you wanted to mimic the functionality of the PICCLIP.VBX control which ships with Visual Basic Professional. This control allows you to extract portions of a matrix of pictures and assign the "cell picture" to another picture control.

To accomplish this, you might create a subroutine which extracted a portion of a picture like this and assigned it to another picture property:

```
procedure GetCell (Index:Integer; Dest, Source:TPicture);
var
  BWidth, BHeight:Integer;
  SrcR, DestR:TRect;
begin
  BWidth := Source.Picture.Width div FCols;
  BHeight := Source.Picture.Height div FRows;

  DestR.Left := 0;
  DestR.Top := 0;
```

```
    DestR.Right := BWidth;
    DestR.Bottom := BHeight;

    SrcR.Left := (Index mod Cols) * BWidth;
    SrcR.Top := (Index div Cols) * BHeight;
    SrcR.Right := SrcR.Left + BWidth;
    SrcR.Bottom := SrcR.Top + BHeight;

    Dest.Bitmap.Width := BWidth;
    Dest.Bitmap.Height := BHeight;
    Dest.Bitmap.Canvas.CopyRect (DestR, Source.Canvas, SrcR);
  end;
```

This is a pretty clean routine which doesn't use any API calls. Although it might be “showing off,” you can take this a step further. What if you actually wanted to create a PICCLIP component for Delphi? Where would you begin? You want a control that is prepared to take a picture property, so why not begin with a *TImage* component and work from there. That's exactly what was done in the following unit, which represents an implementation of PicClip for Delphi. You need only install PRCCLIP.PAS into the IDE and you'll have a PicClip component! Here is the complete source code for the PicClip component for Delphi:

```
unit PicClip;

interface

uses Classes, Controls, WinTypes, Graphics, StdCtrls;

type
TPicClip = class(TImage)
private
  FRows:Integer;
  FCols:Integer;
  FPicture:TPicture;
  function GetCell (Index:Integer):TPicture;
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy;override;
  property GraphicCell[Index:Integer]:TPicture read GetCell;
published
  property Rows:Integer read FRows write FRows;
  property Cols:Integer read FCols write FCols;
end;

procedure Register;

implementation

constructor TPicClip.Create(AOwner: TComponent);
begin
 inherited Create(AOwner);
 FPicture := TPicture.Create;
 Visible := False;
end;

destructor TPicClip.Destroy;
begin
  FPicture.Destroy;
  inherited Destroy;
end;

function TPicClip.GetCell (Index:Integer):TPicture;
var
  BWidth, BHeight:Integer;
```

```
  SrcR, DestR:TRect;
begin
  BWidth := Picture.Width div FCols;
  BHeight := Picture.Height div FRows;

  DestR.Left := 0;
  DestR.Top := 0;
  DestR.Right := BWidth;
  DestR.Bottom := BHeight;

  SrcR.Left := (Index mod Cols) * BWidth;
  SrcR.Top := (Index div Cols) * BHeight;
  SrcR.Right := SrcR.Left + BWidth;
  SrcR.Bottom := SrcR.Top + BHeight;

  FPicture.Bitmap.Width := BWidth;
  FPicture.Bitmap.Height := BHeight;
  FPicture.Bitmap.Canvas.CopyRect (DestR, Canvas, SrcR);

  GetCell := FPicture;
end;

procedure Register;
begin
 RegisterComponents('Foxhall', [TPicClip] );
end;

end.
```

It is intended that the examples provided in this document have helped highlight some of the key differences between programming with the interpreter-based Visual Basic Pro and Borland's next-generation Delphi, built around a state of that art optimising native code compiler. Furthermore, the sample code also makes clear the differences between the script-language style of Visual Basic versus the more structured, object-oriented approach made possible by Delphi's Object Pascal language. You should now be able to begin applying your programming skills productively in Delphi. You can also use a variety of third party tools and components to facilitate creation of Delphi or Delphi Client/Server applications including tools that help convert existing code into Delphi code.

As you become familiar with Delphi programming you will find that Delphi and Delphi Client/Server are able to take you further in your application development with greater performance, reuse and scalability than Visual Basic Pro.