Tips & Tricks

The TLabel Enabled Property

In all the control components in the VCL, setting the `Enabled` property will dim the control by setting it to a grey colour. This is a signal to the user that the control cannot be selected. In most cases Windows itself handles the disabling and greying of the control through the `EnableWindow` API.

However, looking at the VCL reveals that `TLabel` doesn't descend from the `TWinControl` class but rather from the `TGraphicControl` class. This means that drawing of the control must be handled by the VCL itself. When the `Enabled` property of a `TLabel` control is set to false, the VCL uses a hard-coded colour of `clGrayText` when drawing the label.

This works fine for a white background, but with default grey forms and panels, the labels simply disappear when `Enabled` is set false. Most of the time, this is not want you want. To hide a label, you set the `Visible` property to false. Setting `Enabled` to false should just dim it, not make it invisible!

The simple solution to this problem is to set the colour to a known grey colour that is slightly different from the background colour, instead of setting the `Enabled` property. For instance:

```
procedure TTestDialog.UpdateDialog(
  Value: boolean);
begin
  NameEdit.Enabled := Value;
  { This is a workaround for the
    Enabled:=false bug for TLabel }
  if Value then
    NameLabel.Font.Color := clBlack
  else
    NameLabel.Font.Color := clGray;
end;
```

When you have a lot of labels this can become quite tiresome. So, I created my own component, called `TFixedLabel`, which solves this problem by re-implementing the `Enabled` property. The code for the `TFixedLabel` component is shown in Listing 1.

The component works by setting the colour of the font to a grey colour that differs from the background colour (the colour of the parent) when it is disabled. The original font colour is saved in a private field. The font colour is restored when the component is enabled again.

First we declare a new `published` property with the same name as the `Enabled` property in the `TLabel` component. This will ensure that only the new property will appear in the Object Inspector.

The original `Enabled` property is always set to true. This ensures that the VCL code will always use the set colour of the font rather than the hardcoded `clGrayText` colour.

With this done, we also need to override the `Font` property to save any new colours assigned to the font. Because the `TLabel.Enabled` property will always be true, we also have to override the handling of the `CM_DialogChar` message and only perform the default behaviour when our own `Enabled` property is true.

Finally, I added the abilty to enable and disable the component associated with the label through the `FocusControl` property. The `FocusEnable` property turns this ability on and off. If `FocusEnable` is true, the associated control will be enabled and disabled in parallel with the label itself.

Now our previous code can be simplified like this:

```
procedure TTestDialog.UpdateDialog(
  Value: boolean);
begin
  NameLabel.Enabled := Value;
end;
```

This assumes that the `FocusEnable` property is true (the default) and the `FocusControl` property is set to `NameEdit`.

Note that overriding properties this way is a static, compile-time override rather than a run-time polymorphic override. This is because property resolution is done at compile-time rather than at run-time.

If the property access methods (`GetEnabled` and `SetEnabled`) had been implemented as non-private virtual methods, we would have been able to override them in a true object oriented sense. Unfortunately, Borland have implemented all property access methods as static and private, making it impossible to override them.

More than just fixing an innocent bug, this code shows how you can override existing properties. Just remember that this method is not polymorphic, that is, existing VCL code will keep accessing the old property, but in all the code you write (and through the Object Inspector) you can access the new overridden property.

Contributed by Hallvard Vassbotn (internet email: hallvard@falcon.no)

Event Chains In Delphi

One of the new language features in Delphi are event handlers. By default, on double clicking an event in the Object Inspector Delphi writes an empty event handler and assigns it to the event. Then it's your turn to fill in the code that should run when that event occurs. In most cases this is all you need, but in some places it would be nice to assign the same event-handler to a number of different components so that they all act in

the same manner, but without losing the individuality of single components.

For example, in a form you have a control which should act in some way if the user presses the F4 key, but you also want the same control to act on an F5 keypress and you'd like to share the code for this with other controls and make it totally independent from the F4 keypress event handler. What do you do? I discovered two different techniques to cope with this situation.

One technique would be to make the event handler virtual and call the previous handler using the `inherited` keyword. This could be problematic because you must define the event handler as virtual when you create it. Also, you must *always* call the inherited handler (which I often forget if I am writing code quickly!). Finally, it is not really independent of the form, so I threw this idea away!

Another method is to save the assigned event handler and re-assign a new one. I call this technique *Event Chaining*. In Turbo Pascal since version 4.0 there was a way to add your own procedure into a program's exit handler using a procedure pointer called `ExitProc` (In Delphi's ObjectPascal the same task is achieved more easily by `AddExitProc`). The technique is:

- Save the last value of `ExitProc` into a variable named `SavExitProc`;
- Re-assign `ExitProc` to your own exit handler;
- If the handler is called restore `ExitProc` to the value of `SavExitProc`, so that after processing your code the system calls the next handler in the chain.

I find this technique very good and after thinking for some minutes I realised I wanted something just like this for event handlers. So far it works very well!

I've included a small example (on the disk with this issue) that demonstrates how to chain two key event handlers (one assigned directly using the Object Inspector, the second defined in code) for one control.

➤ *Listing 1*

```
unit FLabel;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;
type
  TFixedLabel = class(TLabel)
  private
    { Private declarations }
    FEnabled: boolean;
    FFocusEnable: boolean;
    FOrigColor: TColor;
    procedure UpdateFocusedEnabled;
    procedure SetEnabled(Value: boolean);
    function GetFont: TFont;
    procedure SetFont(Value: TFont);
    function IsFontStored: boolean;
    procedure SetFocusEnable(Value: boolean);
    procedure CMDialogChar(var Message: TCMDialogChar);
      message CM_DIALOGCHAR;
  protected
    { Protected declarations }
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
  published
    { Published declarations }
    property Enabled: boolean read FEnabled write
SetEnabled default True;
    property Font: TFont read GetFont
      write SetFont stored IsFontStored;
    property FocusEnable: boolean read FFocusEnable
      write SetFocusEnable default True;
  end;
procedure Register;

implementation
type
  TPublicWinControl = class(TWinControl)
  public
    property Color;
  end;

constructor TFixedLabel.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FEnabled := true;
  FOrigColor := Font.Color;
  FFocusEnable := true;
  { This is the default, just to make sure... }
  inherited Enabled := true;
end;

procedure TFixedLabel.SetEnabled(Value: boolean);
var LabelColor : TColor;
begin
  if Value <> FEnabled then begin
    inherited Enabled := true;
    FEnabled := Value;
    if FEnabled then
      LabelColor := FOrigColor
    else begin
      FOrigColor := inherited Font.Color;
      LabelColor := clBtnFace;
    end;
    if (Parent <> nil) and
       (TPublicWinControl(Parent).Color = LabelColor)
      then LabelColor := clGray;
    inherited Font.Color := LabelColor;
    UpdateFocusedEnabled;
  end;
end;

function TFixedLabel.GetFont: TFont;
begin
  Result := inherited Font;
end;

procedure TFixedLabel.SetFont(Value: TFont);
begin
  inherited Font := Value;
  FOrigColor := inherited Font.Color;
end;

function TFixedLabel.IsFontStored: boolean;
begin
  Result := not ParentFont;
end;

procedure TFixedLabel.UpdateFocusedEnabled;
begin
  if FFocusEnable and (FocusControl <> nil) then
    FocusControl.Enabled := Enabled;
end;

procedure TFixedLabel.SetFocusEnable(Value: boolean);
begin
  if FFocusEnable <> Value then begin
    FFocusEnable := Value;
    UpdateFocusedEnabled;
  end;
end;

procedure TFixedLabel.CMDialogChar(
  var Message: TCMDialogChar);
begin
  if Enabled then
    inherited;
end;

procedure Register;
begin
  RegisterComponents('BugFixes', [TFixedLabel]);
end;

end.
```

Figure 1 shows the example program running. There are several steps.

At the time of form creation I save the current event handler of the `BitBtn` control into a private declared variable called `FOldOnKeyDown`. Then I re-assign the event-handler to my own one with one simple line. After this, each time the event `OnKeyDown` occurs, first of all the method `OverallKeyDown` is called:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  { Save current event handler }
  FOldOnKeyDown := BitBtn1.OnKeyDown;
  { Assign new event handler }
  BitBtn1.OnKeyDown := OverallKeyDown;
end;
```

Once the event handler is called you can process your own code. Then you should call the previous handler if it is assigned:

```
procedure TForm1.OverAllKeyDown(Sender: TObject;
  var Key: Word; Shift: TShiftState);
begin
  {... your code should be inserted here ...}
  { Call next event-handler in the chain }
  if Assigned(FOldOnKeyDown) then
    FOldOnKeyDown(Sender,Key,Shift);
end;
```

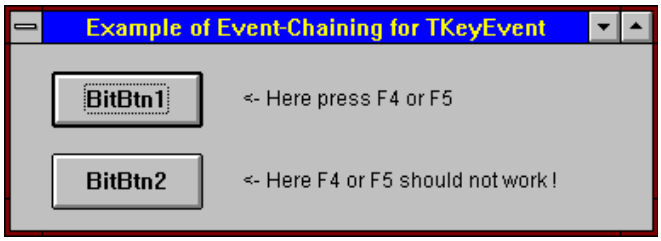
A look at the previous handler shows that it is totally independent of the rest of the program, it does not know that it is called at the end of the chain:

```
procedure TForm1.BitBtn1KeyDown(Sender: TObject;
  var Key: Word; Shift: TShiftState);
begin
  if Key = vk_F4 then begin
    MessageDlg('F4 is pressed on BitBtn1 ',
      mtInformation, [mbOk], 0);
    Key := 0;
  end;
end;
```

After everything is done the event handler should be restored to its correct value. In this case it is not necessary, because after the form is destroyed the event handler doesn't play a role any longer, but to demonstrate how it is done:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  { Restore previous event handler for BitBtn1 }
  BitBtn1.OnKeyDown := FOldOnKeyDown;
end;
```

The example program shows two `BitBtn` controls in a form, but only the first acts on `F4` and `F5` key presses. `F4` is handled by the standard event handler and `F5` is handled by the other event handler. In the example I only use one handler to chain to the old one, but it is



➤ *Figure 1*

also possible to use this technique over a several handlers. It would interesting to manage the list of previous handlers into a `TList` class descendant in the same way as `AddExitProc` acts *[Any takers? Editor]*.

For component developers this technique is also very interesting because you can hook an event property of the `TForm` into which the component is inserted. For example, each time the form receives an `OnHide` event, your component receives it also. I have already tested it by hooking the `FormCreate` method to a component.

Contributed by Stefan Boether, whose email address is 100023.275@compuserve.com