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 4: Wyjątki. Moduły.

Typy `class`. Przeciążanie operatorów.

Rzut oka na typy `class`

Jeśli pewne pojęcia poniżej są dla Ciebie niejasne, poznasz je na przyszłych wykładach.

- Wprowadzone przez Delphi typy `class` są nowoczesnym (zbliżonym do Javy) mechanizmem programowania obiektowego.
- Obiekty typów `class` zawsze powstają na stercie, tzn. zawsze są manipulowane przez referencję (tzn. wskaźnik, ale niewidoczny dla programisty).
- Zwiększony polimorfizm dzięki możliwości implementacji wielu interfejsów (jak w Javie).
- Zapobiega pewnym błędom, możliwym przy obiektach typu `object`:
  - przekazywanie obiektu z metodami wirtualnymi przez wartość, gdy klasy pochodne dodają pola do obiektu;
  - odwoływanie się do obiektu bez wywołania jego konstruktora nawet jeśli klasa tego wymaga; albo po wywołaniu jego destruktora (w przypadku `class` mamy `FreeAndNil` w module `SysUtils`).
- **Nie używamy** operatorów `new` i `dispose`, zamiast tego wywołujemy wybrany konstruktor klasy, który zwraca obiekt (dokładniej, referencję).
  - Konstruktory są często nazwane `Create`, a destruktory `Destroy`.

- Deklarujemy klasy `class` tak samo jak klasy `object`, czyli podobnie do `record`, ale z metodami `procedure` / `function` / `constructor` / `destructor`.
- Zmienne typów `class` nie przydzielają pamięci na cały obiekt a jedynie na referencję. Można do nich przypisywać `nil`.
    - Zmienne lokalne nie są zainicjalizowane na `nil`!
    - Pamięć obiektu jest przydzielana przy wywołaniu konstruktora.
- Preferowane zwalnianie obiektu: procedura `FreeAndNil` modułu `SysUtils` ustawia zmienną / pole na `nil` i zwalnia obiekt wskazywany przez tą zmienną / pole.
- Moduł `SysUtils` dostarcza m.in. podstawowych mechanizmów kontroli poprawności programu. Od dzisiaj zawsze będziemy go dodawać do `uses`.
- `TObject` jest przodkiem wszystkich klass: `type TNode = class ... end;` jest równoważne `type TNode = class (TObject) ... end;`
- `TObject` dostarcza „domyślnych" konstruktorów `Create` i `Destroy`.
- `FreeAndNil` wywołuje `Destroy` (więc lepiej trzymać się tej nazwy).

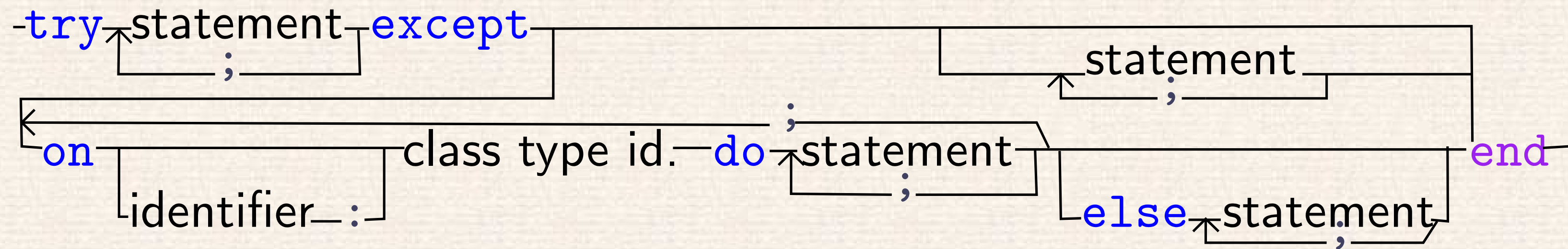
Wyjątki

- Wyjątki we Free/Delphi Pascalu są, podobnie jak w innych językach obiektowych, obiektami dowolnej klasy (typu `class`), ale powinny dziedziczyć ze zdefiniowanej w `SysUtils` klasy `Exception`.
  - Tradycyjnie w Pascalu nazwy klas dziedziczących z `Exception` zaczyna się od `E` zamiast od `T`.
- `Exception` pamięta stringa opisującego wyjątek: będzie wyświetlony jeśli nie złapiemy wyjątku.
- Rzucamy wyjątek instrukcją `raise`:

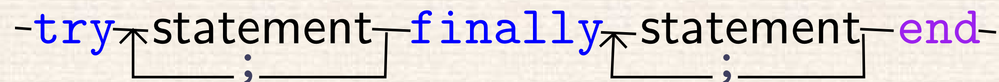
  ```
  raise Exception.Create('test 3');
  ```
  Tworzy i od razu rzuca wyjątek.

  Wykonywanie programu jest od razu przerywane i przeskakuje do najbliższego kodu obsługującego wyjątki.
- Używanie wyjątków na powiadomienie o błędzie, zamiast szczególnej wartości jak `nil`, jest bezpieczniejsze i wygodniejsze!

- Chwytamy wybrane lub wszystkie wyjątki blokiem obsługi wyjątków `try`...`except`.



- Aby zapewnić wykonanie w odpowiednim momencie kodu kluczowego dla poprawności, np. zwalniającego zasoby gdy nie są już potrzebne, używamy `try`...`finally`.



- Obsługę wyjątku typu `EType` umieszczamy w klauzuli `on EType do`… Jeśli potrzebujemy szczegółów: `on E: EType do`… `E` jest zmienną odwołującą się do obiektu wyjątku.
- Po wykonaniu bloku `except`…`end`, jeśli sam nie rzucił wyjątku, wywoływany jest destruktor `Destroy` obiektu wyjątku, obiekt jest zwalniany, obliczenia kontynuują normalnie.
- Wewnątrz obsługi wyjątków możemy użyć `raise;` – czyli `raise` bez instancji (tzn. obiektu) wyjątku – „wywołaj obsługiwany wyjątek ponownie”.
- Jeśli nie ma wyjątku w bloku strzeżonym przez `try`…`except`, to blok `except`…`end` jest pomijany.
- Jeśli żadna klauzula `on`…`do` nie pasuje do wyjątku, to jest wywoływana klauzula `else`, a jeśli jej nie ma to wyjątek jest wywołany ponownie.
  - Jeśli nie chwytamy konkretnych wyjątków `on`…`do`, to nie piszemy `else` tylko od razu obsługę wyjątków.
  - Pascalowe `else` odpowiada `except:` z Pythona; nie ma odpowiednika Pythonowego `else`.

- Blok `finally`...`end` jest zawsze wykonywany.
  - `try`...`finally` przechwytuje wszystkie „językowe'' wyjścia ze strzeżonego bloku! Czyli `raise`, `exit`, `break`, `continue` – ale nie `Halt`.
- Blok `finally`...`end` kończy się tak, jakby skończył się blok strzeżony gdyby nie było `try`...`finally`: normalnie lub wyjątkiem – chyba że sam rzuca wyjątek.
  - Jeśli do `finally` trafiliśmy przez `exit`, ale w `finally`...`end` podmienimy wartość `result`, to funkcja zwróci podmienioną wartość.

Kontrola pamięci: moduły `HeapTrc` i `lineinfo`

- Moduł `HeapTrc` śledzi obsługę pamięci (wywołania `GetMem`/`FreeMem`, operatory `new`/`dispose`...)
- Włączając go do programu – podając w `uses` albo przekazując `-gh` kompilatorowi – dostaniemy „memory dump” informujący m.in. o niezwróconej pamięci.
- Jeśli dodatkowo użyjemy modułu `lineinfo` – najlepiej przekazując `-gl` kompilatorowi – `HeapTrc` wypisze pozycje w źródłach gdzie widzi problem.
- Uruchamiam kompilator z `fpc -dDEBUG` bo w `.fpc.cfg` mam:

```
#IFDEF DEBUG
 -gl -gh -Crtoi -Sa
#ENDIF
```

Moduł `SysUtils`

Moduł `SysUtils` definiuje / inicjalizuje:

- funkcje konwersji pomiędzy liczbami a stringami (również uwzględniając formatowanie);
- manipulacja datą i czasem;
- działania na nazwach plików, katalogach i (trochę) plikach;
- procesy: `ExecuteProcess`, system: `GetEnvironmentVariable`;
- klasy wyjątków, między innymi:
  - bazowa `Exception`,
  - `EConvertError` – konwersje, głównie ze i na string,
  - `EDivByZero`, `EOverflow`, `EIntOverflow` (błędy operacji liczbowych)
  - `EInvalidPointer`, `EOutOfMemory`, etc.
  - `ERangeError` błąd typu zakresowego lub indeks poza tablicą;
- `SysUtils` inicjalizuje wyświetlanie nieprzechwyconych wyjątków
- oraz pełnych nazw błędów runtime (zamiast samych kodów błędów).

Wyjątki: przykłady

Trochę schematycznych przykładów.

```
program TestTextFile;                          Pamiętaj o przekazaniu -Ci kompilatorowi.
uses SysUtils;                                                     Zawsze potrzebne!
var
   F : TextFile;         Pliki tekstowe współpracują z WriteLn i ReadLn (wygodne).
begin
   AssignFile (F, 'TestTextFile.txt');                                  Ustaw plik.
   Rewrite (F);                                                   Otwórz do zapisu.
   try                                (Rewrite samo zamknie plik jeśli będzie problem.)
      WriteLn (F, '2+3=', 2+3, '; 2/3=', 2/3 :0:3);
   finally
      CloseFile (F)                  Zamknij (inaczej możemy stracić zmiany w pliku).
   end
end.
```

```
program Conversion;
uses SysUtils;
begin
  try
    if ParamCount < 2 then Abort;
    WriteLn (ParamStr (1), ' * ', ParamStr (2), ' = ',
             StrToFloat (ParamStr (1))
             * StrToFloat (ParamStr (2)) :0:3)
  except
    WriteLn ('Bad commandline arguments.');
    WriteLn ('Syntax: Conversion [number] [number]')
  end
end.
```

Wygodne funkcje konwersji ze stringów na liczby: `StrToFloat` i `StrToInt`.

Rzuca wyjątek `EAbort`.

Chwytamy wszystkie wyjątki.

```
program TestFinally;
uses SysUtils; const test = 0;
type
  EMyExc = class (Exception);
  MyStrangeExc = class (TObject);
begin
  repeat
    WriteLn ('in repeat');
    try
      WriteLn ('in try');
      case test of
        0: break;
        1: continue;
        2: exit;
        3: Halt;
        4: raise Exception.Create('test 3');
        5: raise EMyExc.Create('test 3');
        6: raise MyStrangeExc.Create;
        7: ;
      end; WriteLn ('after control transfer')
    finally
      WriteLn ('finally section')
    end;
    WriteLn ('finishing loop')
  until True;
  WriteLn ('after until')
end.
```

Deklarujemy nowy wyjątek.
To nie jest poprawny wyjątek...

Bez względu jak spróbujemy opuścić pętlę

(no, prawie)

trafimy do sekcji końcowej.

```
{From http://wiki.freepascal.org/File_Handling_In_Pascal }
program FileTest;
{$I+} {Same as -Ci parameter to fpc.}
uses SysUtils;
var FileVar: TextFile;
begin
   WriteLn('File Test');
   AssignFile(FileVar, 'Test.txt');
   try
      Rewrite(FileVar);  // creating the file
      Writeln(FileVar,'Hello');
      CloseFile(FileVar);
   except
      on E: EInOutError do
      begin
         WriteLn('File handling error occurred. Details: ',
                 E.ClassName, '/', E.Message);
      end;
   end;
   WriteLn('Program finished. Press enter to stop.');
   ReadLn;
end.
```

Parametr `-Ci` włącza rzucanie wyjątków przez funkcje na plikach „starego typu”.

W końcu! Chwytamy obiekt wyjątku i wydobywamy jego dane.

```pascal
program StringListTest;
uses Classes;
begin
  with TStringList.Create do
    try
      Add('LINE1');
      Add('LINE2');
      SaveToFile('filename')
    finally
      Free
    end
end.
```

Jedno z „nowoczesnych” podejść do plików:

klasa „lista stringów”

(dodaj parę elementów)

dysponuje metodą „zapisz liniami do pliku”.

```
program TestExFunc;
uses SysUtils;
type
  EMyExc = class (Exception);

  function Test : Integer;
  begin
    WriteLn ('starting Test');
    try
      exit (1);
      //raise EMyExc.Create ('test exc')
    finally
      WriteLn ('in exc handler');
      Test := 2;
    end;
    WriteLn ('before end');
    Test := 3;
  end;

begin
  WriteLn (Test)
end.
```

Zwracamy „nielokalnie” wartość z funkcji.

Przechwytujemy wyjście z funkcji

i podmieniamy zwracaną wartość.

Tej wartości nie zwróci, bo `finally` kontynuuje wychodzenie z funkcji (podobnie do kontynuowania rzucania wyjątku).

```
{Overlay an image over a background; handle errors.}
program ImageSDLExn;
uses SysUtils, SDL, SDL_image;
const
   BackgroundFile = 'background.bmp';
   BallFile = 'ball.bmp';
   pixX = 200; pixY = 200;
var
   screen : PSDL_Surface = nil;
   background : PSDL_Surface = nil;
   ball : PSDL_Surface = nil;
   SrcRect, DstRect : SDL_Rect;
begin
   try try
      if SDL_Init(SDL_INIT_VIDEO) < 0 then
         raise Exception.Create ('Couldn''t initialize SDL');
      background := IMG_Load (BackgroundFile);
      if background = nil then
         raise Exception.Create ('Couldn''t load ' + BackgroundFile);
      screen := SDL_SetVideoMode(background^.w, background^.h,
                                 background^.format^.BitsPerPixel,
                                 SDL_SWSURFACE);
      if screen = nil then
      raise Exception.Create ('Couldn''t set video mode');
      ball := IMG_Load (BallFile);
      if ball = nil then
         raise Exception.Create ('Couldn''t load ' + BallFile);
```

```
    if SDL_BlitSurface(background, nil, screen, nil) < 0 then
      raise Exception.Create ('BlitSurface background error');
    with SrcRect do begin
      x := 0; y := 0; w := ball^.w; h := ball^.h
    end;
    with DstRect do begin
      x := pixX; y := pixY; w := pixX + ball^.w; h := pixY + ball^.h
    end;
    if SDL_BlitSurface (ball, @SrcRect, screen, @DstRect) < 0 then
      raise Exception.Create ('BlitSurface ball error');
    SDL_Flip (screen);
    ReadLn;
  except
    on E: Exception do
      WriteLn ('ERROR: ', E.Message, ' -- ', SDL_GetError);
  end
  finally
    if background <> nil then SDL_FreeSurface(background);
    if ball <> nil then SDL_FreeSurface(ball);
    if screen <> nil then SDL_FreeSurface(screen);
    SDL_Quit
  end
end.
```

Przydzielanie i zwalnianie obiektów `class`

- Używaj typów `object` (lub typów *extended* `record`) tylko gdy możesz się obejść bez mechanizmów OO: konstruktorów, destruktorów i metod wirtualnych. Typy „extended `record`" dodatkowo nie mają dziedziczenia.
  - Ale potrzebujesz metod, np. ze względu na typy generyczne.
  - Nie tylko problemy z `object`, ale też zalety typów `class` charakterystyczne dla Delphi (/ Free) Pascala.
- Konstruktory obiektów `class` wywołujemy zwykle jako *metody klasy*, tzn. jako pole klasy: np. `TClass.Create`. Wtedy zostanie przydzielona pamięć przez `GetMem`, i na przydzielonym obiekcie wywołany konstruktor.
  - Możemy też wywołać konstruktor jako zwykłą metodę na istniejącym obiekcie.
- Wywołanie destruktora na obiekcie powoduje, po wykonaniu go jako zwykłej metody, zwolnienie pamięci obiektu przez `FreeMem`.
- Konstruktory zwykle nazywamy `Create` lub `Create`X, a destruktory `Destroy`.

- W metodach mamy dostęp do referencji na aktualny obiekt przez słowo kluczowe `self` (odpowiednik `this` z C++ i Javy), ale też do pól i metod danej klasy bez dodatkowej kwalifikacji
(tak jakby metody były w klauzuli `with self`).

- Są powody żeby destruktor zawsze nazywać `Destroy`:

np. klasa `TObject` (bazowa dla wszystkich obiektów) definiuje:

```
procedure TObject.Free;
  begin
     // the call via self avoids a warning
     if self<>nil then
       self.destroy;
  end;
```

a w `sysutils` mamy:

```
procedure FreeAndNil(var obj);
  var temp: tobject;
  begin
    temp:=tobject(obj);
    pointer(obj):=nil;
    temp.free;
  end;
```

- Jeśli konstruktor zostanie przerwany, np. wyjątkiem, to zwróci `nil`.
  - Dlaczego następujący zwięzły kod jest poprawny?

```
Foo1 := nil;
Foo2 := nil;
Foo3 := nil;
try
  Foo1 := TFoo.Create;
  Foo2 := TFoo.Create;
  Foo3 := TFoo.Create;

  ... operacje na Foo1,2,3 ...
finally
  FreeAndNil(Foo3);
  FreeAndNil(Foo2);
  FreeAndNil(Foo1);
end;
```

- Warto wspomnieć, że procedury `Reset` i `Rewrite` otwierające plik, podobnie same zajmują się jego zamknięciem w razie niepowodzenia, dlatego pozostawiamy je poza klauzulą `try`...`finally`.

Moduły

- Moduły pozwalają m.in. na korzystanie z funkcji bibliotecznych i osobną kompilację fragmentów programu.
- Implementacji modułów które podajemy w klauzuli `uses` kompilator Free Pascala szuka w plikach z rozszerzeniem `.ppu` w trzech wersjach wielkości liter: np. `uses SysUtils` szuka w `sysutils.ppu`, użytej formie `SysUtils.ppu`, oraz `SYSUTILS.ppu`.
- Kompilator Free Pascala rekompiluje potrzebne moduły jeśli ma dostęp do kodu źródłowego który jest młodszy od odpowiednich plików `.ppu`.
- Nazwy zadeklarowane przez użyty moduł są dostępne bezpośrednio, chyba że nazwa jest przesłonięta przez taką samą z innego modułu.
- Zawsze można użyć nazwy kwalifikowanej: `modul.nazwa`.
- Moduły są „ładowane” w kolejności w jakiej występują w `uses`, i nazwy później załadowanych przysłaniają nazwy wcześniej załadowanych.
- Domyślnie użytym modułem jest `System`, oraz we Free Pascalu `objpas` (w trybach `objfpc` – który zakładamy – oraz `delphi`).

- Moduły dostępne z Free Pascalem i Lazarusem można poklasyfikować na:

  1. Należące do „Run Time Library" RTL (patrz *Run Time Library: reference guide*): wspierane na wszystkich platformach i zazwyczaj potrzebne (również obsługa klawiatury i myszy), oraz istniejące dla kompatybilności z Turbo Pascalem i Delphi.

  2. Należące do „Free Component Library" (patrz *Free Component Library: reference guide*) oraz do „Lazarus Component Library".

  3. Pakiety włączone z zewnątrz, np. moduły powiązane z biblioteką SDL włączone z projektu *JEDI-SDL*.

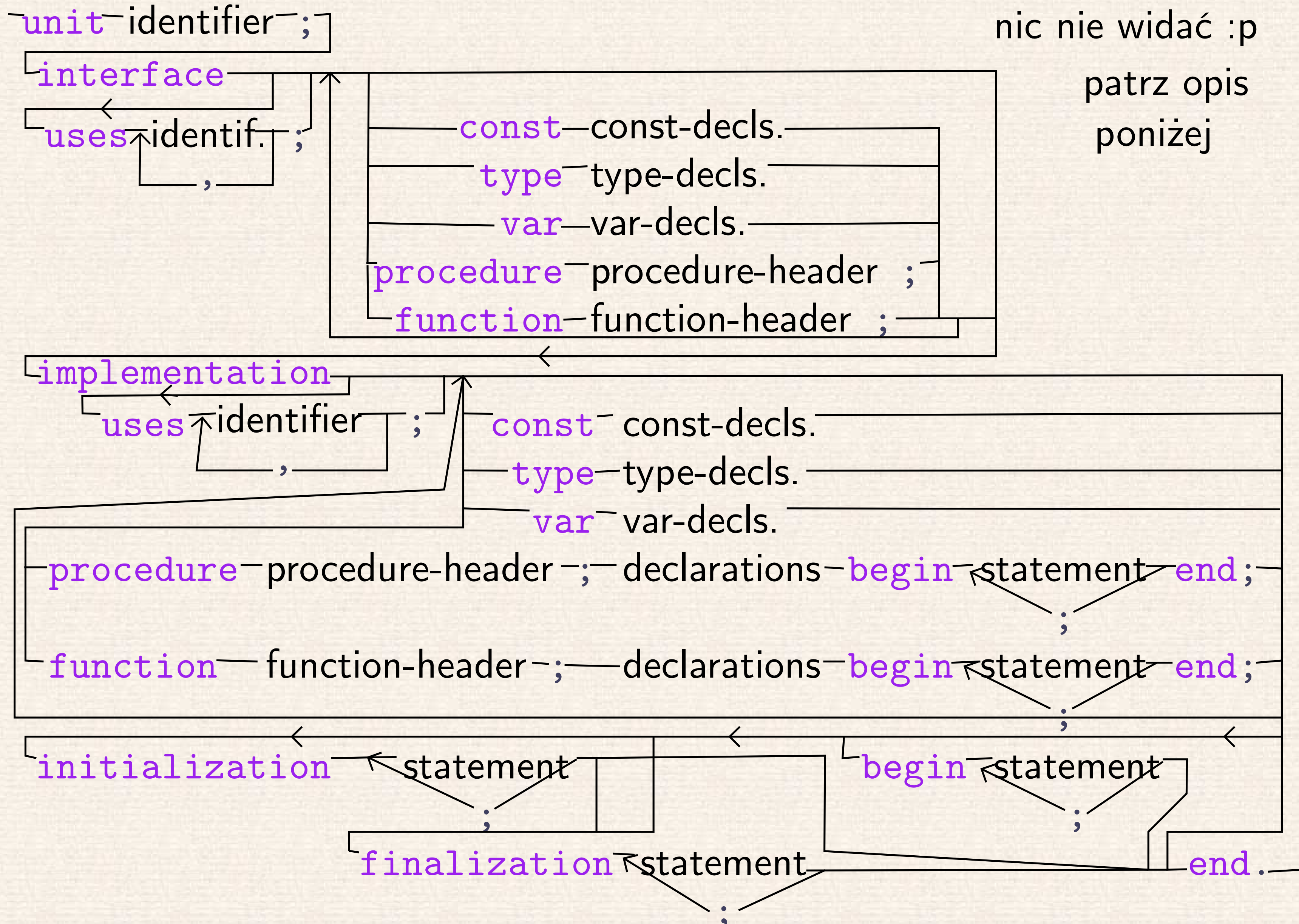
  Część modułów (te w „reference guide") jest dobrze udokumentowana, niestety niektóre nie posiadają dokumentacji wcale.

  - Czasami komentarze w kodzie źródłowym są wystarczającą dokumentacją, np. w modułach interfejsu do SDL, więc warto sprawdzić.

Struktura modułu

nic nie widać :p

patrz opis poniżej



- Moduł musi składać się z części:
  - `unit` identyfikator-modułu`;`
  - `interface` – deklaracje jak w programie, ale procedury i funkcje tylko z sygnaturami;
  - `implementation` – deklaracje-definicje jak w programie, procedury z sygnaturami w interfejsie muszą być zdefiniowane;
  - `end.`
- Moduł może mieć inicjalizację i (niezależnie) finalizację.
- Inicjalizacja może być wprowadzona przez `initialization` lub `begin`.
- Finalizacja jest wprowadzona przez `finalization` – wtedy inicjalizacja nie może być przez `begin`.
- Inicjalizacje następują w kolejności zgodnej z kolejnością w klauzuli `uses`, chyba że moduł był użyty przez inny moduł wcześniej zainicjalizowany.
- Finalizacje są w kolejności odwrotnej do inicjalizacji.

- Tylko deklaracje z `interface` modułu będą widoczne na zewnątrz, pozostałe są lokalne.
  - Moduł może wprowadzić zmienne globalne w `interface` (co nie znaczy że warto).
- Deklaracje z modułów użytych przez dany moduł nie będą automatycznie dostępne po użyciu tego modułu, nawet jeśli były w klauzuli `uses` interfejsu danego modułu.
  - Czyli jeśli mamy:
    `unit A; interface uses B; ... implementation ... end.`
    to po `uses A;` możemy mieć komunikaty o błędach z typami z `B`, ale sami nie możemy używać typów z `B`.
- Deklaracje z `interface` mogą korzystać tylko z typów wprowadzonych w `uses` interfejsu – dlatego osobne `uses` w implementacji ma sens, te typy i procedury będą do użycia tylko lokalnie.
- Cykl w zależnościach pomiędzy modułami jest dopuszczalny jeśli przechodzi przez `uses` w implementacji.

- Zmienne globalne przeszkadzają wykorzystaniu modułu przez niezależne fragmenty programu i wielowątkowości (ale są wygodne w prostych przypadkach).

```
unit ImageManager;
```

Będziemy ładować obrazki i zwalniać je przy finalizacji modułu.

```
interface
uses SysUtils, SDL, SDL_image;

var RaiseOnError : Boolean;
```

Zmienna globalna: flaga rzucania wyjątku przy niepowodzeniu operacji.

```
function LoadImage (name : String) : PSDL_surface;
```

Sygnatura-nagłówek funkcji.

```
type EImageManager = class (Exception);
```

Deklaracja klasy. (Nie dodajemy metod więc nie będzie wzmianki w implementacji.)

```
implementation

type
```

Lista wiązana (w przyszłości zastąpimy je kontenerami Free Pascala) – typ lokalny dla modułu.

```
  PNode = ^TNode;
  TNode = record
    elem : PSDL_surface;
    tail : PNode
  end;
```

```
var
  First, Last : PNode;
```

Zmienne modułowe trzymające „bufor” obrazków.

```
function LoadImage (name : String) : PSDL_surface;
begin
  try
    Last^.elem := IMG_Load (PChar(name));
    LoadImage := Last^.elem;
    new (Last^.tail);
    Last := Last^.tail;
    Last^.elem := nil; Last^.tail := nil
  except
    if Last^.elem <> nil then
      SDL_FreeSurface (First^.elem);
    Last^.elem := nil;
    LoadImage := nil;
    if RaiseOnError then
      raise EImageManager.Create ('Couldn''t load ' + name +
                                  ' -- ' + SDL_GetError);
  end
end;
```

Implementacja głównej funkcji.

```
initialization
   RaiseOnError := True;
   new (first);
   First^.elem := nil;
   First^.tail := nil;
   Last := First
```

Inicjalizacja zmiennych „globalnych i modułowych”.

```
finalization
   {Uses Last as temporary variable.}
   repeat
      if First^.elem <> nil then SDL_FreeSurface (First^.elem);
      Last := First;
      First := First^.tail;
      dispose (Last)
   until First = nil
end.
```

Zwalnianie zasobów.

Przeciążanie operatorów

- Operatory to funkcje o specjalnej składni upraszczającej ich użycie w wyrażeniach (z wyjątkiem operatorów przypisania). We Free Pascalu mamy operatory:
  - Przypisania: `:=`
  - Bitowe: `shl shr << >>`
  - Arytmetyczne: `div mod + - * / ** ><`
  - Porównania: `= < <= > >= <> in`
  - Logiczne: `or and not xor`
  - Operatory związane z typami: `as is`
  - Przypisania w stylu C: `+= -= *= /=`
    – po włączeniu, np. `{$COPERATORS ON}`

  Operatory przypisania są składniowo instrukcjami.

- Wszystkie operatory w Pascalu łączą w lewo. Priorytety operatorów:
  - najwyższy: operatorny unarne (`not`, `@`),
  - operatory „mnożenia": `shl shr << >> div mod * / ** and as`
  - operatory „dodawania": `+ - or xor`
  - operatory porównania: `= < <= > >= <> in is`

- Procedury, funkcje i operatory we Free Pascalu można przeciążać: mieć kilka procedur o tej samej nazwie ale różnych typach argumentów.
- Przeciążanie operatora jest jak definiowanie funkcji, zamiast `function` używamy `operator`.
- Operatory przypisania definiujemy jako funkcje: biorące wartość prawej strony i zwracające wartość do zapisania w przypisywanej zmiennej.
- Nie możemy stosować nazwy operatora jako zmiennej, więc zapisujemy wartość funkcji używając słowa kluczowego `result`:

```
operator := (r : Real) : Complex;
begin
   result.re:=r; result.im:=0.0;
end;
```

- Inna metoda to nadać identyfikator definicji operatora i zapisywać do niego:

```
operator := (r : Real) z : Complex;
begin
   z.re:=r; z.im:=0.0;
end;
```

- Przeciążać można operatory: przypisania, arytmetyczne i porównania.
  - Operatorów przypisania w stylu C nie można przeciążać.
- Przeciążanie „różności” `<>` i „należenia do” `in` jest dopiero we Free Pascalu 2.6.
  - Po przeciążeniu równości `=` automatycznie dostaje się różność `<>`, ale `<>` można też przeciążyć niezależnie.

```
program OperatorTest;
uses HeapTrc, SysUtils;
type
   TStrList = record
      elem : Integer;
      tail : ^TStrList
  end;
   PStrList = ^TStrList;

   operator ** (e : Integer; list : PStrList) : PStrList;
   begin
      new (result);
      result^.elem := e;
      result^.tail := list
   end;

var list, node : PStrList;
begin
   list := nil;
   list := 1 ** (3 ** (5 ** (7 ** (9 ** list))));
   repeat
      Write (list^.elem, ', ');
      node := list^.tail; dispose (list); list := node
   until list = nil;
   WriteLn
end.
```

Tutaj „forwarding pointers” nie są konieczne począwszy od Free Pascala 2.6, ale zadeklarowane typy wskaźnikowe ciągle są konieczne, np. w nagłówkach funkcji.

Wszystkie operatory wiążą w lewo.

Rzutowanie typów

- Określeniem **rzutowanie na typ** oznacza się dwie różne czynności:
  - wspartą językowo konwersję z jednego typu na inny,
  - poinformowanie systemu typów że widzimy dany obiekt jako należący do innego typu niż mu się wydaje. (Bezpieczny wariant: rzutowanie do klasy bazowej, niebezpieczny: rzutowanie do klasy pochodnej.)
- Operatory przypisania definiują ten pierwszy rodzaj rzutowania: **domyślne konwersje** typów.
- Rzutowania (konwersje) są stosowane domyślnie przy rozstrzyganiu odniesienia do procedury / funkcji / operatora.
- Kompilator rozstrzyga odniesienia funkcji przeglądając kontekst wstecz od punktu wywołania: najpierw deklaracje w obecnym module od najbliższej wywołaniu wstecz, potem deklaracje w użytych modułach począwszy od ostatniego w klauzuli `uses`.
  - Przeciążając funkcję działającą na starym typie i jednocześnie deklarując konwersję ze starego do nowego typu zasłonimy starą funkcję!

- Rzutowania (konwersje) możemy też wywołać ręcznie. Składnia działająca dla obu rodzajów rzutowań to:
  `TypDocelowy(wyrazenie)`

- Składnia `wyrazenie as TypDocelowy` działa tylko dla rzutowań drugiego rodzaju, tzn. pomiędzy typami `class`.

- Patrz przykłady `OverloadingAndConversions.pas` i `TypeCast.pas` w paczce z programami do wykładu.

Przeciążanie a typy generyczne

- Typy generyczne we Free Pascalu są zrealizowane jak szablony / wzorce w C++ (nie jak typy parametryczne w Javie czy C#), czyli jako makro generujące kod w miejscu specjalizacji.
- Pomimo to, rozstrzyganie odniesień nazw w czasie specjalizacji szablonu jest na bazie kontekstu nazw z punktu definicji szablonu a nie z punktu specjalizacji!
- Czyli nie możemy liczyć na to, że procedury i operatory przeciążone dostosują się do typu dla którego specjalizujemy.
- Oznacza to, że potrzebne, zależne od specjalizacji operacje musimy zapakować w typach względem których parametryzujemy.