Salvador Jover – salvador@sjover.com

Modales por favor... (Parte I)

Salvador Jover salvador@sjover.com

Vamos a hablar, en esta ocasión, sobre algunos aspectos de las ventanas modales y sobre algunas de las cosas que siempre se dicen, porque todo el mundo las conoce. ¿o no...?

Introducción increíblemente básica.

Las primeras lineas de cualquier escrito suelen ser cruciales para quien las escribe, porque no siempre se tiene clara la forma mas correcta de enfocarlas, en este caso concreto en forma de artículo, si es que se me permite la licencia de llamarlo así. Si el tema es complejo porque es complejo..., y si es absurdamente sencillo porque es sencillo... pero el caso es nunca ponerse de acuerdo. De cualquier forma, estas lineas van dirigidas a las personas que se inician en Delphi, que dan sus primeros pasos en el entorno, y no dudo que resultarán obvias para muchos compañeros, teniendo en cuenta que las cuestiones que se pueden plantear, obedecen a los primeros meses de nuestra experiencia en la herramienta. Yo al menos lo veo así.

Hoy en día, afortunadamente, existe información abundante en Internet sobre cualquier tema relacionado con Delphi: desde los -digámoslo- normales, hasta los mas rebuscados. Trucos, faqs, foros, una patada y surgen como setas, pero siempre me queda la idea de que en el fondo, en algunos la idea que subyace no es aprender... sino solucionar una necesidad. Las necesidades son importantes y hay que cubrirlas pero a la larga tenemos el peligro de convertirnos en buscadores: buscadores de trucos, de información rápida, de recetas milagrosas... Y prueba de ello son algunas de las webs donde se hacinan componentes y componentes, miles, apretuñados bajo un cartel que dice algo así como: ¡lo que necesites está aquí... por algún lado pero te aseguro que está! Ni el que lo ha puesto sabe ya lo que tiene... y la verdad es que tampoco creo que le importe demasiado.... Y en los foros de Delphi, creo que se respira en ocasiones parecidos aires, fruto de la misma mentalidad: llegan a ser hasta graciosas algunas preguntas del tipo “tengo que entregar una aplicación mañana a las 5 y ando perdido... así que necesitaría que me enviaseis todo el código fuente que tengáis sobre ... (aquí la petición). No me dejéis tirado porque de verdad que es muy urgente....”. Nosotros... tu y yo no tenemos tanta prisa.

Empecemos a pensar un poco, y centremos -si os parece- nuestra atención en el tema que nos ocupa... Es difícil suponer que la acción se desarrolle siempre sobre una única ventana, ficha o Form, (a gusto vuestro como queráis llamarla), aunque esto no es cierto del todo, o es una verdad a medias. Existen -por ejemplo- las aplicaciones de tipo consola, los servicios, etc., o incluso podríamos considerar si queréis rizar el rizo, la existencia de una única ventana y jugar con la creación de Frames, que nos permite trabajar en tiempo de diseño, pero por lo general, y en un buen número de casos, por no decir en todos, el flujo de nuestro programa dependerá de una ventana principal y una serie indeterminada de ventanas, que serán, o bien creadas, o bien activadas en demanda de esta. Es el esquema más habitual. El interfaz de usuario que conocemos, compuesto normalmente por un menú y por varias barras de botones, situadas en la parte superior del área cliente de nuestra ventana. Si al final hay entre los lectores algún purista que acabe diciendo que “todo” es una ventana en Windows y ... ¡pero bueno!, entre nosotros y para andar por casa creo que se puede asumir el llamar “ventana” a los descendientes de la clase TForm, al menos dentro de este contexto. Si os parece vamos a suponerlo así. Es posible que en artículos posteriores, podamos hacer un hueco pequeño para hablar de los *frames* (TFrame), que comparten en la jerarquía de la VCL con respecto a TForm, una misma ascendencia hasta la clase TscrollingWinControl. Ahora estamos abordando el tema

concreto desde la perspectiva de las ventanas Modales, que son aquellas, según la guía misma de Desarrollador en Delphi 5, "*en las que el usuario debe efectuar alguna accción antes de pasar a otra ficha...*", una definición algo estrambótica para denotar simplemente, que la ventana debe obligatoriamente ser cerrada para que nuestro usuario pueda acceder al resto de ventanas de nuestra aplicación. En tanto permanezca abierta dicha ventana, nuestro usuario se obligará a ejecutar determinadas acciones en el modo previsto por el desarrollador, en una actitud un tanto dictatorial si cabe y recordando aquella actitud que manteníamos frente a las aplicaciones de MSDos. Por el contrario, las ventana no modales, son aquellas que pueden ser activadas o desactivadas a discreción del usuario, sin necesidad de que sean cerradas para entregar el foco a cualquier control de otra ventana que permanezca oculta tras ella

Nuestro supuesto de trabajo, es que estáis dando vuestros primeros pasos en Delphi. Ya conocéis el marco de trabajo y la mecánica de operación con la fichas en tiempo de diseño, por lo que, habéis pasado determinado número de horas añadiendo componentes y modificando sus propiedades en el inspector de objetos. Habéis hecho ya vuestras primeras aplicaciones: desde el botón que responde a la pulsación con el socorrido "Hola Mundo..." hasta la primera calculadora, o el programa de dibujo que despliega una linea o un rectángulo sobre el canvas de nuestra ventana. ¡¡Mira que puede uno pasar horas haciendo esto!! Todos hemos pasado por esos puntos, en ese o en distinto orden, y como todos, existe un día en el que se nos plantea una aplicación que necesita de más de un ficha o Form. Ese es nuestro punto de partida... hagamos nuestra primera parada si os parece.

Cuando necesito mas de una ventana...

Vamos a empezar por el principio: Abrimos el entorno de Delphi y nos situamos en la opción *File/New* y ya dentro de ésta, elegiremos *Application*. Inmediatamente se abrirá un nuevo proyecto que por defecto tomará como nombre *Project1*, y será cargada en memoria una nueva ficha (*Form1*) que representará la ventana principal de nuestra aplicación. La parte visual de esta ficha está representada en la ventana de diseño del entorno, responsable de que podamos insertar aquellos componentes que deseemos formen parte del interfaz de nuestro usuario, en tiempo de diseño. Ilusión óptica al fin y la cabo, si acabáis descubriendo la otra vista de la misma ventana de diseño y que se corresponde con el fichero de texto dfm. Y por otro lado, el inspector de objetos nos va a ayudar a acceder a las propiedades publicadas de nuestros componentes (no confundir con propiedades publicas), que van a poder ser modificadas por nosotros en tiempo de diseño.

Este es el esquema habitual de cualquier proyecto que se inicia:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

La ventana principal de la aplicación, permanecerá mientras el programa este ejecutándose, y a poco que lo penséis bien, es en la última de las rutinas en donde se desarrolla todo el flujo de la aplicación, en tanto no sea cerrada la ventana principal o se invoque el método *Terminate* del objeto Application. Es decir, en la ejecución del método *Run* de Application.

Por otro lado, y desde el punto de vista de la ficha o form, el esquema por defecto del módulo que lo contiene será:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

end.
```

De aquí lo que más nos interesa, a efectos de este artículo, es la linea de declaración de la variable *Form1*, en donde se declara, en la parte pública de nuestro módulo, una referencia a una instancia de la clase TForm1. Cuando el entorno crea el módulo '*.pas', da a la variable el mismo nombre que el tipo declarado en el mismo, pero sin la T inicial. Sois libres de modificar todo esto, pero no os lo recomiendo. Mas que nada por seguir un poco las convenciones.

Puede convertirse esta referencia, en una forma sencilla de acceder a nuestra ventana, pero ni es obligatoria en las nuevas ventanas adicionales, ni ciertamente la única forma de acceso a las mismas. No se si me explico. El entorno nos facilita una variable global al módulo como parte de una mecánica de trabajo, que intenta buscar la mayor sencillez para acceder a la invocación de métodos o asignación de propiedades desde otros módulos. Una vez incluido el nombre de nuestro módulo en el **uses** de otro diferente, el interfaz público del primero queda visible para el segundo y por ende, la variable pública declarada.

Se puede programar ignorando la variable global. La ventana principal es la única que se puede salvar del experimento. Sin ir más lejos podríamos considerar una aplicación del tipo MDI en donde una de las propiedades del objeto Applicatión, nos mantiene una referencia a la ventana hija activa (*Application.MainForm.ActiveMdiChil*). Se podrían haber creado el resto de ventanas con variables locales a procedimientos y funciones, y seguríamos teniendo acceso a nuestras ventanas hijas. No pasa nada.

Así que convenimos que dicha declaración es obligatoria, para al menos la ventana que ha de convertirse en principal, lo que nos deja en uno de los primeros dilemas que se nos presentan y que nos habla sobre la necesidad de definir qué fichas han de ser cargadas en memoria al inicio de nuestro programa y sobre aquellas otras que deben ser creadas, cargadas en memoria, en tiempo de ejecución. El sentido común y la experiencia suele ser una buena guía para resolver estos conflictos. La guía del desarrollador nos dice: "*No siempre es deseable tener cargadas en memoria a la vez todas la fichas de una aplicación. Para reducir la cantidad de memoria necesaria en el momento de cargar la aplicación, es recomendable no crear algunas fichas hasta que se necesite usarlas...*" , lo cual parece indicado a propósito de los cuadros de diálogo, como conviene lineas después, que son tan solo necesarios al momento de la acción de nuestro usuario. Sin embargo, el uso de las ventanas modales, como veremos en el próximo artículo se puede extender a otros uso distintos del cuadro de diálogo específico.

No nos rasguemos las vestiduras si decimos que no hace falta siquiera para crear la ventana. Variable global o local o incluso no existir. Veamos el ejemplo que nos pone la misma Borland con respecto a un hipotético archivo de proyecto al modo “squash”.

```
begin
  Application.Initialize;
  with TForm5.Create(nil) do
  try
    ProgressBar1.Max := 100;
    Show;   // show a splash screen contain ProgressBar control
    Update; // force display of Form5
    Application.CreateForm(TForm1, Form1);
    ProgressBar1.StepBy(25);
    Application.CreateForm(TForm2, Form2);
    ProgressBar1.StepBy(25);
    Application.CreateForm(TForm3, Form3);
    ProgressBar1.StepBy(25);
    Application.CreateForm(TForm4, Form4);

    ProgressBar1.StepBy(25);
  finally
    Free;
  end;
  Application.Run;
end.
```

Si decidiéramos, pese a la advertencia y desoyendo cualquier indicación, que se hace necesario tener todas las ventanas de nuestra aplicación cargadas en memoria, Delphi se ocuparía de generar las lineas de código necesarias a tal efecto, incluyendo las rutinas de creación de las ventanas, en el archivo de proyecto, y justo antes de la invocación del método Run de Application. Supongamos que sean dos, las fichas que deben ser creadas:

```
...
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);  //es creada la primera ficha
  Application.CreateForm(TForm2, Form2);  //es creada la segunda
  Application.Run;
...
```

A partir de ese momento, Form1 referencia a una instancia de la clase TForm1, y la variable Form2 hace lo propio respecto a TForm2, siendo el programador, libre de usarlas para acceder al interfaz (propiedades y métodos) que hacen público cada uno de estos objetos.

Si os apetece pensar un poco mas sobre esto, vamos a hacer lo siguiente. Sobre nuestro proyecto inicial, hagamos algo con poco sentido, como añadir una nueva ficha para así tener dos (en la barra de botones corresponde con un botón cuyo hint es “New Form”) y eliminaremos de ambas, tanto de Unit1 como de Unit2, la cláusula de declaración de las variables Form1 y Form2. Basta con comentarlas añadiendo las dobles barras (//). Tiene que quedar alto tal que así, tanto en Unit1 como en Unit2:

```
//var
//  Form1: TForm1;
```

Hecho esto, trasladémoslas al archivo de proyecto, incluyendo además dos lineas para que las variables puedan acceder a los métodos *Show* y *ShowModal* respectivamente. En otras palabras:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  Unit2 in 'Unit2.pas' {Form2};
```

```
{$R *.res}

var
  Form1: TForm1;
  Form2: TForm2;

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Form1.Show;
  Form2.ShowModal;
  Application.Run;
end.
```

Además, y para acabar de redondear el código, un tanto sin sentido, que acabamos de escribir, vamos a añadir dos botones a nuestro Form2 en la ficha de diseño. Pulsáis sobre la ficha Form2 para que se convierta en la ficha activa de la ventana de diseño, y tras hacer clic en la paleta de componentes sobre TButton en la pestaña *Standard*, volvéis a hacer clic sobre la ficha, quedando inserta en la misma el componente. Una vez hecho esto, damos código a cada uno de los botones haciendo doble clic sobre ellos.

El código resultante, tras añadir los dos botones y las dos lineas mencionadas es:

```
unit Unit2;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm2 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

//var
//  Form2: TForm2;

implementation


{$R *.dfm}

procedure TForm2.Button1Click(Sender: TObject);
begin
  Application.MainForm.Close;
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
  Screen.Forms[0].Close;
end;

end.
```

Podemos recapitular, antes de ejecutar el programa en el Ide con la pulsación del botón cuyo icono asemeja al símbolo Play en color verde, tan conocido de nuestros reproductores de música: Tenemos un

proyecto con dos fichas que son creadas en memoria desde el principio, en las que hemos eliminado la declaración de variables, que vamos a hacer desde el mismo proyecto, añadiendo unas lineas de código en el mismo, para que lance la segunda ventana en forma modal. ¿Cual es el resultado de ejecutar este extraño proyecto?. Veamos...

La rutina *ShowModal* de Form2, producirá que se visualice la segunda de las fichas y que adquiera el foco, y que en tanto no sea cerrada, no podamos acceder a la primera. Anecdóticamente, y para que se pudiera ver claramente, se ha añadido la primera referencia a Show que hace visible a Form1, que de otra forma y en esas condiciones quedaría oculta. Lo podéis ver en la **figura 1**. La pulsación del primer botón, provocaría que la aplicación concluyera su ejecución, dado que siendo Form1 la ventana principal (la primera que se ha creado), y recibiendo el mensaje de cierre de la misma, ejecutaría las acciones necesarias para que la aplicación finalice. La pulsación del segundo botón, provocaría que la ventana activa se cierre, retornando el foco a Form1.

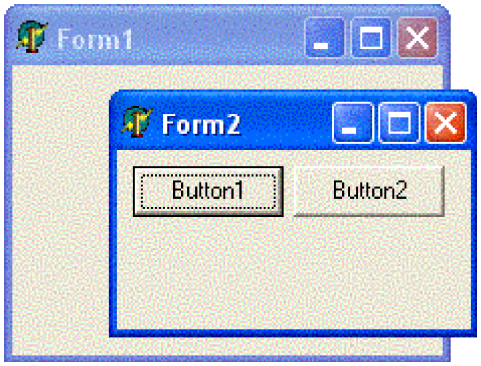

Figura 1: Ejecución del programa.

En el **listado 1** tenéis el código correspondiente al método *ShowModal* invocado por la ventana 2, y que ejecuta uno de sus ascendentes: TCustomForm. Retened en vuestra memoria el bucle **repeat ... until** y en el que se puede leer una linea de código (se ha resaltado de amarillo en el listado)

```
if Application.FTerminate then ...
```

Esta linea tan peculiar relaciona el mantenimiento del bucle con la recepción del mensaje WM_QUIT, estableciéndose como una de las condiciones de salida del mismo. Volveremos sobre esto un poco más tarde. Tened paciencia.

Pero no sigamos adelante sin volver sobre el hecho anecdótico de situar la declaración de las variables en el mismo proyecto. Digo ciertamente anecdótico porque no es algo que se tenga como correcto, sino todo lo contrario, si nos dejamos llevar por un uso lógico de Delphi. La idea era únicamente llamar la atención, de que el lector pudiera percibir claramente que la vida y el ámbito de esta variable es algo diferente de la vida y el ámbito del objeto al que representa. A mi me ha servido para destacar y compartir con vosotros varios aspectos: el más importante es que el objeto (en este caso una ventana) una vez creado, existe aun a pesar de su variable, y que esta es tan solo un puntero, una referencia hacia el (hacia una zona determinada de memoria). Da igual, ciertamente que la variable tenga un ámbito global o lo tenga local a efectos del objeto creado. Repercutirá en que se pueda perder el acceso al mismo desde dicha referencia, si como en el caso de ser local a un procedimiento o función, es eliminada de la pila la variable, pero el objeto en sí seguirá existiendo en tanto no sea liberada la memoria reservada para el.

En el caso que nos ocupa en este momento, he elegido en la pulsación de sendos botones, la ejecución de los métodos de dos instancias de las clases TScreen y TApplication respectivamente (es decir el objeto Screen y el objeto Application), para acceder a las fichas y a la invocación de métodos propios de las mismas. Lo cual nos deja la idea de que no existiendo las variables, existen métodos alternativos para acceder a las ventanas y en general a todos los controles.

Sigamos con el experimento... Nos situamos ahora en la ventana Form2, y desde dentro de la implementación de cualquier de los dos eventos del clic del botón, intentemos compilar el programa tras añadir la linea:

```
Form1.Close;
```

El resultado es que nuestro compilador se quejará, no sin razón, puesto que no conoce desde ese ámbito la variable Form1 ni tenemos forma alguna de acceder a la misma. Ya no nos vale el **uses** puesto que el archivo de proyecto no es una unidad (*.pas) y se nos demandará el compilado (*.dcu).

Creo que ya podemos volver a dejar todo tal y como estaba, y proseguir de un forma algo más normal... tenemos que dar marcha atrás y desandar parte del camino andado. Ya podéis eliminar la declaración de las variables incluidas en el proyecto, y volver a situarlas en cada módulo respectivo, tal y

```
function TCustomForm.ShowModal: Integer;
var
  WindowList: Pointer;
  SaveFocusCount: Integer;
  SaveCursor: TCursor;
  SaveCount: Integer;
  ActiveWindow: HWnd;
begin
  CancelDrag;
  if Visible or not Enabled or (fsModal in FFormState) or
    (FormStyle = fsMDIChild) then
    raise EInvalidOperation.Create(SCannotShowModal);
  if GetCapture <> 0 then SendMessage(GetCapture, WM_CANCELMODE, 0, 0);
  ReleaseCapture;
  Include(FFormState, fsModal);
  ActiveWindow := GetActiveWindow;
  SaveFocusCount := FocusCount;
  Screen.FSaveFocusedList.Insert(0, Screen.FFocusedForm);
  Screen.FFocusedForm := Self;
  SaveCursor := Screen.Cursor;
  Screen.Cursor := crDefault;
  SaveCount := Screen.FCursorCount;
  WindowList := DisableTaskWindows(0);
  try
    Show;
    try
      SendMessage(Handle, CM_ACTIVATE, 0, 0);
      ModalResult := 0;
      repeat
        Application.HandleMessage;
        if Application.FTerminate then ModalResult := mrCancel else
          if ModalResult <> 0 then CloseModal;
      until ModalResult <> 0;
      Result := ModalResult;
      SendMessage(Handle, CM_DEACTIVATE, 0, 0);
      if GetActiveWindow <> Handle then ActiveWindow := 0;
    finally
      Hide;
    end;
  finally
    if Screen.FCursorCount = SaveCount then
      Screen.Cursor := SaveCursor
    else Screen.Cursor := crDefault;
    EnableTaskWindows(WindowList);
    if Screen.FSaveFocusedList.Count > 0 then
    begin
      Screen.FFocusedForm := Screen.FSaveFocusedList.First;
      Screen.FSaveFocusedList.Remove(Screen.FFocusedForm);
    end else Screen.FFocusedForm := nil;
    if ActiveWindow <> 0 then SetActiveWindow(ActiveWindow);
    FocusCount := SaveFocusCount;
    Exclude(FFormState, fsModal);
  end;
end;
```

Listado 1. Implementación del método ShowModal en la clase TCustomForm

como estaba al principio. Posiblemente, para alguno de vosotros, se produzca el efecto secundario siguiente: darse cuanta de que existe un archivo de proyecto y de que puede manipularse. Para mi, en los primeros pasos con el entorno, concretamente con Delphi 2.0 y acompañado de una pequeña enciclopedia de programación, fue una sorpresa... pensaba que tan solo existían los módulos (.pas) o por lo menos que el (.dpr) era un ser extraño modificable solo por nuestro compilador... Suele ser habitual encontrar ejemplos de pantallas de presentación, al inicio de la ejecución de nuestra aplicación, que hacen uso del archivo de proyecto.

¿Habéis echo ya la modificación que hemos comentado?... Ahora mismo ya tienen que estar colocadas la declaración de las variables en la posición original. Eliminamos también del archivo de proyecto las lineas que acceden a los métodos Show y ShowModal, quedando en la misma situación en que estaba al principio del experimento. Vuelve a quedar...

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  Unit2 in 'Unit2.pas' {Form2};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

Serán creadas y cargadas en memoria antes de que se inicie la ejecución. En el orden descrito. Modificad ademas desde el inspector de objeto el valor de la propiedad Visible de Form2, asignandola a True. Ejecutad la aplicación nuevamente (F9). El resultado es que aparecen las dos ventanas. El foco en la ventana principal y Form2 se hace visible pero queda en un segundo plano. Ambas han sido creadas desde el método Show aunque haya sido de echo esto de forma implícita, por lo que podemos activar una u otra, dependiendo sobre que control o zona señalemos el clic de nuestro ratón.

Desde el momento en que hemos incorporado de nuevo la declaración de las variables en sus respectivos módulos, nos basta incluir una referencia en el uses del módulo para que la variable sea accesible desde el que ha sido añadida. Supongamos de nuevo el módulo Unit1, donde tenemos declarado la clase TForm1:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

```
implementation

uses Unit2;  // <----- Linea añadida

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Caption:= 'Hola. Estoy jugando con mis ventanas';
end;

end.
```

Si analizamos la situación, tras la ejecución del programa, ahora ya nos es posible acceder a la variable Form2 y modificar el título de nuestro form mediante su propiedad *Caption*, dado que la variable hace referencia a una zona válida de memoria, el objeto existe y ha sido creado con anterioridad. Aunque no sea este caso concreto, (puesto que Form2 existirá durante la ejecución del programa), si que nos adelanta uno de los problemas a los que se enfrenta el programador que se inicia en este tipo de entornos y que está muy relacionado con el uso de métodos desde referencias de memorias no válidas.

Vamos a forzar que sea así. Generemos una violación de la memoria. ¿Os parece...? Para ello y desde el entorno, vamos a eliminar de *Autocreate* la ficha 2. Elegimos *Project/Options* y dentro de ésta, en la pestaña *Forms*, desplazamos la ficha Form2 hacia la zona *Available Forms*. Automáticamente, Delphi modificó nuestro proyecto eliminando la linea:

```
Application.CreateForm(TForm2, Form2);
```

De nuevo ejecutamos la aplicación, pero esta vez, la pulsación del botón 1 provocará una excepción desde el momento en que intentamos ejecutar la asignación de la propiedad Caption sobre una referencia invalida. La ficha a la que referencia Form2 todavía no ha sido creada y la variable apunta a **Nil.** Si queréis verlo con vuestros propios ojos tan solo tenéis que poner un punto de parada en la linea de asignación y observar desplazando el ratón sobre la variable tras la pulsación del botón y la obtención de la parada. Se resaltará el valor **Nil**.

Para poder acceder correctamente a la variable en la nueva situación nos bastaría con crear previamente la segunda ficha. Lo vamos a hacer de las dos siguientes formas:

Primero añadiremos a la pulsación del botón en la ficha Form1 el siguiente código:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Form2: TForm2;
begin
  Form2:= TForm2.Create(Self);
  Form2.Visible:= True;
end;
```

A su vez, Form2 también va a ser modificado en la implementación de uno de los botones:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
  Form2.Close;
end;
```

Ejecutad la aplicación de dos formas: una tal y como está, y la otra comentando el código de declaración de la variable local

```
procedure TForm1.Button1Click(Sender: TObject);
//var
//  Form2: TForm2;
...
```

¿Conclusiones?

Cuando no se comenta la declaración de variable local, la asignación de la referencia Form2,

```
Form2:= TForm2.Create(Self);
```

Form2 hace referencia a la variable declarada localmente, y como tal, cuando salimos del ámbito del procedimiento o función, es eliminada de la pila, liberándose la memoria que ocupa la misma variable. Si apunta a un objeto creado este sigue existiendo y corremos el peligro de no poder acceder al mismo si fuera el caso. En ese caso concreto, la asignación posterior, tras la pulsación del botón, en la que intentamos hacer visible la ventana 2, con la misma lógica, la variable Form2 sigue apuntando a Nil, dado que nunca fue asignada.

En el caso de que deseemos comentar las lineas de la declaración local de la variable, observaremos que Form2 ya no genera excepción alguna, porque precisamente la creación de la ventana asignó la referencia a la zona de memoria válida donde se aloja la instancia de la clase: Hablamos de la variable pública declarada en Unit2 y a la que establecimos un vinculo al añadir en el cláusula **uses** de Unit1 la referencia a dicho módulo.

Os puede parecer una tontería, pero os aseguro que involuntariamente se pueden caer en este tipo de errores de la forma más tonta y cuando menos lo esperamos: cuando invocamos un método sin comprobar anteriormente que la referencia no apunta a Nil (se suele hacer mediante la función *Assigned( )*). Tenedlo en cuenta siempre. Es una regla de Oro a seguir: No dar por seguro el valor de una referencia, en un contexto en el que pueda quedar invalidada.

Por otro lado está la otra lección: El ámbito. El ejemplo nos ha valido para comprender que la creación de Form2 no depende de que la variable sea local o global, dado que lo que realmente importa es la ejecución del método constructor de la instancia de clase, momento en el que se reserva memoria para la misma. Y enfocando el tema desde el lado contrario: su destrucción tampoco depende de la existencia o no de la variable que lo creó. ¿Veis que este matiz ya no quedaba tan claro? Dependerá de que en algún momento se invoque al método que sea, llamémosle x, que inicie la liberación de toda la memoria reservada.

ShowModal o Show... ¿con cual me quedo?. (Próximamente).

La mayoría de los programadores que como yo, son extremadamente perezosos, encuentran que la creación de un formulario Modal es algo cuasi perfecto... ideal en aquellas ocasiones en las que debemos situar a nuestro usuario en situación de elegir, y sin derecho a seguir avanzado en tanto no se haya hecho dicha elección. Ese será el tema que va a ocupar buena parte del segundo artículo, donde intentaré abordar lo más claramente posible, el por qué a pesar de todo, algún programador que se inicia en Delphi se atasca en ese punto. Continuaremos también la explicación que iniciábamos en el **listado 1**, intentando comprender que es lo que sucede exactamente cuando invocamos el método *ShowModal*.

Nos vemos. Hasta entonces, recibid un saludo,

Salvador Jover