PRESEK
List za mlade matematike, fizike, astronome in računalnikarje

ISSN 0351-6652
Letnik **20** (1992/1993)
Številka 6
Strani 358–365

Jože Marinček:

# UVOD V SVET OBJEKTOV

Ključne besede: računalništvo, programiranje.

Elektronska verzija: http://www.presek.si/20/1151-Marincek.pdf

# UVOD V SVET OBJEKTOV

## Objekti

Vsi poznamo zapise (**record**-e) v pascalu. Omogočajo nam, da zberemo skupaj med seboj povezane podatke. Objekt dobimo, če dodamo še podprograme, s katerimi obdelujemo te podatke. Tem podprogramom pravimo *metode*. Podatki določajo, kaj objekt *ve*. Metode določajo, kaj objekt *zna*. Na primer: celo število (`integer`) sicer ve za svojo vrednost, a si z njo ne zna pomagati. Naredimo objekt, ki bo svojo vrednost znal izpisati:

```
type
  IntegerPlus = object
    vem: integer;
    procedure znam;
  end;


procedure IntegerPlus.znam;
 begin
  write(' ',vem,' ')
 end; {IntegerPlus.znam}
```

Enostaven primer uporabe bi bil:

```
var
 I: IntegerPlus;
begin
 I.vem := 1;
 I.znam;                                    { izpiše "␣ 1␣" }
end.
```

Seveda so lahko podatki, ki jih združuje objekt, mnogo bolj zapleteni kot so v našem primeru. Zato poznamo dve posebni metodi, *konstruktor* in *destruktor*. Konstruktor je metoda, ki naj bi jo uporabili prvo. Ponavadi jo izkoristimo za prireditev začetnih vrednosti. Destruktor naj bi uporabili kot zadnjo.

## Dedovanje

Dedovanje je pomembna lastnost objektov. Omogoča nam, da ustvarimo nove objekte, ki "vedo" in "znajo" vse, kar znajo stari objekti, lahko pa vedo ali znajo še kaj več. Naredimo objekt celega števila, ki se zna izpisati, zna pa tudi povedati, ali je praštevilo:

```
type
  IntegerPlusPlus = object(IntegerPlus)
    function Prastevilo: boolean;
  end;


function IntegerPlusPlus.Prastevilo: boolean;
 var
  test: boolean;
  i: integer;
 begin
  test := Odd(vem);                                  { Praštevilo je vedno liho. }
  i := 3;
  while test and (Sqr(i) <= vem) do begin
    test = vem mod i <> 0;
    i := i + 2;                  { Ker je vem liho število, ima le lihe delitelje. }
  end;
  Prastevilo := test
 end; {IntegerPlusPlus.Prastevilo}


var
 I: IntegerPlusPlus;
begin
 I.vem := 1;
 I.znam;                                                        { izpiše '␣ 1␣" }
 writeln(' 'Je prastevilo?',I.Prastevilo)      { izpiše " Je␣ praštevilo?␣ TRUE"}
end.
```

Vidimo, da je metoda lahko tudi funkcijski podprogram. Vsaka metoda pozna vse, kar objekt ve, torej vse njegove podatke, in jih lahko poljubno uporablja in spreminja.

Objekt `IntegerPlusPlus` ve in zna vse, kar zna objekt `IntegerPlus`. Zato nam metode `znam` ni bilo treba še enkrat deklarirati in kasneje napisati. Še več, objekt `IntegerPlus` lahko zapremo v neko enoto (**unit**) in prijateljici odstopimo samo prevedeno datoteko (.TPU), pa bo ona še vedno mogla definirati objekt `IntegerPlusPlus`, hkrati pa ne bo mogla šariti po našem programu (kar še posebej velja za njenega mlajšega brata). Pravimo, da je `IntegerPlusPlus` *naslednik* objekta `IntegerPlus`. Pascalu to sporočimo tako, da za besedico **object** napišemo v oklepaju ime prednika. Tudi vsem morebitnim naslednikom objekta `IntegerPlusPlus` bomo rekli nasledniki objekta `IntegerPlus`. Torej je lahko vsak objekt oče, ded, praded... cele družine objektov.

Zaradi dedovanja objektov so postala drugače stroga pascalova pravila o prirejanju nekoliko ohlapnejša. Do sedaj je veljalo, da lahko spremenljivki

določenega tipa priredimo samo spremenljivko oziroma izraz istega tipa. Edina izjema so bile spremenljivke tipa `Real`, ki smo jim mogli prirediti tudi celoštevilske izraze. Pri objektih pa velja, da mu smemo prirediti objekt istega tipa ali njegovega naslednika. Če objekt, ki mu prirejamo vrednost, ne ve ali zna vsega, kar mu prirejamo, se odvečna informacija enostavno izgubi. Pravilo je torej enostavno: izraz mora biti sposoben zapolniti celoten objekt. Drugače bi lahko ostalo kakšno polje brez prirejene vrednosti, čemur pa se moramo na vsak način ogniti.

Nasledniki objekta lahko metode objekta (eno ali več) tudi spremenijo. Prepišimo objekt `IntegerPlusPlus` tako, da bo pred praštevilom izpisana zvezdica! Podprogram `Pokazi` pa bo izpisal vrednost enemu ali drugemu objektu.

```
type
  IntegerPlusPlus = object(IntegerPlus)
    procedure znam;
    function Prastevilo: boolean;
  end;


function IntegerPlusPlus.Prastevilo: boolean;
 var
  test: boolean;
  i: integer;
 begin
   {... kot prej ...}
 end;


procedure IntegerPlusPlus.znam;
 begin
  write(' ');
  if Prastevilo then write('*');
  write(vem,' ')
 end; {IntegerPlusPlus.znam}


procedure Pokazi(var o: IntegerPlus);
{ Sprejme tudi argument tipa IntegerPlusPlus in sploh vseh naslednikov. }
 begin
  o.znam
 end; {Pokazi}


var
 I: IntegerPlus;
 J: IntegerPlusPlus;
```

```
begin
 I.vem := 17;  J.vem := 23;
 I.znam;                                  { izpiše "␣ 17␣" }
 J.znam;                                  { izpiše "␣ *23␣" }
 I := J;
 I.znam;                                  { izpiše "␣ 23␣" }
 Pokazi(I);                               { izpiše "␣ 23␣" }
 Pokazi(J);                               { izpiše "␣ 23␣" }
end.
```

Bodimo pozorni! Nasledniki lahko spreminjajo metode (in dodajajo nove), podatkovnemu polju objekta pa tipa ni moč spreminjati. Nasledniki lahko le dodajajo nova podatkovna polja.

### Navidezne metode

Pri objektih ločimo dve vrsti metod: statične in navidezne. S statičnimi metodami se ukvarja prevajalnik. Ko pri prevajanju sreča objekt, ki uporabi neko svojo statično metodo, reče: "Aha! Točno vem, kaj hočeš" in to tudi naredi. Včasih pa to ni najbolje. Tako podprogram `Pokazi` ne more izkoristiti tega, da je ob drugem klicu parameter J pravzaprav tipa `IntegerPlusPlus`, ki zna izpisati zvezdico pred praštevilom. Ko je pascal prevajal podprogram `Pokazi`, mu je bilo povsem jasno, da ima opravka z objektom tipa `IntegerPlus`, torej je tudi "vedel", katero metodo bo uporabil.

Hoteli bi, da se prevajalnik šele ob klicu metode odloči, katero metodo bo uporabil: tisto iz objekta `IntegerPlus`, ono iz objekta `IntegerPlusPlus`, ali pa morebitno tretjo metodo iz nekega naslednika teh dveh objektov. To nam omogočajo navidezne metode. Trik je enostaven: za imenom metode napišemo rezervirano besedico **virtual**. Ko objekt pokliče tako metodo, si prevajalnik samo zapiše, da na tem mestu od objekta pričakujemo določeno akcijo.

Vsak objekt, ki uporablja vsaj eno navidezno metodo, se mora na tako delo pripraviti. To stori s posebno metodo, konstruktorjem. Ta poskrbi, da se podatki o objektu zapišejo v posebno tabelo, brez katere navidezne metode ne znajo delati. Prepišimo sedaj naša objekta z navideznimi metodami:

```
type
  IntegerPlus = object
    vem: integer;
    constructor dobro_jutro(n: integer);
    procedure znam; virtual;
  end;
```

```
IntegerPlusPlus = object(IntegerPlus)
  constructor dobro_jutro(n: integer);
  procedure znam; virtual;
  function Prastevilo: boolean;
end;


constructor IntegerPlus.dobro_jutro(n: integer);
 begin
  vem := n
 end; {IntegerPlus.dobro_jutro}

constructor IntegerPlusPlus.dobro_jutro(n: integer);
 begin
  vem := n
 end; {IntegerPlusPlus.dobro_jutro}
```

Seveda ni potrebno, da konstruktor nastavi začetne vrednosti objekta. Kode, ki konstruktor loči od ostalih metod, tako ali tako nikoli ne vidimo, saj jo doda prevajalnik. Konstruktor zato svoje delo opravi, tudi če je v pascalu povsem prazen. Zato mora imeti vsak objekt, ki ima vsaj eno navidezno metodo, svoj konstruktor (tisti, ki ga ima oče, ni dober).

Poglejmo, kaj si o novih objektih misli isti podprogram `Pokazi`:

```
begin
 I.dobro_jutro(17);          { Postavi vrednost polja I.vem na 17. }
 J.dobro_jutro(23);          { Postavi vrednost polja J.vem na 23. }
 Pokazi(I);                                  { izpiše "␣ 17␣" }
 Pokazi(J)                                   { izpiše "␣ *23␣" }
end.
```

V zadnjem primeru se nikoli nismo dotaknili polja `vem`. *Pravilo lepega programiranja pravi, da podatkovna polja objekta uporabljajo izključno metode tega objekta.* Skladno s tem moramo objektu `IntegerPlus` dodati vsaj še dve metodi:

```
type
  IntegerPlus = object
    vem: integer;
    constructor dobro_jutro(n: integer);
    procedure priredi(n: integer);
    function vrednost: integer;
    procedure znam; virtual;
  end;
```

```
procedure IntegerPlus.priredi(n: integer);
 begin
  vem := n
 end; {IntegerPlus.priredi}

function IntegerPlus.vrednost(n: integer);
 begin
  vrednost := vem
 end; {IntegerPlus.vrednost}
```

Sedaj že vemo, da isti metodi poznajo tudi objekti tipa `IntegerPlusPlus`.

Način, na katerega se objekti tipa `IntegerPlus` in `IntegerPlusPlus` kažejo podprogramu `Pokazi`, imenujemo *polimorfizem*. Ime metode je vedno enako, v našem primeru je to `znam`. Kaj pa se v resnici zgodi, je odvisno od objekta, kateremu metoda pripada.

Razvijmo za vajo še soroden objekt, ki bo poznal znak in ga bo znal izpisati:

```
type
  CharPlus = object
    vem: char;
    constructor dobro_jutro(c: char);
    procedure priredi(c: char);
    function vrni: char;
    procedure znam; virtual;
  end;

constructor CharPlus.dobro_jutro(c: char);
 begin
  vem := c
 end; {CharPlus.dobro_jutro}

procedure CharPlus.priredi(c: char);
 begin
  vem := c
 end; {CharPlus.priredi}

function CharPlus.vrni: char;
 begin
  vrni := vem
 end; {CharPlus.vrni}

procedure CharPlus.znam;
 begin
  writeln(' ',vem,' ')
 end; {CharPlus.znam}
```

Vidimo, da sta si objekta `IntegerPlus` in `CharPlus` zelo podobna. Kako bi lahko to sorodnost zapisali tudi v pascalu?

Objekti močno poudarijo staro pravilo (katerega nihče ne posluša), da velja temeljito razmisliti in proučiti problem, predno se lotimo programiranja. Obema objektoma, `IntegerPlus` in `CharPlus`, je skupno to, da znata izpisati neki podatek. Kateri podatek izpišeta in kako to naredita, pa je seveda njuna stvar. Sedaj skušajmo vse sorodnosti potegniti iz obeh objektov v nov, skupni objekt:

```
type
  Osnova = object
    constructor dobro_jutro;
    procedure znam; virtual;
  end;

constructor Osnova.dobro_jutro;
 begin
 end; {Osnova.dobro_jutro}

procedure Osnova.znam;
 begin
 end; {Osnova.znam}
```

Metoda `Osnova.znam` je prazna, saj osnovni objekt ne nosi nobene informacije. Njegova naloga je samo to, da opiše, kaj je skupnega vsem njegovim naslednikom. Vsak naslednik pa bo že poskrbel, da se bo izpisal na pravilen način. Sorodnost med objekti (v našem primeru med `IntegerPlus` in `CharPlus`) bomo ohranili z dedovanjem, drugačne izpeljave metod pa nam bodo prinesle razlike.

```
type
  IntegerPlus = object(Osnova)
    vem: integer;
    constructor dobro_jutro(n: integer);
    procedure priredi(n: integer);
    function vrednost: integer;
    procedure znam; virtual;
  end;

  CharPlus = object(Osnova)
    vem: char;
    constructor dobro_jutro(n: char);
    procedure priredi(n: char);
    function vrednost: char;
    procedure znam; virtual;
  end;
```

Manjkajoče metode lahko bralec napiše sam.

Sedaj lahko spremenimo še podprogram Pokazi:

```
procedure Pokazi(var o: Osnova);
 begin  o.znam  end;
```

Tako napisan podprogram pa pravilno izpiše objekte vseh v članku omenjenih tipov. Tako nam objekti omogočijo, da nam ni treba pisati enega podprograma za izpis celih števil, drugega za izpis znaka, še tretjega za nov tip, ki ga nenadoma potrebujemo. Dovolj je, da ga napišemo enkrat, vsakemu od teh tipov pa razložimo, kakšen je v resnici njegov izpis. Poenostavitve zaradi dela z objekti pridejo do izraza, ko se postopki zapletejo. Tako nam objekti prihranijo ogromno dela, ko se lotimo risanja grafičnih objektov (daljice, krogi itd.). Prav pri takšnih postopkih objekti resnično zaživijo.

*Jože Marinček*