MexDelphis shortcuts

Nachdem immer wieder die Frage auftaucht, wie bekomme ich eine Klasse aus einer dll, habe ich meine Beispiele zu diesem kleinen Tutorial zusammengefasst.

Es gibt zwei einfach zu handhabende – delphieigene – Lösungen für das Problem. Zum einen mithilfe von Virtual; abstract .. zum anderen mit Interfaces.

Virtual; Abstract;

Als erstes erstellen wir eine dll mithilfe des dllExperten – es erscheint eine Unit mit einem freundlichen Hinweis zum Umgang mit Strings und dll´s. An sich ist hier nichts ungewöhnlich, lediglich steht am anfang nicht Unit sondern library. Für dieses Tutorial habe ich die library virtualabstracttest getauft und gespeichert – dies ist später auch der Name der dll.

Jetzt fügen wir noch 3 weitere Units in diese dll mit ein und speichern diese unter:

Dllexport.pas, testklasse.pas und SharedGlobals.pas

Beginnen wir mit **SharedGlobals.pas** und deklarieren einige Datentypen und eine „virtuell-abstrakte“ Klasse.

```
{*-------------------------------------------------------------------------------
SharedGlobals

in dieser Unit wird alles deklariert was sowohl in der dll als auch in der
Anwendung verwendung findet.
-------------------------------------------------------------------------------*}
unit SharedGlobals;

interface

{*-------------------------------------------------------------------------------
einige Datentypen die sowohl in der dll als auch in der Anwendung die diese Unit
mit einbindet verwendet werden können :)
-------------------------------------------------------------------------------*}
type
  P_DB_Vector = ^T_DB_Vector;
  T_DB_Vector = record
     x,y,z,w : Single;
  end;

type
  P_DB_Color = ^T_DB_Color;
  T_DB_Color = record
    case Integer Of
    0 : (Red,
         Green,
         Blue,
         Alpha : Single);
    1 : (RGBA  : Array[0..3] of Single);
  end;
```

```
{*-------------------------------------------------------------------------------
alle Methoden sind
 - Virtual(kann in abgeleiteten Klassen überschrieben werden)
 - Abstract(kennzeichnet Teile von Klassen, die grundsätzlich erst in
          abgeleiteten Komponenten definiert und als published deklariert werden)
-------------------------------------------------------------------------------*}
type
  TDB_PointLight = class
    public
      Procedure Render; virtual; abstract;

      procedure SetRadius(Radius : Single); virtual; abstract;
      procedure SetColor(R,G,B : Single); overload; virtual; abstract;
      procedure SetColor(Color : T_DB_COLOR); overload; virtual; abstract;
      procedure SetPosition(x,y,z : single); overload; virtual; abstract;
      procedure SetPosition(Position : T_DB_Vector); overload; virtual; abstract;
      procedure SetRotationCenter(CenterPointX,CenterPointY,CenterPointZ,
StartPointX,StartPointY,StartPointZ : Single ); virtual; abstract;
      procedure RotateArround(RotaX,RotaY,RotaZ: Single); virtual; abstract;
      procedure MoveAdd(x,y,z : Single); virtual; abstract;
      function  MoveTo(Px,Py,Pz, Speed : Single):BOOLEAN; virtual; abstract;

      procedure GetPosition(out x,y,z : Single);  virtual; abstract;
      function  GetPosX : Single; virtual; abstract;
      function  GetPosY : Single; virtual; abstract;
      function  GetPosZ : Single; virtual; abstract;
      procedure GetRotationCenterPos(out x,y,z : Single); virtual; abstract;
      procedure GetRotationCenterRot(out x,y,z : Single); virtual; abstract;
      function  GetRotationCenterRotX : Single; virtual; abstract;
      function  GetRotationCenterRotY : Single; virtual; abstract;
      function  GetRotationCenterRotZ : Single; virtual; abstract;
  end;

implementation

end.
```

Zwei Datentypen und eine Klasse die ein Punktlicht repräsentieren soll. Der eigentliche Clou sind die Wörtchen Virtual, abstract hinter jeder Methode. Ersteres sorgt dafür, dass die Methode in abgeleiteten Klassen überschrieben werden kann – zweiteres sorgt dafür, dass die Methode NICHT implementiert werden muss .. was wir hier auch tunlichst unterlassen ☺

Kommen wir zur nächsten Unit: **testklasse.pas.**

```
unit testklasse;

interface

uses
  SharedGlobals, Dialogs;

{*-------------------------------------------------------------------------------
die "echte" Klasse, abgeleitet von der "virtuell-abstracten" Klasse
alle funktionen/proceduren müssen hier implementiert und überschrieben werden!
-------------------------------------------------------------------------------*}
type
```

```
  TTDB_PointLight = class(TDB_PointLight)
    private
    public
      constructor create(ID : Integer);
      destructor  destroy; override;

      Procedure Render; override;

      procedure SetRadius(Radius : Single); override;

      // auch überschriebene Methoden können überladen werden
      procedure SetColor(R,G,B : Single); overload; override;
      procedure SetColor(Color : T_DB_COLOR); overload; override;
      // nochmal überladen
      procedure SetPosition(x,y,z : single); overload; override;
      procedure SetPosition(Position : T_DB_Vector); overload; override;

      procedure SetRotationCenter(CenterPointX,CenterPointY,CenterPointZ,
StartPointX,StartPointY,StartPointZ : Single ); override;
      procedure RotateArround(RotaX,RotaY,RotaZ: Single); override;
      procedure MoveAdd(x,y,z : Single); override;
      function  MoveTo(Px,Py,Pz, Speed : Single):BOOLEAN; override;

      procedure GetPosition(out x,y,z : Single);  override;
      function  GetPosX : Single; override;
      function  GetPosY : Single; override;
      function  GetPosZ : Single; override;
      procedure GetRotationCenterPos(out x,y,z : Single); override;
      procedure GetRotationCenterRot(out x,y,z : Single); override;
      function  GetRotationCenterRotX : Single; override;
      function  GetRotationCenterRotY : Single; override;
      function  GetRotationCenterRotZ : Single; override;
  end;

implementation

{ TTDB_PointLight }

constructor TTDB_PointLight.create(ID: Integer);
begin
inherited create;
ShowMessage('constructor');
end;

procedure TTDB_PointLight.Render;
begin
ShowMessage('rendert');
end;

destructor TTDB_PointLight.destroy;
begin
ShowMessage('destructor');
inherited destroy;
end;

{... hier die restlichen Methoden implementieren}

end.
```

Diese Unit bindet SharedGlobals ein und deklariert eine Klasse TTDB_PointLight die von TDB_PointLight abgeleitet ist. Da die „Mutterklasse“ alle Methoden als Virtuell deklariert hat, können wir diese hier überschreiben ... und da dort keine Implementation erfolgte da abstract, müssen wir das sogar. Zwei Methoden sind hier sogar noch überladen und müssen hier, wie in der „Mutterklasse“ mit overload gekennzeichnet werden.

Nun haben wir eine „echte Klasse“ und müssen nur noch schauen, dass wir diese irgendwie aus der dll rausbekommen ins reale Applikationsleben ;)

Dies bewerkstelligen wir mithilfe der verbleibenden Unit: **Dllexport.pas**

```
{*-------------------------------------------------------------------------------
Methoden um die Klassen zu erzeugen und die "Virtuell Abstrakten" Klassen
zurückugeben.
-------------------------------------------------------------------------------*}
unit dllexport;

interface

uses
  testklasse, SharedGlobals;

function TDB_PointLightCreate(id : Integer) : TDB_PointLight; stdcall; forward;

implementation

{*-------------------------------------------------------------------------------
die "echte" Klasse wird erzeugt und als "virtuell-abstrakte" Klasse
zurückgegeben
-------------------------------------------------------------------------------*}
function TDB_PointLightCreate(id : Integer) : TDB_PointLight;
begin
Result := TTDB_PointLight.create(id);
end;

end.
```

Hier schreiben wir eine kleine Funktion die, die abgeleitete Klasse TTDB_PointLight erstellt und als „Mutterklasse“ TDB_PointLight zurückgiebt, also sozusagen als „virtuell abstrakte“ Klasse allerdings mit den Implementationen der Methoden der abgeleiteten Klasse.

Diese Unit ist nicht wirklich zwingend notwendig, wenn allerdings mehrere Klassen auf diese art erstellt und auch exportiert (machen wir gleich) werden sollen, ist die Sache so übersichtlicher und leichter zu handeln.

Nun fehlt nur noch der Export aus der dll. Dies bewerkstelligen wir mit der Exports Klausel, die wir in die erste, vom dllExperten erzeugten library einfügen.

Die **library virtualabstracttest** sollte jetzt so aussehen:

```
library virtualabstracttest;

{ blabla über sharemem }

uses
  SysUtils,
  Classes,
  testklasse in 'testklasse.pas',
  SharedGlobals in 'SharedGlobals.pas',
  dllexport in 'dllexport.pas';

{$R *.res}

// hier die Exports klausel
exports
  TDB_PointLightCreate;

begin
end.
```

In die Exports Klausel schreiben wir unsere Funktion die, die erstellte Klasse zurückgibt.

Der wichtige Teil:

Kommen wir jetzt zu dem teil, der es uns ermöglicht die dll in einer Anwendung zu nutzen. Als erstes müssen wir natürlich die dll erzeugen und können dann das Projekt erst mal Schließen. Erstellen wir eine kleine Test.exe, einfach ein kleines Form mit einem Button drauf.

```
unit main;

interface

{*-------------------------------------------------------------------------
in SharedGlobals sind die "virtuell-abstracten" Klassen deklariert, muss hier
mit eingebunden werden.
-------------------------------------------------------------------------*}
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, SharedGlobals, StdCtrls;

{*-------------------------------------------------------------------------
hier wird die Routinen aus der dll geladen.

Achtung Fehlerquelle:
da bei diesem konstrukt jede Routine einzeln importiert werden muss und somit
auch jedesmal eine neue importfunktion getippt wird - sollte man sich angewöhnen,
wenn man zB. die Parameter in der "virtuell-abstracten" Klasse ändert, dies
auch sofort in der entsprechenden importzeile zu tun. Bei folgendem Aufruf
```

wurde nachträglich ein Parameter hinzugefügt und vergessen ihn hier zu übernehmen ...

-> function TDB_PointLightCreate : TDB_PointLight; stdcall;external 'virtualabstracttest.dll';

dieser Aufruf würde anstandslos funktionieren und KEIN Gemecker des Compilers auslösen
aber eine Zugriffsverletzung bei der Freigabe der Klasse bewirken was zu nervigem suchen
führen kann, da die Meldung alles andere als aussagekräftig ist!

Richtig ist so:

```
----------------------------------------------------------------------------*}
function TDB_PointLightCreate(id : Integer) : TDB_PointLight; stdcall;external 'virtualabstracttest.dll';

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

var
  Form1: TForm1;
  Light :  TDB_PointLight;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
Light := TDB_PointLightCreate(1);
light.Render;
light.Free;
end;

end.
```

Diese unit bindet **SharedGlobals** mit ein, und zwar die gleiche Unit, unverändert!

Dies ist der eigentliche Trick an der ganzen sache.
Wir laden hier unsere Funktion TDB_PointLightCreate(id : Integer) aus unserer dll und erzeugen mithilfe der „virtuell abstrakten“ Klasse aus SharedGlobals eine TDB_PointLight-Klasse mit den implementiierten Methoden der abgeleiteten Klasse TTDB_PointLight in der dll.

Interfaces

Die Vorgehensweise bei Interfaces ist absolut identisch, deshalb liste ich hier lediglich die Units auf.

Die dll:

```
{-------------------------------------------------------------------------------
unit SharedGlobals

diese Unit muss sowohl in der dll als auch in der jeweiligen .exe
eingebunden werden! - keine Kopie !!
-------------------------------------------------------------------------------}
unit SharedGlobals;

interface

// das Interface, hier gibt es kein virtual abstract, Interfaces sind immer virtual abstract,
// man kann sich also einiges an ripperei sparen ☺
type
  IMyInterface = Interface
  ['{FD7F94AF-4E74-421B-B753-2DB85774623F}']   // <- Strg + Shift + G
    procedure Info;
  end;

implementation

end.
```

```
{-------------------------------------------------------------------------------
unit testklasse

Die klasse, abgeleitet von TInterfacedObject - dann klappts auch
mit der referenzzählung :) ... implementiert das
interface -> IMyInterface
-------------------------------------------------------------------------------}
unit testklasse;

interface

uses
  SharedGlobals, Dialogs, SysUtils ;


type
  TTestklasse = class(TInterfacedObject,IMyInterface)
    published
      constructor create;
      destructor destroy; override;
      procedure Info; // die procedure aus IMyInterface
    end;

implementation

{ TTestklasse }

constructor TTestklasse.create;
```

```
begin
inherited create;
ShowMessage('constructor');
end;

destructor TTestklasse.destroy;
begin
ShowMessage('destructor');
inherited destroy;
end;

procedure TTestklasse.Info;
begin
// eine Nachricht anzeigen
ShowMessage('hat geklappt');
end;

end.
```

```
unit dllexports;

interface

uses
  testklasse, SharedGlobals;

// diese Funktion exportieren wir aus der dll ..
// als ergebniss wird IMyInterface geliefert
function IMyInterfaceCreate : IMyInterface; stdcall; forward;

implementation

function IMyInterfaceCreate : IMyInterface;
begin
// da TTestklasse IMyInterface implementiert, können
// wir hier direkt das Interface als TTestklasse zurückgeben.
result := TTestklasse.Create;
end;

end.
```

Die Anwendung:

```
{-------------------------------------------------------------------------------
unit Main:

Demo um den export von Interfaces aus einer dll unter Delphi zu demonstrieren.

- lädt 'interfacetest.dll' statisch
- lässt Interface in dll erzeugen
- gibt interface frei

WICHTIG:

die unit  SharedGlobals  muss sowohl in der dll als auch in der exe eingebunden
```

```
sein !!
-------------------------------------------------------------------------------}
unit Main;

interface

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

// Satisches laden der dll
function IMyInterfaceCreate : IMyInterface; stdcall;external 'interfacetest.dll';

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

var
  Form1: TForm1;
  test  : IMyInterface;   // die Interface Variable
  test1 : IMyInterface;
implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
// Interface aus der dll
test := IMyInterfaceCreate;
test.Info;

test1 := test;
test1.Info;

// Interface freigeben
test := nil;
test1 := nil;
end;

end.
```

Wichtig auch hier, das einbinden von SharedGlobals.

Hier noch mal alles als übersicht: