# Tips & Tricks

This is ***your*** column! Here is your opportunity to share with your fellow Delphi enthusiasts those hard-won hints and helps that make your life easier day by day. Please do send in your Tips & Tricks to us (preferably by email, to the Editor, at 70630.717@compuserve.com, or alternatively on disk), whether large or small, on any aspect of Delphi or related issues. We're looking forward to hearing from you!

## Popup Menus

A question arose at a Delphi Developers' Group meeting in London recently. How do you find out which component caused a popup menu to pop up? Remember that a popup menu may be associated with many components. No one knew an immediate answer, and so I set to work looking for a solution. I came up with the techniques programmed in Listing 1 before finding the documented `PopupComponent` run-time property. If a right-click on a component caused a menu to pop up, this property points to that component. I just wasted all that time working out my own method! Or so I thought.

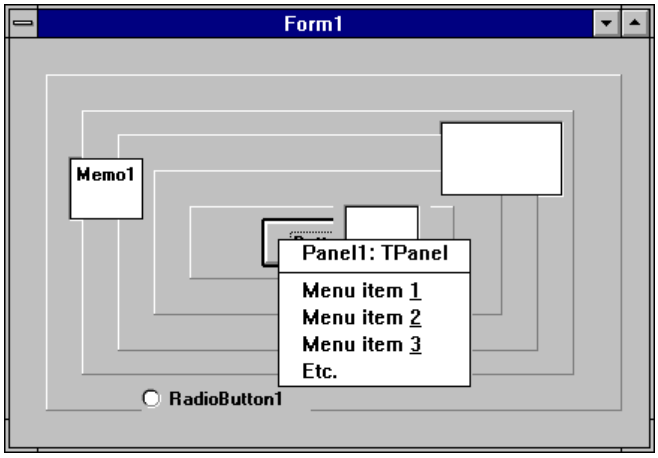
It seems `PopupComponent` doesn't work consistently. It often gives mis-information, as shown in Figure 1: when I right-click over a button that's on some panels, the popup thinks that `Panel1` is under the cursor. So, let's dig that old code out of the bin and persevere. But there's another thing we can do...

Delphi uses popup menus in the form designer. However, when you right click, a menu pops up which is specific to the currently selected component, not the component under the mouse cursor. Delphi can generate its popup using a keystroke, `Alt-F10` and again, the selected component is used.

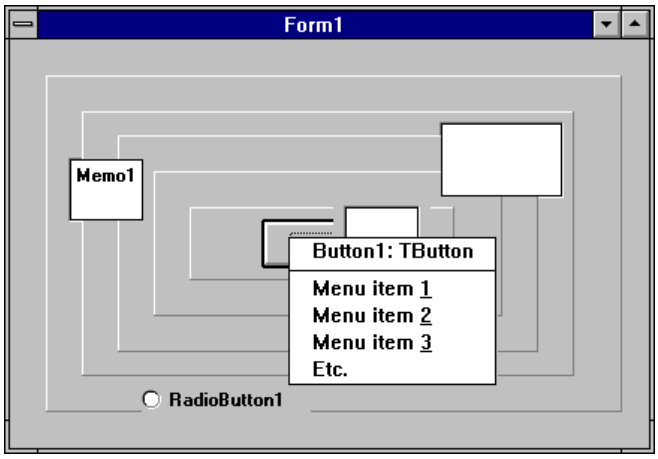
Given that VCL popups work on the current cursor position, it may also be desirable to allow a keystroke (`Alt-F10`) to popup a menu, but again, make it specific to the component under the cursor, if there is one.

We can use the technique implemented in Listing 1 to do this. The menu that's generated is a bit like a Paradox for Windows popup. The first item is descriptive (in this case it tells you the component name and class) and non-selectable. It is disabled, but not greyed out. If there is no component under the cursor, the top menu item and its separator are hidden.

The `Alt-F10` keystroke is trapped by setting the form's `KeyPreview` property to `True` and using an




➤ *Figure 1 PopupComponent gets it wrong...*




➤ *Figure 2 FindComponentAtCursor gets it right!*

`OnKeyDown` event handler. It then pops up the menu at the current mouse position.

To find which component is under the mouse cursor when a popup menu pops up we need to implement an `OnPopup` event handler for the menu itself and then do some exploratory work. The `PopupMenu1Popup` event handler in Listing 1 calls `FindComponentAtCursor` to do the job. `FindComponentAtCursor` returns the appropriate component, and the event handler sets the first menu item's caption to the name and class of the component. It then ensures the menu item is disabled, but not greyed, by using a Windows API call.

The `FindComponentAtCursor` routine works by identifying the current position of the cursor and then using another API call to identify the handle of the window that contains the cursor position. This handle is passed to `FindControl` to identify which `TWinControl`-based component has that handle as its `Handle` property.

Contributed by Brian Long (whose email address is 76004.3437@compuserve.com)

## Form Design Quick Keys

Delphi makes designing the visual interface of your applications so much easier, but there are tricks that can make design even slicker! If you find it a pain

```
unit Popupsu;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, Menus, StdCtrls,
  ExtCtrls, Grids, Outline;
type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
    Panel3: TPanel;
    Panel4: TPanel;
    Panel5: TPanel;
    Button1: TButton;
    PopupMenu1: TPopupMenu;
    DummyItem: TMenuItem;
    Memo1: TMemo;
    ListBox1: TListBox;
    RadioButton1: TRadioButton;
    Notebook1: TNotebook;
    Outline1: TOutline;
    N1: TMenuItem;
    Menuitem1: TMenuItem;
    Menuitem2: TMenuItem;
    Menuitem3: TMenuItem;
    EtcItem: TMenuItem;
    procedure PopupMenu1Popup(Sender: TObject);
    procedure FormKeyDown(Sender: TObject;
      var Key: Word; Shift: TShiftState);
  private
    { Private declarations }
  public
    { Public declarations }
    function FindComponentAtCursor: TWinControl;
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}

function TForm1.FindComponentAtCursor: TWinControl;
var  Pt: TPoint;
begin
  GetCursorPos(Pt);
  Result := FindControl(WindowFromPoint(Pt));
end;
procedure TForm1.PopupMenu1Popup(Sender: TObject);
begin
  with PopupMenu1 do begin
    PopupComponent := FindComponentAtCursor;
    { Write to the menu first, to make sure it is
      brought into life. If you do this last, the
      EnableMenuItem call will have had no effect,
      since the menu won't actually exist }
    if PopupComponent <> nil then
      DummyItem.Caption := PopupComponent.Name + ': ' +
        PopupComponent.ClassName;
    { No component of ours under cursor  so get rid of
      menu item ... }
    DummyItem.Visible := PopupComponent <> nil;
    { ... and seperator }
    N1.Visible := PopupComponent <> nil;
    { Disable the dummy menu item, but _don't_ grey it
      out. The Enabled property does grey the menu item
      when set to False }
    EnableMenuItem(Handle, DummyItem.Command,
      mf_ByCommand or mf_Disabled);
  end;
end;
procedure TForm1.FormKeyDown(Sender: TObject;
  var Key: Word; Shift: TShiftState);
var  Pt: TPoint;
begin
  GetCursorPos(Pt);
  if (ssAlt in Shift) and (Key = vk_F10) then
    PopupMenu1.Popup(Pt.X, Pt.Y);
end;
end.
```

➤ *Listing 1*

moving controls on your forms into exactly the right position using the mouse, try these:

- ➢ Use `Ctrl` plus the cursor keys to move the current control on the form in one pixel increments;
- ➢ Use `Shift` plus the cursor keys to re-size the current control on the form in one pixel increments;

Also, pressing the `Esc` key while a control is selected on a form passes the focus to the underlying control or form.

Contributed by Tony McKiernan

## More Editor Shortcuts

Ever wanted to remove a column of text at the end of your lines? For example:

```
Statement1; { Comment 1 }
Statement2; { Comment 2 }
Statement3; { Comment 3 }
Statement4; { Comment 4 }
Statement5; { Comment 5 }
Statement6; { Comment 6 }
Statement7; { Comment 7 }
```

Suppose you wanted to remove the Comments. Normally you would have to remove them one by one by going to the beginning of the comment and pressing `Ctrl-Q-Y` (to remove all the text up to the end of the line). The new Borland IDEs (Delphi 1.0 and BC++ 4.x) however support a new feature: you can now *mark a column of text*. To do this using the Default or Classic keyboard mapping:

***Using the keyboard:***

- ➢ Type `Ctrl-O-C` (to enter column selection mode),
- ➢ Now select the part you want to remove, by using the Shift and cursor keys,
- ➢ After this you can remove the text by pressing `Ctrl-Del` (or `Shift-Del` to cut it to the clipboard).

This method is also ideal if you want to *swap* two columns of text: just mark the column, cut it to the clipboard and paste it where you want it. To return to the normal selection mode, press `Ctrl-O-K`.

***Using the mouse:***

- ➢ Use `Alt Left Mouse Button` to select a block of text.

You can also try out the other marking methods like inclusive block marking or line marking (`Ctrl-O-I` and `Ctrl-O-L` respectively). Here's some more editor tips:

- ➢ Ever wanted to put a complete word into uppercase or lowercase in the IDE? While the cursor is in the word, press `Ctrl-K-E` to change it to lowercase, or press `Ctrl-K-F` for lowercase.
- ➢ Ever wanted to go directly to a specific line number in the IDE? Press `Ctrl-O-G` and then enter the line number you want to go to.

Contributed by Arjan Jansen

## Replacing if..then..else

Rather than using clumsy `if..then..else` statements such as:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  if Button1.Top + Button1.Height div 2 <
    ClientHeight div 2 then
    Button1.Caption := 'Top Half'
  else
    Button1.Caption := 'Bottom Half';
end;
```

you can take advantage of the anomalous typed constant construct in Delphi, and also the fact that arrays can have elements indexed by any ordinal type. So, the condition above becomes:

```
procedure TForm1.FormResize(Sender: TObject);
const
  Captions: array[False..True] of String[11] =
    ('Bottom Half', 'Top Half');
begin
  Button1.Caption :=
    Captions[Button1.Top + Button1.Height div 2 <
    ClientHeight div 2];
end;
```

Notice that we are using the result of a Boolean expression to index the array, which was set up with Boolean element indices. Another example uses a compound statement in the `if` statement:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  if Button1.Top + Button1.Height div 2 <
    ClientHeight div 2 then begin
    Button1.Caption := 'Top Half';
    Button1.Enabled := True;
  end else begin
    Button1.Caption := 'Bottom Half';
    Button1.Enabled := False;
  end;
end;
```

A simplification of this becomes:

```
procedure TForm1.FormResize(Sender: TObject);
const
  Captions: array[False..True] of
    String[11] = ('Bottom Half', 'Top Half');
begin
  Button1.Enabled :=
    Button1.Top + Button1.Height div 2 <
      ClientHeight div 2;
  Button1.Caption := Captions[Button1.Enabled];
end;
```

Contributed by Brian Long

## Are We In Range?

If we want to check if an ordinal value is in a particular range, or is one of a number of values, many third and fourth generation languages instill the following techniques in us:

```
if (TestValue >= 3) and (TestValue <= 7) then
  ShowMessage('It''s between 3 and 7');
if (TestValue <> 3) and (TestValue <> 5) and
  (TestValue <> 7) then
  ShowMessage('Found it');
```

However Object Pascal offers us alternative constructs to express this with:

```
if TestValue in [3..7] then
  ShowMessage('It''s between 3 and 7');
if not (TestValue in [3, 5, 7]) then
  ShowMessage('Found it');
```

These expressions make use of *sets* which are a convenient aid to reducing typing and making code (in my opinion) more readable. Sets have limitations here though: they only take values which take one byte, in other words no `Integer`, `Longint` or `Word` variables, amongst others. So the following *won't* work:

```
if not (Message.Msg in [wm_LButtonDown,
  wm_LButtonDblClk]) then
  inherited WndProc(Message);
```

To cater for other types, we can avoid saying:

```
if (Message.Msg <> wm_LButtonDown) and
  (Message.Msg <> wm_LButtonDblClk) then
  inherited WndProc(Message);
```

by using a `case` statement. This allows you to specify ranges and lists of values, as you can in a set, but without the one byte restriction. So, for example, the following may be a preferable scheme (bear in mind that in many cases the tests we need to perform in our programs may be rather larger than these simple ones):

```
case Message.Msg of
  wm_LButtonDown, wm_LButtonDblClk: ;
else
  inherited WndProc(Message);
end;
```

Contributed by Brian Long

**Thanks to all our contributors to this issue's Tips & Tricks column. Sorry to those who submitted Tips that we didn't have space for in this issue, they're in the file for Issue 4!**