Kurs języka Object/Delphi Pascal

na bazie implementacji Free Pascal.

AUTOR ŁUKASZ STAFINIAK

Email: lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
Web: www.ii.uni.wroc.pl/~lukstafi

Jeśli zauważysz błędy na slajdach, proszę daj znać!

Wykład 8: Metaklasy. Interfejsy.

Zliczanie dowiązań. for..in.

Zagadnienia

Dwie myśli przewodnie:

- Elastyczny polimorfizm.
- Automatyczne zarządzanie pamięcią.

Trzy zagadnienia:

- **Referencje klas** (znane też jako *metaklasy*): typ `class of` i (elementy RTTI) pole `Class`. Konstruktory wirtualne.
- Interfejsy.
- Zliczanie dowiązań vs. model z właścicielem.
  - Zwalnianie obiektów przez kontenery `FPG`.

Jedno uzupełnienie:

- Jeszcze raz o składni `for`..`in`.

Referencje klas

- Pamiętasz metody klas? `class procedure`/`class function`
- Konstruktory też są metodami klas.
- Metody klasy wywołujemy jakby klasa była obiektem.
- Okazuje się, że jest!
  - Typem klasy `TExample` jest:
    ```
    type TExampleClass = class of TExample
    ```
  - Jeśli klasa `TExample` jest `sealed`, to typ `TExampleClass` jest jednoelementowy (singleton).
    - Posługując się żargonem *design patterns*, obiekt `TExample` możemy nazwać singletonem.
  - Jeśli `TX` jest podklasą `TY`, to `TXClass` jest podtypem `TYClass`.

- Klasy są więc w Delphi (/Free) Pascalu *first-class*
  - mamy parametryzację algorytmów i struktur danych typami
    - przy typach generycznych: statycznie,
    - przy referencjach klas: dynamicznie.
- Przekazywanie klas jako argumentów funkcji czy wartości zmiennych ma dopiero sens, gdy klasy te mają wirtualne metody klas. Tylko wirtualne mogą być inne niż z deklaracji zmiennej.
- Zastosowania:
  - Wybór implementacji przez użytkownika,
  - pluginy,
  - algorytmy dynamicznie sparametryzowane ze względu na typy danych,
  - abstrakcyjne algorytmy.

```
program singleton;
uses SysUtils, Classes;
type
   TBase = class
      class var b: Integer;
      class function VGetP: Integer; virtual;
      class procedure VSetP(const I: Integer); virtual;
      class function GetP: Integer;
      class procedure SetP(const I: Integer);
   end;
   TBaseClass = class of TBase;
   TDeriv = class sealed (TBase)
      class var d: Integer;
      class function VGetP: Integer; override;
      class procedure VSetP(const I: Integer); override;
   end;
   TDerivClass = class of TDeriv;
```

```
var
   MyClass0, MyClass1, MyClass2: TBaseClass;
   MyClass3: TDerivClass;
begin
   MyClass0 := TBase;
   MyClass1 := TDeriv;
   MyClass2 := TDeriv;
   MyClass3 := TDeriv;
   MyClass3.VSetP(8);
   MyClass0.VSetP(3);
   MyClass2.VSetP(7);
   MyClass1.SetP(5);
   WriteLn ('MyClass0.VGetP=', MyClass0.VGetP,
            '; MyClass1.GetP=', MyClass1.GetP,
            '; MyClass2.VGetP=',MyClass2.VGetP,
            '; MyClass3.VGetP=', MyClass3.VGetP);
   WriteLn ('TBase.b=', TBase.b, '; TDeriv.d=',TDeriv.d);
end.
```

```
unit plugin_form;

...

procedure TForm1.DeleteCtrlClick(Sender: TObject);
var i: Integer;
begin
  i := ListCtrls.ItemIndex;
  ListCtrls.Items.Objects[i].Destroy;
  ListCtrls.Items.Delete(i);
end;

procedure TForm1.EditCtrlClick(Sender: TObject);
begin
  CurrentCtrl := TEdit;
end;
```

```
procedure TForm1.FormMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  NewCtrl: TControl;
  CtrlName: String;
begin
  NewCtrl := CurrentCtrl.Create(self);
  NewCtrl.Visible := False;
  NewCtrl.Parent := self;
  NewCtrl.Left := X;
  NewCtrl.Top := Y;
  Inc (Counter);
  CtrlName := CurrentCtrl.ClassName + IntToStr(Counter);
  Delete(CtrlName, 1, 1);
  NewCtrl.Name := CtrlName;
  NewCtrl.Visible := True;
  ListCtrls.AddItem(CtrlName, NewCtrl);
end;
```

Interfejsy

Problemy z hierarchią podtypowania bazującą tylko na dziedziczeniu klas:

- Dziedziczenie implementacji niszczy enkapsulację (patrz item 16 „Effective Java" – slajd 16 java-lecture11.pdf).
- Jednokrotne dziedziczenie – nie można zdefiniować nowej klasy by pełniła wiele wcześniej zdefiniowanych ról.
- Słaba separacja specyfikacji i implementacji:
  - część prywatna klasy musi być w interfejsie modułu,
  - implementacja klasy musi być dostępna do skompilowania jej użycia.

Interfejs to zbiór sygnatur (nagłówków) metod definiujący nowy typ. Obiekty tego typu pochodzą z klas *implementujących* interfejs: muszą mieć publicznie wszystkie metody z interfejsu.

Zalety interfejsów w Delphi(/Free) Pascalu: 

- Mniejsza zależność między modułami,
- Jedna klasa może implementować wiele interfejsów-zachowań,
- które w Delphi(/Free) Pascalu mogą nawet mieć metody o tych samych nazwach, a różnych implementacjach w jednej klasie!
- Klasa może agregować zachowania, delegując do osobno zdefiniowanych klas implementację wybranych interfejsów – w Delphi(/Free) Pascalu bez żadnego nadmiarowego kodu.
- Można opublikować interfejs rozprowadzając kod jedynie w postaci skompilowanej.
- Interfejsy mogą tworzyć hierarchię z wielodziedziczeniem.

Wady interfejsów w Delphi(/Free) Pascalu: 

- W języku domyślnie interfejsy związane są ze zliczaniem dowiązań chociaż to zupełnie różne sprawy (cierpią i interfejsy i zliczanie dowiązań),
- Niezależność kompilacji tylko gdy trzymamy interfejsy w osobnych modułach niż (implementujące je) klasy. Pamiętaj by nie rozdmuchać nadmiernie „architektury" programów.

Interfejsy są mniej popularne niż w Javie, częściowo przez popularność referencji klas.

```
program multi_intf;
uses SysUtils, Classes;
{$interfaces CORBA}
type
   IBase = interface
      function ShowBase: String;
   end;
   IA = interface (IBase)
      function Show: String;
   end;
   IB = interface (IBase)
      function Show: String;
   end;
```

```
TMulti = class (IA, IB)
    function Show: String;
    function ShowBase: String;
    function ShowA: String;
    function ShowB: String;
    function IA.Show = ShowA;
    function IB.ShowBase = ShowB;
end;
function TMulti.Show: String;
begin Show := 'TMulti.Show'; end;
function TMulti.ShowBase: String;
begin ShowBase := 'TMulti.ShowBase'; end;
function TMulti.ShowA: String;
begin ShowA := 'TMulti.ShowA'; end;
function TMulti.ShowB: String;
begin ShowB := 'TMulti.ShowB'; end;
```

```
var
   Multi: TMulti;
   Base: IBase;
   A: IA;
   B: IB;
begin
   Multi := TMulti.Create;
   //Base := Multi;
   A := Multi;
   B := Multi;
   WriteLn ('A.ShowBase=', A.ShowBase, '; B.ShowBase=',
B.ShowBase);
   WriteLn ('A.Show=', A.Show, '; B.Show=', B.Show);
   Multi.Free;
end.
```

```
program no_counting;
uses SysUtils, Classes;
type
   IBasic = interface
      function Show: String;
   end;

   TNoCount = class (IBasic)
      function Show: String;
      function QueryInterface(constref IID:TGUID; out Obj)
         :LongInt; CDecl;
      function _AddRef:LongInt; CDecl;
      function _Release:LongInt; CDecl;
   end;
   function TNoCount.Show: String;
   begin
      Show := 'TNoCount.Show';
   end;
```

```
function TNoCount.QueryInterface
   (constref IID:TGUID; out Obj):LongInt; CDecl;
begin
   if GetInterface(IID, Obj) then Result := 0
end;
function TNoCount._AddRef:LongInt; CDecl;
begin
   Result := -1
end;
function TNoCount._Release:LongInt; CDecl;
begin
   Result := -1
end;
```

```
var
   Basic: IBasic;
   NoCount: TNoCount;
begin
   NoCount := TNoCount.Create;
   Basic := NoCount;
   WriteLn ('Basic.Show=', Basic.Show,
            '; NoCount.Show=', NoCount.Show);
   Basic := nil;
   NoCount.Destroy;
end.
```

Zliczanie dowiązań

- O ile nie używamy `{$interfaces CORBA}`, zmienne, pola i parametry typu interfejsowego będą objęte tzw. *zliczaniem dowiązań*. Jest to mechanizm zarządzania pamięcią przejmujący od programisty obowiązek niszczenia obiektów (do których odnoszą się zmienne i pola typu interfejsowego).
- Inny mechanizm zarządzania pamięcią w Delphi/Free Pascalu to model z właścicielem (*ownership*): właściciel obiektu jest odpowiedzialny za jego destrukcję. Jest realizowny m.in. przez klasy pochodne z `TComponent`.
  - Ponieważ komponenty korzystają z interfejsów dla zwiększenia polimorfizmu, `TComponent` implementuje metody obsługi zliczania dowiązań tak by one nic nie robiły. Zupełnie jak w przykładzie `no_counting` powyżej.
- **Ostrożnie przy mieszaniu zmiennych typu klasy i zmiennych typu interfejsowego** do przechowywania tych samych obiektów – gdy zniknie ostatnia referencja poprzez typ interfejsowy, obiekt zostanie zniszczony, i możemy zostać z wiszącą (*dangling*) referencją w zmiennej typu klasy.

```
program bad_counting;
uses SysUtils, Classes;
type
   IBasic = interface
      function Show: String;
   end;

   TBadCount = class (TInterfacedObject, IBasic)
      function Show: String;
   end;
   function TBadCount.Show: String;
   begin
      Show := 'TBadCount.Show';
   end;
```

```
var
   Basic: IBasic;
   BadCount: TBadCount;
begin
   BadCount := TBadCount.Create;
   Basic := BadCount;
   WriteLn ('Basic.Show=', Basic.Show,
            '; BadCount.Show=', BadCount.Show);
   Basic := nil;
   BadCount.Destroy;
end.
```

Rozpoczęcie zliczania dowiązań dla `BadCount`.

Znika jedyna referencja do `BadCount` więc obiekt zniszczony.

Ups – wiszący wskaźnik, naruszenie pamięci.

```
program interfaces_ref_counting;
uses Classes;
var
  C: TComponent;
  IO: TInterfacedObject;
  I: IInterface;
begin
  C := TComponent.Create(nil);
  C := TComponent.Create(nil);
  I := C;
  I := TComponent.Create(nil);
  IO := TInterfacedObject.Create;

  IO := TInterfacedObject.Create;
  I := IO;
  I := TInterfacedObject.Create;
end.
```

Wyciek pamięci: brak zliczania dowiązań.

Wycieki pamięci: co prawda wywołane zliczanie, ale nic nie robi, bo (tu też) `TComponent` je ignoruje.

Wyciek pamięci bo zmienna nie jest interfejsowa.

Brak wycieku, bo tutaj startujemy ze zliczaniem dowiązań.

Preferowany sposób: brak możliwości pozostania z wiszącą referencją.

```
program ref_lifespan;
type
   TClass = class (TObject, IInterface)...
var
   GO: TClass;
   GI: IInterface;
   procedure LocalProc(Arg: IInterface);
   var
      LO: TClass;
      LI: IInterface;
   begin
      WriteLn ('Entered LocalProc');
      Write('LO := TClass.Create; '); LO := TClass.Create('Local A');
      Write('LI := LO; '); LI := LO;
      //raise Exception.Create('test exit');
      //Exit; //Halt;
      Write('LI := GO; '); LI := GO;
      Write('GI := TClass.Create; '); GI := TClass.Create('Local B');
      WriteLn ('Leaving LocalProc');
   end;
begin
   WriteLn ('Entered main program');
   Write('GO := TClass.Create; '); GO := TClass.Create('Global A');
   Write('GI := GO; '); GI := GO;
   LocalProc(TClass.Create ('Global B'));
   WriteLn ('Leaving main program');
end.
```

```
Entered main program
GO := TClass.Create; Creating Global A
GI := GO; Reference Global A increase to 1
Creating Global B
Reference Global B increase to 1
Entered LocalProc
LO := TClass.Create; Creating Local A
LI := LO; Reference Local A increase to 1
LI := GO; Reference Global A increase to 2
Reference Local A decrease to 0
Destroying Local A
GI := TClass.Create; Creating Local B
Reference Local B increase to 1
Reference Global A decrease to 1
Leaving LocalProc
Reference Global A decrease to 0
Destroying Global A
Reference Global B decrease to 0
Destroying Global B
Leaving main program
Reference Local B decrease to 0
Destroying Local B
Heap dump by heaptrc unit
4 memory blocks allocated : 64/64
4 memory blocks freed     : 64/64
0 unfreed memory blocks : 0
```

Po odkomentowaniu `raise Exception.Create('test exit');` – zwróć uwagę że wszystkie obiekty TClass są zwolnione:

```
Entered main program
GO := TClass.Create; Creating Global A
GI := GO; Reference Global A increase to 1
Creating Global B
Reference Global B increase to 1
Entered LocalProc
LO := TClass.Create; Creating Local A
LI := LO; Reference Local A increase to 1
Reference Local A decrease to 0
Destroying Local A
Reference Global B decrease to 0
Destroying Global B
An unhandled exception occurred at $080485BC :
Exception : test exit
  $080485BC  LOCALPROC, ...

Reference Global A decrease to 0
Destroying Global A
Heap dump by heaptrc unit
14 memory blocks allocated : 1165/1184
11 memory blocks freed     : 1065/1080
3 unfreed memory blocks : 100
```

Po odkomentowaniu `Exit;`:

```
Entered main program
GO := TClass.Create; Creating Global A
GI := GO; Reference Global A increase to 1
Creating Global B
Reference Global B increase to 1
Entered LocalProc
LO := TClass.Create; Creating Local A
LI := LO; Reference Local A increase to 1
Reference Local A decrease to 0
Destroying Local A
Reference Global B decrease to 0
Destroying Global B
Leaving main program
Reference Global A decrease to 0
Destroying Global A
Heap dump by heaptrc unit
9 memory blocks allocated : 1017/1024
9 memory blocks freed     : 1017/1024
0 unfreed memory blocks : 0
```

Po zamianie `Exit;` na `Halt;`:

```
Entered main program
GO := TClass.Create; Creating Global A
GI := GO; Reference Global A increase to 1
Creating Global B
Reference Global B increase to 1
Entered LocalProc
LO := TClass.Create; Creating Local A
LI := LO; Reference Local A increase to 1
Reference Global A decrease to 0
Destroying Global A
Heap dump by heaptrc unit
9 memory blocks allocated : 1017/1024
7 memory blocks freed     : 985/992
2 unfreed memory blocks : 32
```

Cykle i „słabe referencje”

- Jak widzieliśmy powyżej, zliczanie dowiązań jest „w miarę” niezawodne: jeśli procedura ma parametr lub zmienną lokalną typu interfejsowego, kompilator umieści jej kod w klauzuli `try`..`finally` żeby zawsze przy jej opuszczaniu dekrementować referencje znajdujące się w jej „ramce stosu”.
  - Dodatkowa klauzula `try` może trochę pogorszyć wydajność.
- Ale zliczanie dowiązań nie zwolni obiektów jeśli tworzą one cykl.
- Gdy musimy mieć dane z cyklicznymi odwołaniami, i zależy nam na zarządzaniu pamięcią przez interfejsy-zliczanie dowiązań, to któreś z referencji trzeba „przeciąć”, np. przez zrzutowanie na wskaźnik uniwersalny `Pointer` albo przez referencję po „oryginalnej” klasie obiektu: byle nie mieć referencji po interfejsie.

Zarządzanie pamięcią w FGL

`TFPGObjectList` zwalnia obiekty, *o ile parametr FreeObjects dla konstruktora jest true* (więc nie jest to konieczny mechanizm).

Zwalnia nie tylko przy niszczeniu listy, także przy usuwaniu elementów z listy przez `Remove`, przez `Clear` etc. `TFPGObjectList` jest w pełni odpowiedzialne za zarządzanie pamięcią elementów. Jako ucieczka, zawsze można wyjąć element z listy przez `Extract`.

`TFPGInterfacedObjectList` wydaje się zbędne: ma zakodowane wywoływanie `_Add` i `_Release`, ale ponieważ `TFPGList` posługuje się zwykłymi przypisywaniami, to kompilator automatycznie wrzuci te wywołania jeśli będziemy przechowywać listę interfejsów w liście.

Jeszcze raz o składni `for..in`

Szczegóły i przykłady: patrz `http://wiki.freepascal.org/for-in_loop`.

- By korzystać ze składni `for..in` wystarczy by klasa kontenera implementowała metodę `GetEnumerator`, zwracającą obiekt-iterator mający metodę `MoveNext` przesuwającą iterator na następny element lub zwracającą `False` jeśli się nie da, oraz właściwość (tylko do odczytu) `Current` zwracającą aktualny element.
- Zawsze twórz obiekt-iterator „jednorazowego użytku", bo będzie zwolniony po użyciu. Nie zwracaj kontenera jako iteratora!
- Iterować można po dowolnym obiekcie posiadającym `GetEnumerator`, więc możesz zaimplementować metody zwracające różne strategie enumeracji, najprościej jako iteratory posiadające dodatkowo `GetEnumerator` zwracającą `self`.
- Dla elegancji, kontener może implementować interfejs `IEnumerable`, i odpowiednio iterator interfejs `IEnumerator`.
- Możesz dorzucić składnię `for..in` do istniejącego typu przeciążając operator `Enumerator`.