Przykład aplikacji bez obsługi zdarzeń

Rozmieszczenie na formatce 11 etykiet i komponentu Image.

Rozbudowa pliku głównego aplikacji do postaci:

```
program Project1;

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

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  with Form1 do
    begin
    Label7.Caption := ExtractFileName(Application.ExeName);
    Label8.Caption := ExtractFilePath(Application.ExeName);
    Image1.Height := Application.Icon.Height;
    Image1.Width := Application.Icon.Width;
    Image1.Canvas.Draw(0, 0, Application.Icon);
    Label9.Caption := IntToStr(Screen.Height);
    Label10.Caption := IntToStr(Screen.Width);
    Label11.Caption := IntToStr(Screen.PixelsPerInch);
    end;
  Application.Run;
end.
```

UWAGA: Application i Screen to dwa obiekty predefiniowane dostępne automatycznie w każdej aplikacji w Delphi.

Programowanie modułowe

Moduł jest konstrukcją języka Pascal umożliwiającą grupowanie deklaracji typów, stałych, zmiennych i procedur w oddzielnych plikach. Koncepcja modułu pozwala na podział tekstu całego programu na części przechowywane w oddzielnych plikach. Wyodrębnione części programu mogą być osobno opracowywane i kompilowane.

Struktura modułu w Pascalu

Struktura modułu jest podobna do struktury programu. Moduł składa się z nagłówka, części deklaracyjnej i części wykonawczej. W odróżnieniu od programu część deklaracyjna modułu dzieli się na część publiczną zawierającą deklaracje elementów, które powinny być znane **poza modułem** i część prywatną zawierającą deklaracje elementów **lokalnych**. Część publiczna rozpoczyna się słowem **interface** i poprzedza część publiczną. Część prywatna rozpoczyna się słowem **implementation**.

Budowa modułu:

unit <nazwa>;
interface
<deklaracje - część publiczna>
implementation
<deklaracje - część prywatna>
begin
<część wykonawcza - inicjująca>
end.

Deklaracje procedur występujące w części publicznej mają postać zapowiedzi i powinny zawierać tylko nagłówki.
Część prywatna powinna zawierać deklaracje procedur lokalnych oraz kompletne deklaracje procedur publicznych, których nagłówki umieszczono w części interfejsowej.
Część wykonawcza modułu jest przeznaczona na umieszczenie w niej instrukcji inicjujących moduł i często pozostawiana jest pusta. Jeżeli część wykonawcza nie zawiera żadnych instrukcji, a więc składa się jedynie ze słów **begin** i **end** , to słowo kluczowe **begin** może zostać pominięte.

Dyrektywa Uses

Deklaracje zawarte w części publicznej modułu stają się widoczne w programie w wyniku umieszczenia w nim dyrektywy **uses** o postaci:

uses <lista nazw modułów>

Dyrektywa ta musi występować w części deklaracyjnej programu jako pierwsza.
W czasie tłumaczenia programu kompilator dołącza wszystkie obiekty modułów z listy dyrektywy **uses**.
Uruchomienie skompilowanego programu powoduje kolejne wykonanie części inicjujących wszystkich modułów z listy, a następnie części wykonawczej programu głównego.

Biblioteki

Moduły stosuje się do gromadzenia w jedna całość pewnych zestawów procedur (wraz z towarzyszącymi strukturami danych). Tak zgrupowane zestawy procedur nazywane są bibliotekami.

Przykład

unit Ciagi; { moduł – biblioteka przetwarzania ciągów – w pliku CIAGI.PAS}
interface

const max_wym = 10;
type Tab = **array** [1..max_wym] **of** real;

procedure czytaj(n: integer; **var** x: tab);
function Suma(n: integer; x: tab): real;

```
function Srednia (n: integer; x: tab): real;

implementation

function Suma(n: integer; x: tab): real;
var i: integer;
    s: real;
begin
s:=0;
for i:=1 to n do s:=s+x[i];
suma:=s;
end;

function Srednia (n: integer; x: tab): real;
begin
if n>0 then srednia:=suma(n, x)/n else srednia:=0;
end;

procedure czytaj(n: integer; var x: tab);
var i:integer;
begin
writeln('Podaj elementy ciagu');
for i:=1 to n do read(x[i]);
end;

end.

program prog1;
{ Program wykorzystujący bibliotekę Ciagi pamiętany w pliku PROG1.DPR }
uses Ciagi;

var x: tab;
     n: integer;
    m1, m2: real;
begin
Writeln('Podaj liczność ciągu');
Readln(n);
Czytaj(n, x);
Writeln('Średnia=', srednia(n, x):10:2);
end.
```

Zapis modułu powinien być umieszczony w pliku o nazwie tożsamej z nazwą modułu wymienioną w nagłówku i rozszerzeniu PAS. W przypadku modułu **Ciagi** zawartego w przykładzie nazwa pliku tekstowego zawierającego moduł powinna być CIAGI.PAS.
Jeżeli nazwa modułu jest dłuższa niż 8 znaków to nazwa pliku powinna być identyczna z pierwszymi 8 znakami.
Istnieje możliwość nadania dowolnej nazwy pliku zawierającego moduł poprzez wykorzystanie dyrektywy kompilatora {$U nazwa}. Dyrektywa ta zawierająca nazwę pliku powinna być umieszczona w nagłówku modułu bezpośrednio za słowem **unit**, a przed nazwą modułu.

Kompilacja modułów

Translacja programu składającego się z pliku programu głównego i plików modułu odbywa się w dwóch fazach.

Dla programu z przytoczonego powyżej przykładu pierwszą fazę można schematycznie przedstawić następująco:

CIAGI.PAS ⇒ CIAGI.DCU

W fazie tej odbywa się kompilacja modułu i utworzenie pliku binarnego zawierającego skompilowany moduł
W drugiej fazie odbywa się kompilacja programu głównego, konsolidacja programu i utworzenie binarnego pliku wykonywalnego.

CIAGI.DCU + PROG1.DPR ⇒ PROG1.EXE

Obie fazy mogą być oddzielone w czasie. Ponieważ do utworzenia programu wykonywalnego niepotrzebny jest plik źródłowy modułu, to utworzone wcześniej biblioteki mogą być rozpowszechniane w postaci skompilowanych plików *.TPU.

Typ rekordowy

Rekordem nazywa się strukturę, której składowe, zwane polami, mogą być rożnych typów. Definicja typu rekordowego specyfikuje typ każdego pola oraz identyfikatory tych pol.

Składnia deklaracji rekordu

```
record
<lista_nazw_pol1> : <typ1>
<lista_nazw_pol2> : <typ2>;
 . . .
end;
```

Przykład deklaracji (bezpośrednia deklaracja zmiennej rekordowej):

```
var  s1: record
         nazwisko, imie: string;
         wiek: integer;
         stypendium: real;
         end;
```

lub (deklaracja typu wykorzystana do deklaracji zmiennych)

```
type student=  record
               nazwisko, imie: string;
               wiek: integer;
               stypendium: real;
```

```
              end;
var s1, s2: student;
```

Poszczególne pola mogą być same obiektami złożonymi: tablicami, rekordami itp.
Liczba i struktura pól rekordu jest ustalona w deklaracji i nie może być potem zmieniana.

Bardziej złożone przykłady:

```
type data= record
           dzien: 1..31;
           miesiac: 1..12;
           rok: integer;
           end;
    punkt= record
           x, y: real
           end;
    okrag= record
           promien: real;
           srodek: punkt
           end;
  student= record
           nazwisko: string [20];
           imie: string [15];
           plec: boolean;
           ocena: real;
           dataur: data
           end;
```

Do poszczególnych pól odwołujemy się za pomocą deskryptorów pól zapisywanych z użyciem notacji kropkowej:

```
<nazwa rekordu> . <nazwa pola>
```

Deskryptor pola (desygnator pola) wskazuje pole o podanym identyfikatorze w danej zmiennej rekordowej. Zmienna ta może być zmienną całościową lub też może być elementem innej struktury złożonej (tablicy lub rekordu).

Zakładając definicje typów jak wyżej oraz deklaracje zmiennych:

```
   var  urodziny: data;
            P, Q: punkt
              OK: okrag;
        Kowalski: student;
           grupa: array [1..30] of student;
```

możemy, przykładowo, użyć następujących deskryptorów:

```
         urodziny.dzien
         P.y
         ok.srodek.x
```

```
grupa[12].ocena
Kowalski.dataur
grupa[4].dataur.rok
```

Przykład w programie:

```
type trojkat = record
               a, b, c: real;
               end;
function obwod(t: trojkat): real;
begin
obwod := t.a + t.b + t.c;
end;
var tr: trojkat;
begin
readln( tr.a, tr.b, tr.c );
writeln('Obwod=', obwod(tr));
 . . . .
```

Instrukcja with

Składnia:

with <zmienna rekordowa> **do** <instrukcja>

Z zastosowaniem instrukcji **with** wczytania boków trójkąta można zapisać następująco:

```
with tr do readln(a, b, c);
```

Przykłady programów z rekordami

1 - średnia powierzchnia prostokąta

```
type
      prostokat = record
                  x,y: real;
                  end;
var A: array [1..20] of prostokat;
  s: real;
  n, i: integer;
begin
readln(n);
for i:=1 to n do
  begin
  writeln;
  writeln ('podaj dane ',i,' prostokata:');
  with A[i] do
    begin
    write ('dlugosc     :'); readln (x);
    write ('szerokosc :'); readln (y);
    end;
  end;
s:=0;
for i:= 1 to n do
      s:=s+A[i].x*A[i].y;
writeln;
writeln ('Srednia powierzchnia=', s/n:8:2);
```

```
readln;
end.
```

2 - Nazwisko najlepszego ucznia

```
type
  uczen =  record
           imie,nazwisko: string;
           ocena: integer
           end;
var  lista: array [1..20] of uczen;
     rob: uczen;
     n, i: integer;
begin
readln(n);
for i:=1 to n do
  begin
  writeln;
  writeln ('podaj dane ',i,' ucznia:');
  with lista [i] do
     begin
     write ('imie     :'); readln (imie);
     write ('nazwisko :'); readln (nazwisko);
     write ('ocena    :'); readln (ocena);
     end;
  end;
rob:= lista [1];
for i:= 2 to n do
   if lista[i].ocena > rob.ocena then rob := lista[i];
writeln;
with rob do
   writeln ('Najlepsza ocene=', ocene, ' ma uczen: ', imie, ' ', nazwisko);
readln;
end.
```

Problem – budowa biblioteki procedur dla wyznaczania elementów trójkąta.

Prosty zapis (p1):

```
var a,b,c: real;
function pole:real;
var p:real;
begin
p:=(a+b+c)/2;
pole:=sqrt(p*(p-a)*(p-b)*(p-c));
end;
begin
writeln('Podaj dl bokow trojkata');
readln(a, b, c);
writeln('Pole=', pole);
readln;
end.
```

W zapisie powyżej zadeklarowano 3 zmienne globalne **a**, **b** i **c**. Zmienne są dostępne zarówno w funkcji jak i w programie głównym. Takie postępowanie jest nazywane wykorzystywaniem efektów ubocznych. Jeżeli deklaracja zmienneych a,b i c zostanie przesunięta poniżej funkcji, to uruchomienie programu okaże się niemozliwe.

Zwykle unika się wykorzystywania efektów ubocznych.

W rozwiązaniu zastosowano 3 zmienne globalne co blokuje możliwość stosowania ich oznaczeń (a, b i c) dla innych celów. Może to stanowić istotny problem jeśli rozwiązywane zadanie jest obszerne i obejmuje także inne zagadnienia w których wygodne byłoby zastosowanie tych oznaczeń do innych celów, np. dla nazywania boków innych figur płaskich.
Zastosowanie rekordów (p2):

```
type Ttrojkat= record
               a, b, c: real;
               end;
var t: Ttrojkat;
function pole:real;
var p:real;
begin
p:=(t.a+t.b+t.c)/2;
pole:=sqrt(p*(p-t.a)*(p-t.b)*(p-t.c));
end;

begin
writeln('Podaj dl bokow trojkata');
readln(t.a, t.b, t.c);
writeln('Pole=', pole);
readln;
end.
```

Lepiej, bo oznaczenia boków mamy w skrzynce-rekordzie, ale komplikują się nazwy, dodatkowo nazwa funkcji obliczającej pole nie może już być wykorzystana do identyfikacji innej procedury.
Jeszcze jedno rozwiązanie rekordowe (p2a):

```
type Ttrojkat= record
               a, b, c: real;
               end;
function pole(t:Ttrojkat):real;
var p:real;
begin
p:=(t.a+t.b+t.c)/2;
pole:=sqrt(p*(p-t.a)*(p-t.b)*(p-t.c));
end;

var t: Ttrojkat;
begin
writeln('Podaj dl bokow trojkata');
readln(t.a, t.b, t.c);
writeln('Pole=', pole(t));
readln;
end.
```

Rozwiązanie zgrabniejsze (funkcja bez wykorzystania efektów ubocznych) ale problem z nazwą **pole** pozostał.

Żeby umożliwić rozdzielne rozwiązywanie zagadnień na które podzielono złożony problem w sposób doskonalszy, pozwalający na zupełnie dowolne dobieranie oznaczeń (także procedur i funkcji) należy zastosować programowanie obiektowe.

Obiekty

Rozwiązanie obiektowe problemu (p3):

```
type Ttrojkat= object
               a, b, c: real;
               function pole:real;
               end;
var t: Ttrojkat;
function Ttrojkat.pole:real;
var p:real;
begin
p:=(a+b+c)/2;
pole:=sqrt(p*(p-a)*(p-b)*(p-c));
end;

begin
writeln('Podaj dl bokow trojkata');
readln(t.a, t.b, t.c);
writeln('Pole=', t.pole);
readln;
end.
```

Jak widać z powyższego przykładu obiekt jest rozwinięciem idei rekordu. Pozwala na zamknięcie w jednym pojemniku wszystkich nazw wykorzystywanych do opisu problemu: zarówno zmiennych-pól jak i nazw procedur.

Typy obiektowe w Pascalu definiowane są z wykorzystaniem słowa kluczowego **object** lub **class.**

type <identyfikator typu obiektowego> = **object** [(nazwa przodka)]
[<lista deklaracji pól obiektu>]
[< lista zapowiedzi metod obiektu>]
end;

W prostym przypadku gdy w deklaracji nie umieszczono nazwy przodka oraz nie zdefiniowano żadnej metody obiekt ma budowę i własności rekordu. Na przykład:

type TArgumenty = **object**
a, b, c: Real;
end;

Definicja typu obiektowego umieszczona powyżej różni się od definicji rekordu użyciem słowa kluczowego **object**. Pola tak zdefiniowanego obiektu można wykorzystywać jedynie w sposób rekordowy.

Obiekt hermetyzuje nie tylko nazwy pól ale także obiektowych procedur i funkcji - nazywanych metodami. Dodatkowo z powodu bezpośredniego dostępu do składowych obiektu upraszcza się zapis metod. W metodach nie potrzeba stosować notacji kropkowej w odniesieniu do składowych obiektu. Zbędne jest także przekazywanie tych składowych przez parametry metod.

W przykładzie można zaobserwować charakterystyczny sposób posługiwania się obiektami i składowymi obiektów. Podstawowe zasady można sformułować następująco:

- W deklaracji typu obiektowego w pierwszej kolejności wymieniane są pola obiektu, a w drugiej metody.
- Każda metoda deklarowana jest w typie obiektowym w postaci zapowiedzi zawierającej jedynie nagłówek metody.
- Kompletna deklaracja metody podawana jest odrębnie poza deklaracją typu obiektowego. Nazwa metody jest uzupełniana prefiksem – nazwą typu obiektowego. Lista parametrów w nagłówku nie może być zmieniona w stosunku do poprzedzającej zapowiedzi. Może być jedynie powtórzona bez zmian lub opuszczona w całości.
- Typy obiektowe mogą być deklarowane tylko globalnie, w programie lub w module lecz nie wewnątrz procedury.
- Zmienne obiektowe muszą być deklarowane przez wykorzystanie identyfikatora typu (zdefiniowanego zawsze globalnie), a nie opisu typu.
- Metoda należy do obiektu i w związku z tym posiada dostęp do pól i innych metod obiektu bezpośrednio. W treści procedur można korzystać z nazw pól bez konieczności dopisywania prefiksów lub umieszczania ich w parametrach metod - obydwa rozwiązania są nawet zabronione. Np. niedopuszczalny jest zapis:

  function Ttrojkat.pole:real;
  var p:real;
  begin
  p:=(Ttrojkat.a+ Ttrojkat.b+ Ttrojkat.c)/2;
  pole:=sqrt(p*(p- Ttrojkat.a)*(p- Ttrojkat.b)*(p- Ttrojkat.c));
  end;

- Poza definicją typu obiektowego i definicjami związanych z nim metod przy odnoszeniu się do pól lub metod obiektu należy używać nazw dwuczęściowych:

  <nazwa zmiennej obiektowej> **.** < nazwa pola lub metody>

  Podobnie jak w przypadku rekordów możliwe jest uproszczenie dostępu do składowych obiektu przy wykorzystaniu instrukcji **with**.

- Zmienne lokalne i parametry metody nie mogą mieć tych samych nazw co pola obiektu.
- W metodach można wywoływać inne metody obiektu nawet te zdefiniowane szczegółowo później (bez prefiksu, wystarczy nazwa metody).

Przykład bardziej rozbudowany (p4):

```
type Ttrojkat= object
               a, b, c: real;
               function obwod:real;
               function pole:real;
               end;

function Ttrojkat.pole:real;
var p:real;
begin
p:=obwod/2;
pole:=sqrt(p*(p-a)*(p-b)*(p-c));
end;
```

```
function Ttrojkat.obwod:real;
begin
obwod:=a+b+c;
end;

var t: Ttrojkat;

begin
writeln('Podaj dl bokow trojkata');
readln(t.a, t.b, t.c);
writeln('Pole=', t.pole);
readln;
end.
```

Poniżej przykład w którym te same nazwy zastosowano dla elementów trójkąta i prostokąta (p5).

```
type Ttrojkat= object
               a, b, c: real;
               function obwod:real;
               function pole:real;
               end;
type Tprostokat= object
                  a, b: real;
                  function obwod:real;
                  function pole:real;
                  end;

function Ttrojkat.obwod:real;
begin
obwod:=a+b+c;
end;
function Ttrojkat.pole:real;
var p:real;
begin
p:=obwod/2;
pole:=sqrt(p*(p-a)*(p-b)*(p-c));
end;

function Tprostokat.obwod:real;
begin
obwod:=2*a+2*b;
end;
function Tprostokat.pole:real;
begin
pole:=a*b;
end;

var t: Ttrojkat;
    p: Tprostokat;
begin
writeln('Podaj dl bokow trojkata');
readln(t.a, t.b, t.c);
writeln('Pole=', t.pole);
readln;
writeln('Podaj dl bokow prostokąta');
readln(p.a, p.b);
writeln('Obwod prostokata=', p.obwod);
readln;
end.
```

Przykłady powyższe ilustrują bardzo ważną cechę programowania obiektowego:

Możliwość enkapsulacji (zamknięcia) składowych (pól i metod) wewnątrz obiektu.

Druga bardzo ważna cechą – **dziedziczenie obiektów**.

Obiektu mogą być łatwo rozbudowywane – wyposażane w nowe składowe – bez zmiany zapisu (deklaracji) istniejącego obiektu.

Przykład: (p6):

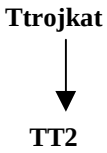
```
type Ttrojkat= object
               a, b, c: real;
               function obwod:real;
               function pole:real;
               end;

function Ttrojkat.obwod:real;
begin
obwod:=a+b+c;
end;
function Ttrojkat.pole:real;
var p:real;
begin
p:=obwod/2;
pole:=sqrt(p*(p-a)*(p-b)*(p-c));
end;

type TT2= object(Ttrojkat)
          function ha:real;
          end;
function TT2.ha:real;
begin
ha:=2*pole/a;
end;

var t: Ttrojkat;
    t2: TT2;
begin
writeln('Podaj dl bokow trojkata');
readln(t.a, t.b, t.c);
writeln('Pole=', t.pole);
//writeln('Wysokosc na bok a = ', t.ha);
readln;
writeln('Podaj dl bokow drugiego trojkata');
readln(t2.a, t2.b, t2.c);
writeln('Pole=', t2.pole);
writeln('Wysokosc na bok a = ', t2.ha);
readln;
end.
```

Program zawiera powtórzoną z przykładu poprzedniego deklarację typu **Ttrojkat**. Deklaracja drugiego typu obiektowego rozpoczyna się od zapisu:

```
type TT2= object(Ttrojkat)
```

co wskazuje że obiekt **TT2** dziedziczy całe wyposażenie od obiektu przodka **Ttrojkat**. W definicji tego typu umieszcza się jedynie nowe składowe w tym przypadku opis metody **ha** której zadaniem jest obliczenie wysokości trójkąta opuszczonej na bok **a**.

Pomiędzy typami **TT2** i **Ttrojkat** zachodzi związek hierarchiczny:



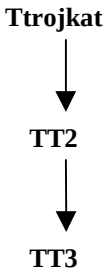
Ttrojkat jest przodkiem TT2. TT2 jest potomkiem Ttrojkat. Typ TT2 posiada wszystkie składowe typu Ttrojkat i dodatkowo metodę ha. Można sobie wyobrazić dalszą rozbudowę typu TT2 np.:

```
type TT3= object(TT2)
          function hb:real;
          function hc:real;
          end;
function TT2.hb:real;
begin
hb:=2*pole/b;
end;
function TT2.hc:real;
begin
hc:=2*pole/c;
end;
```

Opisany wyżej typ TT3 rozpoczyna się od zapisu:

```
type TT3= object(TT2)
```

co oznacza że dziedziczy całe wyposażenie typu TT2 w tym także wyposażenie dziedziczone z typu Ttrojkat.



Hierarchia tworzonych typów obiektowych może być dowolnie długa. Każdy typ obiektowy może dziedziczyć tylko od jednego przodka. Każdy typ obiektowy może być przodkiem wielu typów potomnych (jeden rodzic – wiele dzieci).

Zgodność typów

Wiele rodzajów zgodności:

Zgodność
Zgodność operacyjna
Zgodność w sensie przypisania

Zasadnicza zasada: typy tożsame (jednakowe) są zgodne.

Przykład (ilustrujący jak należy rozumieć tożsamość typów)

```
type
      tab   = array [1.. 20] of integer;
      tab1 = tab;
      tab2 = array [1..20] of integer;
var
      t1: tab;
      t2: tab;
      t3: tab1;
      t4: tab2;
      t6, t5: array[1..20] of integer;
```

zmienne **t1**, **t2** i **t3** są typów zgodnych.
zmienne **t6** i **t5** również są typów zgodnych.

Z przykładu wynika np. że przypisanie tablic postaci: t4:=t2, t2:=t2 są niepoprawne.
Natomiast można np. przypisywać między sobą tablice **t1**, **t2** i **t3** (oraz **t6** i **t5**)

Szczegółowe zasady dla wszelkich form zgodności i wszelkich rodzajów danych (zmiennych prostych, tablic, rekordów itp.) są rozbudowane. Z praktycznego punktu widzenia ważne jest pamiętanie przytoczonej głównej zasady i uzupełnień :

Typ real i integer są zgodnie operacyjnie – wynik operacji jest Real

Typ real i integer są zgodnie w sensie przypisania – pod zmienną Real można przypisywać wyrażenie typu integer.

Zgodność w sensie przypisania obiektów: (patrz także Podstawy Delphi 3.5; str 25)

Przykład (p6a)

```
type Ttrojkat= object
               a, b, c: real;
               function obwod:real;
               function pole:real;
               end;

function Ttrojkat.obwod:real;
begin
obwod:=a+b+c;
end;
function Ttrojkat.pole:real;
var p:real;
begin
p:=obwod/2;
```

```
pole:=sqrt(p*(p-a)*(p-b)*(p-c));
end;

type TT2= object(Ttrojkat)
          function ha:real;
          end;
function TT2.ha:real;
begin
ha:=2*pole/a;
end;

var t, ta: Ttrojkat;
    t2: TT2;
begin
writeln('Podaj dl bokow trojkata');
readln(t.a, t.b, t.c);
writeln('Pole=', t.pole);
//writeln('Wysokosc na bok a = ', t.ha);
readln;
ta:=t;
writeln('Obwod po przepisaniu=', ta.obwod);
readln;
writeln('Podaj dl bokow drugiego trojkata');
readln(t2.a, t2.b, t2.c);
writeln('Pole=', t2.pole);
writeln('Wysokosc na bok a = ', t2.ha);
readln;
t:=t2;
writeln('Pole po przepisaniu=', t.pole);
readln;
end.
```

Program zawiera definicje typów jak poprzednio. Zdefiniowano 3 zmienne obiektowe.
Przypisanie

```
ta:=t
```

jest poprawne - zmienne są tego samego typu.

Przypisanie

```
t:=t2
```

jest poprawne - zmienne są typów zgodnych w sensie przypisania.

Komponenty Delphi

Wszystkie komponenty Delphi rozmieszczone na paletach są obiektami, a więc można je sobie wyobrażać jako pojemniki grupujące składowe:

pola (dokładniej właściwości)	opisujące różne cechy obiektu
metody	procedury działające na polach obiektu

Komponenty są obiektami przeznaczonymi do specyficznych zastosowań – są elementami interfejsu aplikacji działającej w środowisku systemu operacyjnego Windows.

Z tego powodu muszą podlegać prawom obowiązującym w tym środowisku – stąd mogą być do nich adresowane zdarzenia odbierane przez system, takie jak:

- wskazanie myszą
- kliknięcie na komponent
- itp.

Komponenty muszą zatem reagować na zdarzenia występujące w systemie.
Stąd z każdym komponentem można skojarzyć wiele specjalizowanych metod – procedur obsługi zdarzeń. Szablon procedury obsługi zdarzenia jest generowany przez Delphi automatycznie. Zadaniem programisty jest uzupełnienie szablonu o instrukcje wykonywane w przypadku wystąpienia zdarzenia.

Z punktu widzenia składni Pascala zdarzenia są „właściwościami typu proceduralnego" - patrz K. Strzałkowski „Podstawy Delphi".

Wśród komponentów Delphi najważniejszy jest Form – formatka. Jest to komponent reprezentujący aplikację i grupujący pozostałe komponenty, które muszą być rozmieszczone na formatkach. Formatka jest właścicielem (owner) wszystkich rozmieszczonych na miej komponentów. Są one wymienione jako składowe typu obiektowego formatki Form.

Wśród pozostałych komponentów można znaleźć jednak takie, które też mogą być traktowane jako pojemniki na inne komponenty. Są to np. Panel i GroupBox. Na tych komponentach można rozmieszczać inne. Komponenty-pojemniki inne niż Form są rodzicami (parent) dla rozmieszczonych na nich innych komponentów. Komponenty składowe są np. przemieszczane razem z komponentem rodzica, mogą mieć niektóre wspólne cechy itp.
Właścicielem komponentów (owner) może być jednak tylko formatka.

Hierarchia komponentów (patrz Podstawy Delphi)

Komponenty niewizualne i sterujące (patrz Podstawy Delphi)

Komponenty graficzne i okienne (patrz Podstawy Delphi)

Ognisko wejścia (patrz Podstawy Delphi)

Obsługa błędów w aplikacji Delphi (patrz Podstawy Delphi)

Instrukcja try … except (patrz Podstawy Delphi)

Przykład w aplikacji konsolowej:

```
program p2;
uses
  SysUtils;

{dla działania obsługi wyjątków w aplikacji konsolowej trzeba
                   dołączyć SysUtils}
{$APPTYPE CONSOLE}

var y, x: integer;
begin
```

```
 try
    Writeln('Podaj dzielnik liczby 100');
    Readln(x);
    y := 100 div x;
    Writeln('Wynik=',y);
 except
    Writeln('Nie wykonam dzialania bo nie mozna dzielic przez zero!')
 end;
 writeln('KONIEC PROGRAMU');
 readln;
end.
```

Delphi "przejmuje" obsługę błędów stąd jeżeli
opcja: Tool/Debbuger Options...
zakładka: Language Exceptions
znacznik: Stop on Delphi Exceptions
jest włączony to efekt nie jest widoczny.

Zastosowanie zdarzenia *OnExit* i konstrukcji:

```
   try
      StrToFloat(Edit1.Text);
   except
      MessageDlg('Komunikat bledu', mtError, [mbOK], 0);
      Edit1.SetFocus;
   end;
```

do obsługi błędnych wpisów liczb w komponencie Edit.