# The Delphi CLINIC

**Edited by Brian Long**

**Problems with your Delphi project? Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com**

## Printing Web Pages

**Q** Further to your *Using Internet Explorer* item in Issue 47 (July 1999) of *The Delphi Magazine* I have been trying, with little success, to print the currently displayed web page.

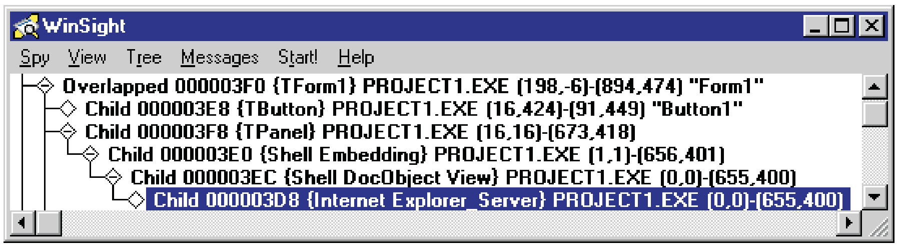
Microsoft's knowledge base suggests that setting the focus to the web browser and then sending `Ctrl+P` to the control by using the `KeyBd_Event` function will generate printed output. This works fine if you have not removed focus from the control since it was created.

But I have found that once keyboard focus has been set to an alternative control (such as a button that gets pressed) it seems impossible to get focus back to the original web browser handle. I've compared the browser's `Handle` property with the return value from `GetFocus` and the values are different.

As a result, `Ctrl+P` only works when the web browser is the active control when the form starts. After this, when the other window handle has focus, `Ctrl+P` no longer works.

**A** The entry in *The Delphi Clinic* referred to in the question covered using Internet Explorer from within a Delphi 4 application by importing it as an ActiveX, or controlling it either with Automation via a `Variant`, or with COM through its interfaces. Delphi 5 introduced the `TWebBrowser` component on the `Internet` page of the Component Palette that gives direct access to the Internet Explorer object (as a pre-installed ActiveX), but the question of how to print a displayed page remains.

Firstly, let's look at this business of the window handles. I made a simple mock-up of the application



```
Overlapped 000003F0 {TForm1} PROJECT1.EXE (198,-6)-(894,474) "Form1"
  Child 000003E8 {TButton} PROJECT1.EXE (16,424)-(91,449) "Button1"
  Child 000003F8 {TPanel} PROJECT1.EXE (16,16)-(673,418)
    Child 000003E0 {Shell Embedding} PROJECT1.EXE (1,1)-(656,401)
      Child 000003EC {Shell DocObject View} PROJECT1.EXE (0,0)-(655,400)
        Child 000003D8 {Internet Explorer_Server} PROJECT1.EXE (0,0)-(655,400)
```

➤ *Figure 1: The windows that make up a TWebBrowser control.*

```
procedure TForm1.actPrintPageExecute(Sender: TObject);
begin
  try
    WebBrowser.ExecWB(OLECMDID_PRINT, OLECMDEXECOPT_PROMPTUSER)
  except
    on E: EOleException do begin
      E.Message := 'Printing cancelled';
      raise
    end
  end
end;
procedure TForm1.actPrintPageUpdate(Sender: TObject);
begin
  (Sender as TAction).Enabled :=
    WebBrowser.QueryStatusWB(OLECMDID_PRINT) or OLECMDF_ENABLED <> 0
end;
```

➤ *Listing 1: Using an action to print a web page, when appropriate.*

described in the question, with a panel containing a `TWebBrowser` (set as the active control at design-time) and a button. A quick browse across the form with WinSight (as shown in Figure 1) tells us how the control is laid out in the eyes of Windows.

You can see that the `TPanel` window has a child window, whose class name is `Shell Embedding`, which has a child window of its own, of class `Shell DocObject View`. This window also has a child, whose class name is `Internet Explorer_Server`.

To see how focus changed between these windows, I selected WinSight's `Spy | Follow Focus` menu item so it would keep track of the focused window for me. It turns out that when the program starts, the `Shell Embedding` window is focused. However, if you either click on another control (such as a button) and back on the browser, or just click directly on the browser, the `Internet Explorer_Server` window gets focus.

This confirms what the questioner suggests about focus changes. However, in my testing (with Internet Explorer version 4) it made no difference which of these windows had focus: a manual press of `Ctrl+P` worked regardless. Given this, I would expect `KeyBd_Event` to demonstrate the same behaviour.

Irrespective of how well the keystroke performs against the web browser, it seems that this is not the preferred way of invoking a print operation from it. Instead, we can do it using one of the methods of the web browser control.

For a general overview of programmatic printing from Internet Explorer, you should refer to the article *Printing with the Internet Explorer WebBrowser Control*, by Microsoft's Dave Templin. I found this as one of the pages of the Web Workshop in the Platform SDK documentation on the MSDN Library CD. If you have that CD, then from *Platform SDK*, choose *Web Services*, then *Workshop*, then *Reusing Browser Technology*, then *Browser Overview*, and then *Printing with the WebBrowser Control*. You can also find the article online at http://msdn.microsoft.com/workshop/browser/wb_print.asp.

The advice given is to pass an appropriate command ID for a print operation to the `ExecWB` method after checking the command is available with `QueryStatusWB`. I've tested this out and it works successfully. Listing 1 shows the implementation of an action that prints the current web page through its `OnExecute` event handler, but whose `OnUpdate` event handler ensures it is only enabled if printing is supported.

The command for printing (`OLECMDID_PRINT`) is passed as the only parameter to `QueryStatusWB`, which returns a bitmask value. If the bits represented by the `OLECMDF_ENABLED` constant are set, the command is available. To print a page, `OLECMDID_PRINT` is passed as the first parameter to `ExecWB`, and the second parameter allows you to specify that the normal printer settings dialog is used.

One point to note is that if the user cancels the printer settings dialog, the printing is aborted and an exception is raised to indicate this. However, the message in the exception is very unrepresentative of the problem, so the code catches the error and changes the error message.

A sample project that uses the action in conjunction with a `TWebBrowser` component is on the disk as called IEPrint.dpr. You can see it running in Figure 2, with its print button enabled. This button is connected to the print action, so the fact that it is enabled indicates printing support is available.

## Unit Ambiguity

**Q** I am trying to call a Win32 API routine (`SetParent`). I have found that it is declared in the Windows.pas import unit but cannot see how to call it. Whenever I try, I get an error because the compiler thinks I want to call the VCL `SetParent` method, which expects one `TWinControl` parameter, not two window handles as expected by the API routine, which I am passing.

**A** This is the age-old scope issue, present in most programming languages. There is the potential for some identifier (a constant, variable, procedure, function or type) to be declared in multiple locations and the challenge is to get the compiler to understand which one you want to refer to.

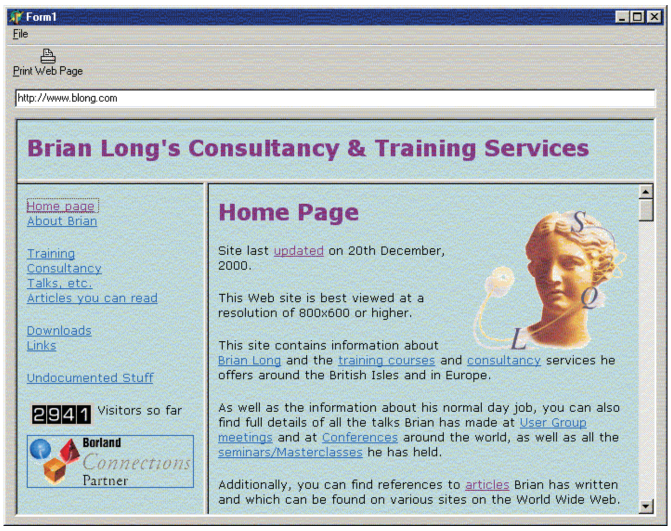
The scope of an identifier is the range of instructions over which the identifier is known, and so can be directly accessed. An identifier is visible within its scope and invisible outside it; however, it is possible to override scope in many cases (we'll come back to this later). The location of an identifier's declaration determines its scope, as determined by a set of rules, some of which rely on an understanding of the term *block*.

Delphi Object Pascal is often described as a block-structured language, as any program is built out of various blocks. A block is the `begin..end` part of a program, procedure, function or method, along with any declarations that immediately precede it (parameters, variables, constants, resource strings, types, procedures, functions, and labels).

Given this definition, here are the scope rules of ObjectPascal:

- If the identifier is declared in the declaration of a program (in an application or DLL project source file), function or procedure, its scope goes from the declaration point to the end of that block.
- If the identifier is declared in the interface section of a unit, its scope goes from that point to the end of the unit, and to any unit or program that uses the unit.
- If the identifier is declared in the implementation section of a unit (outside of any function or procedure), its scope goes from the declaration point to the end of the unit.
- If the identifier is declared as a field in a record definition, its scope goes from the declaration point to the end of the record definition.
- If the identifier is declared as a data field, property or method in a class, its scope goes from the declaration point to the end of the class type definition, and will also include descendants of the class and the code in the methods of the class and its descendants. An exception to this is that items declared in the `private` section of a class will not be in the scope of descendant classes declared in any other unit.

When a program is running and a line of code is executing, a number of nested scopes will be active. For example, if it is compiling a line of code in a method, the method scope is the active scope. The next scope in the current 'scope hierarchy' will be the class scope, followed by the scope of the current unit, followed by the scope of the last unit in the implementation



Site last updated on 20th December, 2000.

This Web site is best viewed at a resolution of 800x600 or higher.

This site contains information about Brian Long and the training courses and consultancy services he offers around the British Isles and in Europe.

As well as the information about his normal day job, you can also find full details of all the talks Brian has made at User Group meetings and at Conferences around the world, as well as all the seminars/Masterclasses he has held.

Additionally, you can find references to articles Brian has written and which can be found on various sites on the World Wide Web.

➤ *Figure 2: Preparing to print a web page.*

section uses clause. Each preceding unit adds another nested scope and the same applies to the interface section uses clause.

When you make a reference to an identifier, the compiler checks each scope in turn, starting with the active scope, looking for a declaration of that identifier. Note that the compiler will not automatically look at the fields in a record variable, or the fields of an object variable when searching for an identifier.

If the identifier you seek is in one scope, but a closer scope defines an identifier with the same name, your target identifier (the *outer declaration*) will be hidden by the closer one (the *inner declaration*). This is referred to as a *naming conflict* and needs to be overcome.

You can (usually) resolve naming conflicts using *qualified identifiers*. A qualified identifier allows you to refer to an identifier declared in a specific scope, assuming there is a way of identifying that scope. We should already be familiar with some forms of qualified identifiers, as we use them to access public or published properties and methods of objects, or fields of forms. When you refer to Button1.Caption, you are qualifying the reference to the Caption identifier with the reference to the scope of Button1. The same applies to accessing record fields.

It is less common to qualify references to global identifiers in a unit, but no less valid. The reason we are less familiar with qualifying references to identifiers in units is that the compiler will automatically look in unit scopes, as long as the units are in a uses clause. But there comes a time (when naming conflicts arise) when we have no choice but to do it.

To try and help understand the naming conflicts problem in general and how to overcome it where possible, let's look at a contrived example, where the same identifier is defined in a whole variety of places.

Take the case of a project (ScopeTest.dpr) with two forms (MainForm and OtherForm) that live in two form units (ScopeTestMainForm.pas and ScopeTestOtherForm.pas). MainForm has a public Integer function which is called Foo declared and OtherForm declares a public Integer data field called Foo. The ScopeTestMainForm unit declares an Integer variable Foo in the implementation section and the ScopeTestOtherForm unit declares an Integer function Foo in the interface section of the unit.

The main form has two buttons on it. The event handlers for both buttons declare a local Integer constant called Foo. The first button is going to access five identifiers all called Foo: the local constant, the main form function, the variable in the same unit, the other form's Integer data field and the other unit's Integer function.

Any simple unqualified reference to Foo will access the local constant, so all the other references must be qualified. Listing 2 shows how to qualify the other references. Self can be used to qualify an identifier declared in the same class, whilst an object reference (such as OtherForm) can be used to qualify references to an identifier defined in another class. Finally, getting back to the original question, you can qualify an identifier in any unit by using the unit name.

A twist to this arrangement occurs when a with statement is used. A with statement enters another nested scope, that of the specified object or record. If you are in a scope orchestrated by a with statement, you can still access most items you need that are suffering from a naming conflict, with the exception of local variables or constants, parameters to the subroutine and nested routines, as shown in Listing 3. From inside the with statement, you can directly access identifiers declared in the other form and its unit, but you cannot access local identifiers which have naming conflicts.

➤ *Listing 2:*
*Qualified identifier references to overcome naming conflicts.*

```
procedure TMainForm.Button1Click(Sender: TObject);
const
  Foo: Integer = 57;
begin
  //Access local variable, parameter or nested routine
  ShowInt(Foo);                      //57
  //Access data field or method of this class
  ShowInt(Self.Foo);                 //1
  //Access data field or method of other class
  ShowInt(OtherForm.Foo);            //7
  //Access variable or routine from this unit
  ShowInt(ScopeTestMainForm.Foo);  //99
  //Access variable or routine from other unit
  ShowInt(ScopeTestOtherForm.Foo); //100
end;
```

➤ *Listing 3:*
*An additional scope causes potential problems for local identifiers.*

```
procedure TMainForm.Button2Click(Sender: TObject);
const
  Foo: Integer = 57;
begin
  with OtherForm do begin
    //Access local variable, parameter or nested routine
    //Cannot be done from inside this with statement
    //Access data field or method of this class
    ShowInt(Self.Foo);               //1
    //Access data field or method of other class
    ShowInt(Foo);                    //7
    //Access variable or routine from this unit
    ShowInt(ScopeTestMainForm.Foo);  //99
    //Access variable or routine from other unit
    ShowInt(ScopeTestOtherForm.Foo); //100
  end
end;
```

## Moved Components Lose Events

**Q** I have taken a tab control from my Delphi application and changed the parent window, so it appears on another application's window (I am trying to add some behaviour to another application). What troubles me is that the `OnChange` event assigned to the control no longer works when it is not a child of a Delphi form. Have you experienced anything like this before?

**A** I haven't noticed the problem personally, but I can understand why it might occur. It's all to do with the way that component events are synthesised from Windows messages and how those messages are delivered.

The problem arises because of how Windows applications were originally intended to be developed. Microsoft supplied pre-built controls in the operating system (such as the button, edit and listbox controls) and you created instances of them and made them children of your window.

In order to customise the behaviour of the window as a whole, you wrote a window procedure for it. The window procedure was given a whole variety of messages as and when needed, which the programmer may or may not be interested in responding to. A large `case` statement was used to filter out the messages of interest and respond to them.

Because of this general architecture, it made sense that when something happened to a control, such as a button getting clicked or a character being typed in an edit control, the message describing the occurrence was sent to the parent window's window procedure. This type of message is called a *notification message* and there are several of them, including `WM_COMMAND` and `WM_NOTIFY` to name but two.

In the Delphi environment, we are not so used to this notification idea. When someone presses a button, the button has its own event that fires. Granted, the event handler will probably be defined as a form method, but the location of the event handler is irrelevant. The key point is that the button component knows when something happens to itself, and it fires its own event when it does.

But a button component is just an object wrapper around a Windows button control that still sends notification message to its parent. The VCL works some internal trickery to enable the button to have the knowledge required to trigger its `OnClick` event.

Whenever a notification message is sent to a form (or any `TWinControl`), the form then immediately sends an internal version of the same message (called a *component notification message*) back to the control that the message relates to. The message has the same parameters associated with it, but has `CN_BASE` added onto the message number. `WM_COMMAND` becomes `CN_COMMAND` and `WM_NOTIFY` becomes `CN_NOTIFY`. In fact, Delphi defines a whole batch of component notification message constants in the `Controls` unit. When the control picks up the component notification message it has the opportunity to trigger an appropriate event.

The problem in the question occurs because the tab control has been given a parent that is not a Delphi `TWinControl`. Consequently, when the new parent window receives notification messages regarding the tab control (such as the one that indicates a new tab has been selected), it does nothing with them.

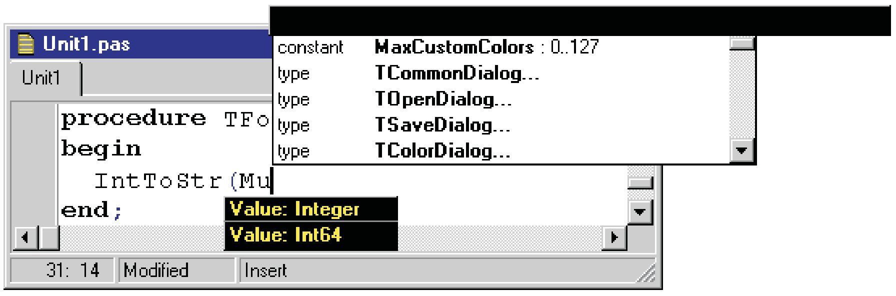
To overcome the problem I recommend placing the tab control (or whatever it happens to be) on a form whose `AutoSize` property is `True`, and whose `BorderStyle` is set to `bsNone`. `AutoSize` was introduced in Delphi 4 and will make the form shrink to be exactly the same size as the tab control. With the form ready, you should change the parent window of this form, rather than the control. The result of this will be that the tab control will still have a VCL parent and so the component notifications will still be sent to the tab control.

To change the parent window of an existing instance of the form, you can use the `SetParent` API (see the *Unit Ambiguity* entry above for details on avoiding a potential problem with calling it). The first parameter is the window handle of the control that is to be moved (the form's `Handle` property) and the second parameter is the handle of the new parent (you can locate a window in another application with the `FindWindow` API). Alternatively you can construct the form with the `CreateParented` constructor which takes a window handle and makes the form be a child to that window.

## Unreadable Tooltips

**Q** I often use the Code Completion feature during development, but recently I noticed a problem with it. When an item in the list is too long to display in its entirety, you can place your mouse over it to display a tooltip which then shows the whole item. However, the tooltip does not seem to use the system tooltip font colour and plays havoc on my system.

You see, I use a colour scheme where tooltips have a yellow font on a black background (the reverse of the normal setup). The Code Completion tooltip displays with a black background, as per Control Panel, but uses a black font regardless of the Control Panel setting, making the tooltip useless. Is there a setting in Delphi which can fix this (I'd rather not change my


➤ *Figure 3: Whilst Code Parameters tooltips behave, Code Completion tooltips might not.*

```
procedure TFo
begin
  IntToStr(Mu
end;
```

```
constant   MaxCustomColors : 0..127
type       TCommonDialog...
type       TOpenDialog...
type       TSaveDialog...
type       TColorDialog...
```

```
Value: Integer
Value: Int64
```

Windows settings for the sake of one tooltip).

**A** Having changed my own Windows settings around, I can see that this is a problem. Figure 3 shows the editor's Code Parameters tooltip using the system colours correctly, but the Code Completion window rather fruitlessly using black on black.

This is a bug in that particular IDE tooltip class, which is called `TKibitzHintWindow` (the Code Completion window is referred to internally as the kibitz window). Whilst it uses `clInfoBk` for the background, it uses `clBlack` instead of `clInfoText` for the font colour. I have reported the problem, which is present in Delphi 3, 4 and 5, so hopefully it will be fixed for Delphi 6.

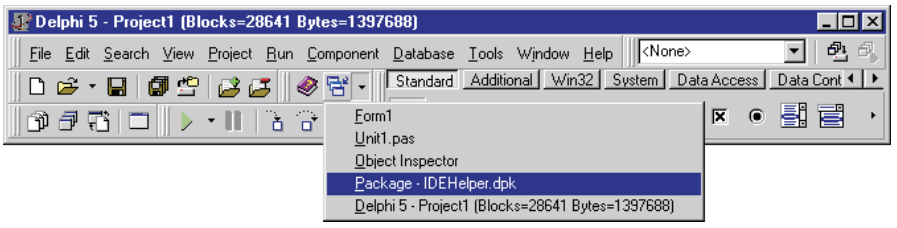
As the questioner says, one resolution is to change the Windows settings, but the request is for a setting to remedy the problem. Unfortunately, since this is a bug, there is no provision in the IDE to change the colour.

I tried to rustle up an expert that could change the `Font.Color` property used by the tooltip window, but I couldn't get it to work. Either the tooltip ignores its own `Font.Color`, or a new tooltip window instance is created each time it is required. So for the time being, changing your Windows settings is the only solution to this problem.

Incidentally, you will probably notice that the tooltip only lasts 2.5 seconds, which makes it difficult to read long function declarations in time before it disappears. I discussed how to remedy this problem by increasing the value of the IDE's `Application.HintHidePause` property in the *IDE Dissatisfaction* entry in *The Delphi Clinic*, back in Issue 43 (March 1999).

## Elusive Package Editors

**Q** I am starting to use packages more and more, but get quite frustrated trying to locate the package editor which seems to keep disappearing behind the source code editor. Is there a keystroke to make the package editor come to the foreground?


Delphi 5 - Project1 (Blocks=28641 Bytes=1397688)
File Edit Search View Project Run Component Database Tools Window Help
<None>
Standard Additional Win32 System Data Access Data Cont
Form1
Unit1.pas
Object Inspector
Package - IDEHelper.dpk
Delphi 5 - Project1 (Blocks=28641 Bytes=1397688)

➤ *Figure 4: The window list drop-down menu.*

**A** I can fully understand your frustration with trying to keep the package editors available. Many Delphi users dock them into the editor or Object Inspector to keep them accessible, so that would be one option to consider.

As for a keystroke to bring a package editor to the foreground, the best you can find in the basic IDE is `Alt+0`. This is the keyboard shortcut for `View | Window List...`, which produces a dialog with a listbox populated by the captions of all available windows in the IDE. You can double-click on the item that matches your package editor's caption and that window will be brought to the foreground.

This solution itself is not that good, because you then get a modal window that you must dismiss to get the package editor displayed. A possible improvement here is to add this window list command onto one of the IDE toolbars in Delphi 5 or later. If you right-click on any toolbar and choose `Customize...` you can select tool buttons that represent most IDE menu items from the `Commands` page of the customisation dialog.

Select `View` from the `Categories:` list and `Window List...` from the `Commands:` list, and drag it onto any toolbar. The tool button that gets created as from version 5 has a small drop-down arrow next to it. If you press this arrow, you get a drop-down menu of all the window captions that can be selected (see Figure 4).

Incidentally, the tool button that corresponds to `File | Open...` has a drop-down button next to it which shows the same things as choosing `File | Reopen`. Also, the `Run | Run` menu tool button has a drop-down button which allows you to select the active project

➤ *Listing 4: Some expert snippets.*

```
const
  MenuBarItemCaption = '&Clinic';
  MenuItemCaption = '&Show package editors';
  MenuItemKey = VK_F2;
  MenuItemShifts = [ssCtrl, ssAlt];
...
constructor TPackageExpert.Create;
var
  FMainmenu: TIMainMenuIntf;
begin
  inherited;
  if Assigned(ToolServices) then begin
    FMainMenu := ToolServices.GetMainMenu;
    if Assigned(FMainMenu) then
      try
        FMenuBarItem := FMainMenu.GetMenuItems.InsertItem(
          FMainMenu.GetMenuItems.GetItemCount-1,
          MenuBarItemCaption, '', '', 0, 0, 0, [mfEnabled, mfVisible], nil);
        FMenuItem := FMenuBarItem.InsertItem(0,
          MenuItemCaption, '', '', ShortCut(MenuItemKey, MenuItemShifts),
          0, 0, [mfEnabled, mfVisible], MenuItemClick);
      finally
        FMainMenu.Free
      end
  end
end;
...
procedure TPackageExpert.MenuItemClick(Sender: TIMenuItemIntf);
var
  I: Integer;
begin
  for I := 0 to Screen.FormCount - 1 do
    if CompareText(Screen.Forms[I].ClassName, 'TPackageEditorForm') = 0 then
      Screen.Forms[I].BringToFront
end;
```

from those set up in the current project group.

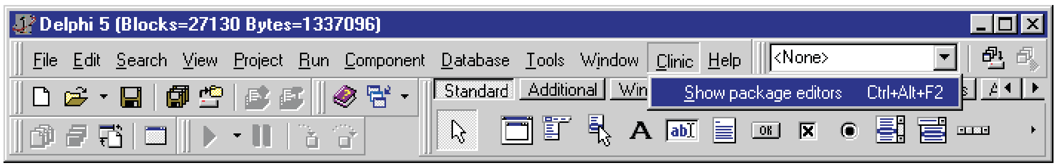
So this new tool button gets rid of the modal window, but that doesn't answer the question of whether there is a dedicated keystroke for the job. The answer to this question is 'not in the product as supplied in the box', but it is not difficult to change this answer to 'yes, with a little creativity'.

To provide a solution to the problem, I have written a small expert whose sole job is to bring any package editors open in the IDE to the foreground. It sits as a menu item in a menu of its own, and has a shortcut key associated with it. I chose `Ctrl+Alt+F2`, but this can easily be changed in the source code.

The package is called DCLPkgEdt50.dpk, which is a Delphi 5 package. The important code is in the PackageExpert.pas unit, which can easily be added to a new package created in other versions of Delphi. The key code from the unit can be seen in Listing 4. The expert constructor locates the IDE's main menu and inserts into it a new top level menu. Then it adds a new menu item into that menu and sets up an `OnClick` event handler for it.

The event handler uses Delphi's `Screen` object to find all the package editors. All forms in any VCL application have references maintained in the `Screen.Forms` array, in which there are `Screen.FormCount` elements. Each form has its class name compared with that of a package editor. If a match is found, the form is sent to the front of the Z-order, so it can be seen. The expert's menu item is visible in Figure 5.

By the way, if you wondered where the IDE's `Window` menu came from in Figure 4 and Figure 5, as well as the oddly formed Component Palette tabs, the answer is from my IDE add-in package called Archaeopteryx (discussed in an article in Issue 27). The usual reason people install this add-in is to get a multi-line Component Palette, but I disabled that option for these screenshots. You can download Archaeopteryx from the *Downloads* area of my website, www.blong.com.


Delphi 5 (Blocks=27130 Bytes=1337096)
File Edit Search View Project Run Component Database Tools Window Clinic Help
<None>
Standard Additional Win
Show package editors Ctrl+Alt+F2

➤ *Figure 5: The new expert menu item.*