Programowanie obiektowe w Pascalu

Po co obiekty?

Przykład – Obliczenie obwodu trójkąta

Sposób tradycyjny:

```
var a,b,c: real;
function obwod: real;
begin
obwod:=a+b+c;
end;

begin
readln(a, b, c);
writeln('Obwod=', obwod);
 . . .
```

Powyższy sposób zapisu jest najbardziej oczywisty i zupełnie wystarczający jeśli naszym celem jest rozwiązanie tylko tego problemu.
Inaczej jest jeżeli jest to jedynie fragment poważnego zadania, do rozwiązania którego zaangażowany jest zespół programistów. Oprogramowanie zagadnień związanych z trójkątem należy do zadań jednego programisty, a inni rozwiązują pozostałe problemy. Wówczas istnieje konieczność uzgodnień oznaczeń stosowanych przez poszczególnych programistów. Stosowane oznaczenia zmiennych a, b, c oraz nazwa funkcji: obwod nie może być zastosowana przez innych. Jeżeli rozwiązywany problem jest duży, opisywany b. wieloma parametrami, to ciągłe uzgadnianie oznaczeń jest b. kłopotliwe i prawie nieuchronnie prowadzi do błędów.

Pewne złagodzenie problemu zachodzi jeśli zapiszemy rozwiązanie następująco:

```
type trojkat =  record
                a, b, c: real;
                end;
function obwod:real;
begin
obwod:=t.a+t.b+t.c;
end;
var t: trojkat;

begin
readln(t.a, t.b, t.c);
writeln('Obwod=', obwod);
 . . . .
```

W takim przypadku wszystkie oznaczenia parametrów trójkąta mogą być dowolne, uzgodnienia wymaga tylko nazwa jednej zmiennej (rekordowej) oraz nazwy zapisanych procedur i funkcji. Osoba, której zadaniem jest oprogramowanie wszystkich zagadnień

związanych z trójkątem może przyjąć zupełnie dowolny system oznaczeń zmiennych – wszystkie one będą polami rekordu i co za tym idzie stosowane w wynikowym programie z prefiksem-nazwą rekordu co pozwoli na ich odróżnianie od innych nazw.

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

```
type trojkat =  object
                a, b, c: real;
                function obwod:real;
                end;
function trojkat.obwod:real;
begin
obwod:=a+b+c;
end;
var t: trojkat;

begin
readln(t.a, t.b, t.c);
writeln('Obwod=', t.obwod);
 . . . .
```

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** zgodnie ze składnią:

```
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.

W przykładzie typu Trojkat 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:

  ```
  procedure. Trojkat obwod;
  begin
  Trojkat obwod := Trojkat.a + Trojkat.b + Trojkat.c;          {błędne odwołania do pól}
  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).

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

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

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

begin
readln(t.a, t.b, t.c);
writeln('Obwod=', t.obwod, ' Pole=', t.pole);
 . . . .
```

Wszystkie metody posiadają niejawny parametr Self reprezentujący obiekt przekazywany przez zmienną. Oznacza to, że do składowych obiektu można odwoływać się także z użyciem prefiksu Self:

Self **.** <nazwa składowej>

Możliwość ta na ogół nie jest wykorzystywana – prościej i równie skutecznie działa odwołanie bez prefiksu Self. Czasami jednak parametr Self jest przydatny, np. gdy pola w rekordzie zadeklarowanym lokalnie w metodzie mają te same nazwy co pola obiektu. Np. w przypadku metody zadeklarowanej następująco:

```
procedure Trojkat.Przykład;
var rek: record
         a: Real;
         end;
begin
with rek do
  begin
  a:=1;          {odwołanie do pola rekordu lokalnego}
  Self.a:=1;     {odwołanie do pola obiektu}
  ...
  end;
...
end;
```

parametr Self umożliwia rozróżnienie odwołań do pola rekordu lokalnego i obiektu.

Hermetyzacja

Zasadą programowania obiektowego jest hermetyzacja dostępu do pól obiektu. W myśl tej zasady nie należy (choć można) korzystać z tych pól inaczej jak za pośrednictwem metod. Każdy obiekt powinien być wyposażony w odpowiednią ilość metod zapewniających inicjowanie pól wartościami oraz metod zwracających wyznaczone wartości.
Poniżej zamieszczono deklarację typu obiektowego spełniającego podobne zadania jak zdefiniowany wcześniej obiekt *Trojkat*. Dzięki zdefiniowaniu w nim dodatkowych metod

UstawDane i *ZwrocDane* możliwa jest realizacja zasady dostępu do pól tylko za pomocą metod.

```
unit BibTr;
interface
type trojkat =  object
                a, b, c: real;
                function obwod: real;
                function pole: real;
                procedure UstawDane(x, y, z: Real);
                procedure ZwrocDane(var x, y, z: Real);
                function BokA: real;
                end;

implementation

function trojkat.obwod:real;
begin
obwod:=a+b+c;
end;
function trojkat.pole:real;
var p: real;
begin
p:=obwod/2;
pole:=sqrt(p*(p-a)*(p-b)*(p-c));
end;
procedure trojkat.UstawDane(x, y, z: Real);
begin
a := x; b := y; c:= z;
end;
procedure trojkat.ZwrocDane(var x, y, z: Real);
begin
x := a; y := b; z:= a;
end;
function BokA: real;
begin
BokA:=a;
end;

end.
```

Deklaracja typu *Trojkat* została umieszczona w module. W takim przypadku wyszczególnienie treści metod zapowiedzianych w deklaracji typu obiektowego powinno być umieszczone w części implementacyjnej modułu.

Program wykorzystujący obiekt zadeklarowany w module *BibTr* może być sformułowany następująco:

```
uses BibTr;

var t: Trojkat;
begin
t.UstawDane(4, 5, 6);
writeln('Pole trojkata=', t.Pole:10:2);
end.
```

W programie zgodnie z zasadą hermetyzacji wszystkie odwołania do pól obiektu realizowane są za pomocą metod.

Dyrektywy Private i Public

W definicji typu obiektowego mogą występować dodatkowe dyrektywy **private** i **public**. Obie służą do sterowania dostępem do składowych obiektu. Znaczenie dyrektyw jest następujące:

- **Private** – oznacza ograniczenie dostępu do składowych obiektu zadeklarowanych poniżej tej dyrektywy. Składowe te są widoczne jedynie wewnątrz modułu, w którym zdefiniowano obiekt.

- **Public** – przywraca pełną dostępność dla składowych deklarowanych poniżej. Używana do zakończenia zakresu dyrektywy **private**.

Przykład

Zmiana deklaracji typu obiektowego w przytaczanym powyżej module *BibTr* na:

```
type trojkat =  object
                  function obwod: real;
                  function pole: real;
                  procedure UstawDane(x, y, z: Real);
                  procedure ZwrocDane(var x, y, z: Real);
                private
                  a, b, c: real;
                end;
```

powoduje, że w programie wykorzystującym ten moduł dostęp do pól obiektu może być realizowany tylko za pośrednictwem metod *UstawDane* i *ZwrocDane*. Bezpośrednie odwołania do tych pól są zabronione.
Należy podkreślić, że ograniczenie dostępu do pól za pomocą dyrektywy **private** funkcjonuje jedynie poza modułem, w którym zadeklarowano typ obiektowy. Wewnątrz tego modułu składowe obiektu są w pełni widoczne. Obiekty zadeklarowane w jednym module mogą wzajemnie swobodnie korzystać z dostępu do swoich prywatnych składowych.

Dziedziczenie

Dziedziczenie jest ważną cechą programowania obiektowego pozwalającą na łatwą rozbudowę, a także modyfikację opracowanych obiektów. Istotne jest, że rozbudowa obiektu nie wymaga ingerencji w już opracowany kod programu. Co więcej, wystarczy by moduły zawierające definicje modyfikowanych typów obiektowych były dostępne w postaci skompilowanej.

Uzupełnienie przedstawionego powyżej typu obiektowego *Trojkat* o metody wyznaczania wysokości oraz promienia koła wpisanego w trójkąt może być dokonane następująco:

```
unit BibTr2;
interface

uses BibTr;
type trojkat2 = object(Trojkat)
                function wysokoscA: real;
                function Rwpisany: real;
                end;

implementation

function trojkat2.wysokoscA: real;
begin
wysokoscA:=2*pole/BokA;
end;
function trojkat2.Rwpisany: real;
begin
Rwpisany:=4*pole/obwod;
end;

end.
```

Powyższy moduł powinien być zapisany w pliku BIBTR2.PAS. W czasie kompilacji powinien być dostępny moduł BIBTR.PAS (wystarczy wersja półskompilowana – BIBTR.DCU)

Typ wymieniony w nawiasach za słowem kluczowym **object** nazywany jest typem bazowym lub przodkiem. Typ definiowany z wykorzystaniem dziedziczenia nazywany jest typem potomnym. Dla typu *trojkat* typem bazowym jest typ *trojkat2*. Typ *trojkat2* jest typem potomnym typu *trojkat*.
Obiekt *trojkat2* posiada składowe - pola i metody odziedziczone od typu *trojkat* a ponadto metody *WysokoscA* i *Rwpisany* zdefiniowane bezpośrednio.

Przykładowe wykorzystanie typu *trojkat2* może być następujące:

```
uses BibTr2;

var t: Trojkat2;

begin
with t do
  begin
  UstawDane(4, 5, 6);
  writeln('Pole trojkata=', t.Pole:10:2);
  writeln('Wysokosc opuszczona na bok A: ', wysokoscA:10:2);
  writeln('Promień okręgu wpisanego: ', Rwpisany:10:2);
  end;
end.
```

Dziedziczenie może być wielopoziomowe tj. każdy potomek może posiadać własnego potomka drugiego i dalszych stopni.
Każdy potomek dziedziczy zarówno składowe swojego bezpośredniego przodka jak i składowe wszystkich dalszych przodków.
Obiekt *trojkat2* może być wykorzystany do zdefiniowania kolejnego potomka:

```
unit BibTr3;
interface
uses BibTr2;

type trojkat3 = object(trojkat2)
                  function UstawDane(x, y, z: real): boolean;
                  function Ropisany: real;
                private
                  alfa, beta, gamma: real;
                end;

implementation

function trojkat3. UstawDane(x, y, z: real): boolean;
var r, p: real;
begin
if (x>y+x) or (y>x+z) or (z>x+y) then UstawDane:=false
else begin
      a := x; b := y; c: = z;
      p:=obwod/2;
      r:=pole/p;
      alfa:=2*arctan((r/(p-a);
      beta:=2*arctan((r/(p-b);
      gamma:=2*arctan((r/(p-c);
      UstawDane:=true;
      end;
end;
```

```
function trojkat3.Ropisany: real;
begin
Ropisany:=a/2/sin(alfa);
end;

end.
```

Zaprezentowana wyżej deklaracja typu *Trojkat3*, a w szczególności metody *UstawDane* jest poprawna tylko wtedy gdy w bazowym typie *Trojkat* składowe-pola a, b, c są zadekretowane jako publiczne (metoda *UstawDane* zawiera odwołania do tych składowych).

Przy konstruowaniu obiektów potomnych należy przestrzegać następujących zasad:

- Obiekt może posiadać tylko jednego bezpośredniego przodka.
- Dziedziczenie obiektów można organizować w kaskadę, gdzie jeden jest potomkiem drugiego, a jednocześnie przodkiem następnego obiektu.
- Obiekt może mieć kilku potomków (równoległych).
- Odmiennie jak nazwy metod, nazwy pól obiektów w hierarchii dziedziczenia nie mogą się pokrywać. W szczególności żadne pole obiektu potomnego nie może posiadać nazwy identycznej z polem przodka (bezpośredniego ani dalszego).
- Metoda może mieć nazwę identyczną jak nazwa metody przodka. Jeżeli taki fakt następuje, to metoda przodka zostaje przesłonięta i staje się niedostępna. Mechanizm przesłaniania metod obowiązuje także w hierarchii wielopoziomowej. Wywołanie metody powoduje każdorazowo przeszukiwanie całej hierarchii dziedziczenia począwszy od danego obiektu poprzez najbliższego przodka do dalszych lecz tylko do momentu znalezienia metody. Metody przodków są przesłaniane jeśli metoda w aktualnym obiekcie ma poszukiwana nazwę,

Z ostatniej ze sformułowanych powyżej zasad wynika, że zdefiniowanie metody *UstawDane* w typie *Trojkat3* powoduje przesłonięcie metody przodka o tej samej nazwie.

Wywoływanie metody o powtórzonej nazwie demonstruje poniższy przykład:

```
uses BibTr, BibTr2, BibTr3;

var t1: Trojkat2;
      t2: Trojkat3;
      a, b, c: real;

begin
writeln('Podaj długości boków trójkąta');
readln(a, b, c);
with t2 do
  begin
  if UstawDane(a, b, c) then
    begin
```

```
    writeln('Pole trojkata=', Pole:10:2);
    writeln('Promień okręgu opisanego =', Ropisany:10:2);
    end;
  end;
with t1 do
  begin
  UstawDane(a, b, c);
  writeln('Pole trojkata=', Pole:10:2);        { błąd jeśli liczby a, b, c nie tworzą trójkąta }
  end;
end.
```

W przykładzie zadeklarowano dwie zmienne obiektowe. Wywołanie metody *UstawDane* dla obu zmiennych powinno być różne. Ponadto, w przypadku zmiennej typu Trójkat3 możliwe jest wykluczenie możliwości powstania błędu gdy użytkownik poda 3 wartości, które nie tworzą trójkąta.

W uzupełnieniu zasad dotyczących dziedziczenia należy dodać, że metody przodków przesłaniane z powodu ponownego użycia identycznych nazw mogą być wywoływane z wnętrza metod obiektów potomnych. W wywołaniu takim nazwa metody musi być poprzedzona nazwą typu obiektowego w którym zdefiniowano przesłoniętą metodę.
Definicja metody *UstawDane* w typie *Trojkat3* może być zatem zapisana w postaci:

```
function trojkat3. UstawDane(x, y, z: real): boolean;
var r, p: real;
begin
if (x>y+x) or (y>x+z) or (z>x+y) then UstawDane:=false
else begin
      Trojkat2.UstawDane(x, y, z);
      p:=obwod/2;
      r:=pole/p;
      alfa:=2*arctan((r/(p-BokA);
      beta:=2*arctan((r/(p-BokB);
      gamma:=2*arctan((r/(p-BokC);
      UstawDane:=true;
      end;
end;
```

Taka postać metody *UstawDane* wydaje się bardziej odpowiednia – zamiast powtórzenia treści już zdefiniowanej metody zawiera jej wywołanie. Dodatkowo, jej działanie jest poprawne także w sytuacji gdy składowe-pola a, b, c w typie bazowym są zadeklarowane jako prywatne.