# 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 1: Przegląd języka

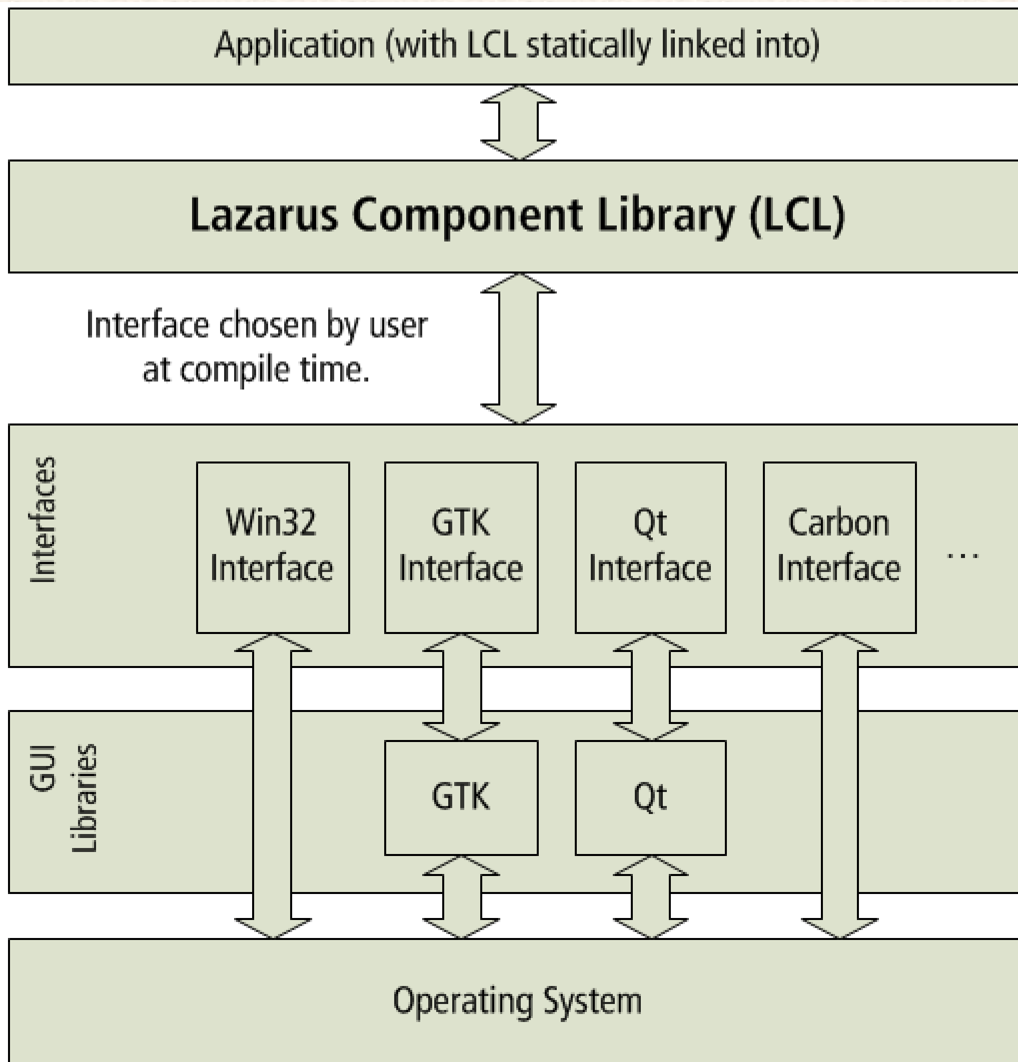
## The Good, The Bad, & The Ugly.

# Trochę historii

- Niklaus Wirth zaczął tworzyć **Pascala** w 1968, opublikował go w 1970, pierwszy kompilator z 1972.
- Wywodzi się z **ALGOLa** (Wirth pracował wcześniej nad ALGOL W).
- Mały, wydajny („systems programming” jak C), edukacyjny.
- Niklaus Wirth przeszedł do pisania serii języków **Modula** i **Oberon**, ale nie zyskały one popularności.
- Na początku lat 80tych Pascala zaportowano na Apple'ach jako alternatywę dla BASICa.
- W 1983, firma Borland tworzy rewelacyjną implementację **Turbo Pascal** (kompilator i IDE) dla MS-DOS i CP/M
  - Szybka, działająca bezpośrednio z RAMu, dobre IDE.
- W 1985 Larry Tesler we współpracy z Wirthem tworzy **Object Pascal**, zaimplementowany w kompilatorach na Apple Macintosh.

- Borland wprowadził moduły `unit` w TP4.0 (1987), oraz dialekt Object Pascala w TP5.5 (1989).
- Gdy po TP7.0 z 1992 roku Borland rezygnuje z wydawania na MS-DOS, Paul Klämpfl zaczyna tworzyć open-source kontynuację Turbo Pascala jako **Free Pascal**.
- W 1995 Borland wydaje **Delphi** („jeśli chcesz porozmawiać z wyrocznią [bazą danych Oracle], idź do Delf.")
- W 1999 Delphi 5 dodaje *frames* (enkapsulacja elementów graficznych) i interfejsy ze zliczaniem dowiązań (*reference counting*).
- W 2000 wychodzi Free Pascal 1.0, Free Pascal zaczyna adaptować cechy językowe wprowadzone do Delphi.
- W 2004 roku Free Pascal umie już kompilować na x86, MC680x0, PowerPC, ARM, SPARC, AMD64.
- W 2008 wychodzi IDE / biblioteka komponentów graficznych **Lazarus**, w 2010 FP 2.4 (m.in. konstrukcja `for...in`, klasy `abstract`).

# "Write once, compile everywhere"



# Program w Pascalu

```
{
  Copyright 2008-2010 Michalis Kamburelis. [...]
  Popsuty w celach demonstracji przyklad.
}
{$mode objfpc}{$H+}{$C+}

program radiance_transfer;

uses VectorMath, Boxes3D, X3DNodes, GL, GLU, CastleWindow;

type
  PViewMode = ^TViewMode;
  TViewMode = (vmNormal, vmSimpleOcclusion, vmFull);

var
  Window: TCastleWindowCustom;
  ViewMode: TViewMode = vmFull;

const
  LightSHBasisCount = 25;

var
  LightSHBasis: array [0..LightSHBasisCount - 1] of Single;
  LightIntensity: Single = 1.0;
```

Komentarz

Dyrektywy kompilatora (m.in. tryb Object Pascal)

Program główny (czyli nie moduł biblioteczny)

Użyte moduły

Definicje/deklaracje typów

Typ wskaźnikowy na typ zdefiniowany później

Typ wyliczeniowy

Zmienne

Zmienna z wartością początkową

Stałe

Deklaracje różnych rodzajów mogą się przeplatać

Tablica stałego rozmiaru

(wtedy podajemy zakres którym indeksowana)

```
procedure DrawLight(ForMap: boolean);
begin
  glPushMatrix;
    glTranslatev(LightPos);
    if not ForMap then
      glDisable(GL_BLEND);
  glPopMatrix;
end;

type
  TMySceneManager = class(TCastleSceneManager)
    procedure RenderFromViewEverything; override;
  end;

procedure TMySceneManager.RenderFromViewEverything;
begin
  glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
  glLoadMatrix(RenderingCamera.Matrix);
end;

function UpdateViewMode: Boolean; forward;

procedure Open(Glwin: TCastleWindowBase);
begin
  glEnable(GL_LIGHT0);
  if UpdateViewMode then WriteLn('OK');
end;
```

Procedura z jednym argumentem (typu `boolean`)

Pascal nie jest wrażliwy na wielkość liter (`boolean` to to samo co `Boolean`)

Wcięcia pomagają uniknąć błędów

Definicja klasy dziedziczącej z `TCastleSceneManager` (procedura dynamicznie zastąpi tą z klasy bazowej)

Implementacja metody

Operacje boolowskie nazywają się `or`, `and`, `not`

Deklaracja funkcji (w celu rekursji wzajemnej)

Wywołując funkcję bezargumentową nie musimy podawać nawiasów

```
function UpdateViewMode: Boolean;
begin UpdateViewMode := True;
end;
```

Definicja funkcji bezargumentowej
Wartość do zwrócenia przypisujemy do zachowującej się jak zmienna nazwy funkcji

```
type
   TOctreeSubnodeIndex = array[0..2]of boolean;

function SubnodeWithPoint(const P: TVector3Single): TOctreeSubnodeIndex;
begin
  result[0] := P[0] >= MiddlePoint[0];
  result[1] := P[1] >= MiddlePoint[1];
  result[2] := P[2] >= MiddlePoint[2];
end;
```

Zamiast nazwy funkcji możemy użyć słowa kluczowego `result`

Rezultat wypełniamy jak każdą inną zmienną (inny przykład: `result.IntField := 7`)

```
procedure Idle(Glwin: TCastleWindowBase);

  procedure ChangeLightPosition(Coord, Change: Integer);
  begin
    LightPos[Coord] += Change * Glwin.Fps.IdleSpeed *
      { scale by Box3DAvgSize, to get similar move on all models }
      Scene.BoundingBox.AverageSize;
    Glwin.PostRedisplay;
  end;

  procedure ChangeLightRadius(Change: Float);
  begin
    LightRadius *= Power(Change, Glwin.Fps.IdleSpeed);
    Glwin.PostRedisplay;
  end;
```

Procedura zagnieżdżona

(może używać argumentów procedury nadrzędnej)

Z parametrem `-Sc` kompilatora możemy używać przypisań w stylu C

Blok kodu procedury

```
begin
  if Glwin.Pressed[K_A] then ChangeLightPosition(0, -1);
  if Glwin.Pressed[K_R] then
  begin
    if mkShift in Glwin.Pressed.Modifiers then
      ChangeLightRadius(1/1.8) else
      ChangeLightRadius(1.8);
  end;
end;
```

Blok kodu programu głównego

```
begin
  Window := TCastleWindowCustom.Create(Application);
  Scene := TCastleScene.Create(Application);
  OnWarning := @OnWarningWrite;

  Window.OpenAndRun;

 FreeAndNil(RenderParams);
end.
```

Przypisanie do zmiennej proceduralnej

Wywołanie procedury bezargumentowej (nawiasy są niepotrzebne)

# Cechy języka Object Pascal – Free Pascal

- Klarowna struktura programów i bibliotek. Porównaj:

```
public class HelloWorld {
  public static void main (String[] args) {
    System.out.println("Hello "+args[0]+"!");
  }
}
```

```
program HelloWorld;
begin
   WriteLn ('Hello '+ParamStr(1)+'!')
end.
```

- Podział na procedury: funkcje, które nie uczestniczą w wyrażeniach, oraz funkcje: procedury, które zwracają (główny) rezultat obliczeń.
- Tryby argumentów procedur: wejściowy `const`, wyjściowy `out` oraz dwukierunkowy `var`.
  - Jest też tryb domyślny, wejściowy – przekazujący przez wartość.

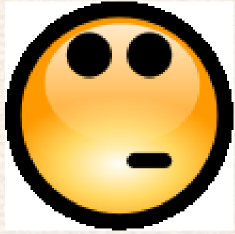
```
procedure FindParents(const Child: Float; out Mother: Parent; out Father: Parent);
```

- Przeciążanie operatorów jak w C++.
- Wbudowany typ zbiorów. (Niestety?) tylko nad typami porządkowymi, i w obecnej implementacji (na bitmapie) ograniczony do 256 elementów.
- Wbudowane typy: podzakresy typów porządkowych. Pozwalają zawężać typy danych, i typy argumentów funkcji, dla pełniejszej zgodności z zamierzoną semantyką (tzn. znaczeniem).
  - Oczywiście sprawdzane w czasie działania programu a nie kompilacji. (I to po włączeniu opcji *range check*.)
  - Idiom: `type TEType = Succ(Low(TPossEType)) .. High(TPossEType);` – gdy pierwsza wartość typu wyliczeniowego oznacza „nic”/„żaden”.
- Tablice o „statycznych” rozmiarach można indeksować podzakresami typów porządkowych, gdy jest powód żeby nie numerować liczbami od 0.
  - Najczęściej typem wyliczeniowym, lub liczbami od 1.

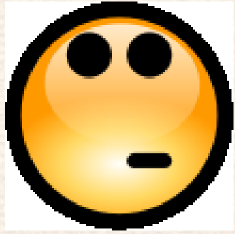
- Instrukcja `for`...`in` jak w Pythonie i Javie.
- Szablony *templates* jak w C++.
- Bardzo „Pascalowa” konstrukcja: *properties*.

```
type MyClass = class
private
   Field1 : Longint;
   Field2 : Longint;
   Field3 : Longint;
   procedure Sety (value : Longint);
   function GetY : Longint;
   function GetZ : Longint;
public
   property X : Longint read Field1 write Field2;
   property Y : Longint read GetY write SetY;
   property Z : Longint read GetZ;
end;
```

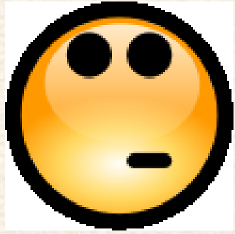
- Zagnieżdżone funkcje, z leksykalnymi zakresami zmiennych.
  - Funkcja zagnieżdżona korzysta z argumentów funkcji zewnętrznej.
- Brak wielokrotnego dziedziczenia, ale klasy implementują interfejsy jak w Javie.
- Klasy ze zliczaniem dowiązań (*reference counted*).
  - W szczególności `AnsiString`.
- W najnowszej wersji, typy zagnieżdżone, jak w Javie.



- Wbudowane typy plikowe `file of Typ` okazały się pojęciem zbyt mało abstrakcyjnym jak na konstrukcję językową, nie dają łatwej kontroli nad rodzajem przeprowadzanego I/O. W praktyce zastępowane przez klasy pochodzące z `TStream`.
- Brak możliwości przeplatania bloku z kodem i definicji/deklaracji typów, zmiennych i procedur. Chyba poprawia to klarowność, o ile pamięta się żeby bloki z kodem były małe.
- Brak możliwości inicjalizowania zmiennych obliczanymi wartościami. Jest to zgodne z separacją deklaracji od bloku z kodem, ale psuje klarowność funkcji zagnieżdżonych.
- Szablony *templates* jak w C++.
- Programista musi decydować o zarządzaniu pamięcią: czy stworzyć rekord/obiekt na stosie, czy na stercie, czy używać *reference counting* czy samemu zwalniać (i wtedy pamiętać np. o `try`...`finally`).



- Podział modułów na interfejs i implementację „niemal” pozwala definiować abstrakcyjne typy danych w lekki sposób na styl SMLa i OCamla.
  - Używane moduły są zawsze otwarte, dostęp kwalifikowany `modul.Funkcja` jest opcjonalny (chyba że nazwa jest zasłonięta).
  - Nie można zadeklarować w interfejsie typów abstrakcyjnych (nawet wskaźnikowych), dlatego abstrakcyjne typy danych to muszą być obiekty / klasy z polami prywatnymi.
- Brak domknięć funkcyjnych *closures*. W nowym Free Pascalu można przypisywać funkcje zagnieżdżone do zmiennych, ale ich zwrócenie z funkcji nadrzędnej jest **niebezpieczne**.
  - Ale można ręcznie budować *closures* jako wskaźniki na metody.
  - (Nawiasem, Java wbrew pozorom ma *closures*: obiekt klasy lokalnej, np. anonimowej, jeśli korzysta tylko z `final` funkcji zewnętrznej.)



- Szok kulturowy dla mnie: niekonsystentność autorów dokumentacji Free Pascala co do wielkości liter w słowach kluczowych i identyfikatorach.
    - Ale w praktyce stosuje się głównie: małe litery dla słów kluczowych, i CamelCase dla identyfikatorów (z wyjątkiem: małe litery dla zmiennych pomocniczych).

- Brak możliwości nadawania nazw wartościom, jak robią zmienne `final` w Javie. Powinien być blok definicji `val` (słowo kluczowe m.in. w Scali), podobny do `var`, ale z wyrażeniem do obliczenia za `=`. Tak zdefiniowana nazwa na wartość nie byłaby przypisywalna.
  - Na pocieszenie, są argumenty `const` funkcji
  - oraz twórca klasy może łatwo zapewnić pola niemodyfikowalne jeśli klasa sama ich nie modyfikuje (pola *read-only*) dzięki *properties*.
  - No cóż, Pascal jest językiem imperatywnym, a nie „zorientowanym na wyrażenia".
  - `<rant>` Idealnie, razem z `val` trzebaby dodać *closures* do języka, zwracalne funkcje wewnętrzne które nie używają `var` funkcji zewnętrznej. Od strony implementacji to byłyby automatycznie budowane obiekty pamiętające co trzeba, i zwracany byłby wskaźnik na metodę takiego obiektu. `</rant>`