

# Objektno orientirano programiranje – na kratko OOP

metoda razmišljanja:

strukturirano programiranje: problemsko rešujemo problem

objektno programiranje: dogodkovno razmišljanje in razlaga problema

Tri glavne lastnosti OOP so:

- enkapsulacija: združevanje lastnosti in obnašanja v enotno strukturo, ki omogoča ločeno obravnavo posameznih razredov in skrivanje podrobnosti implementacije.
- Dedovanje: definirato objekt in nato ga uporabiti za gradnjo hierarhije objektov; vsak objekt deduje lastnosti in obnašanje očeta.
- Polimorfizem: podamo enako ime metodam v vseh objektih v hierarhiji. Toda v vsakem objektu se metoda drugače obnaša.

## Objekti:

Osnova OOP je objekt. Objekt je zaključena celota, ki vsebuje opis lastnosti in obnašanja nekega realnega ali abstraktnega objekta v našem programu, ampak samo tiste, ki jih trenutno potrebujemo.

Zgled:

Če prodajamo avtomobile, bomo kot lastnost objekta “avto” vodili podatke o teži avtomobila, moči motorja, barvi karoserije in podobno. Od obnašanja, pa nam bo zanimivo, kaj se zgodi, ko avto prodamo (zmanjšati stanje v skladišču, rezervirati avto za določenega kupca,...).

## Razred:

Objekte iste vrste združimo v razrede.

Zgled:

Razred nam predstavlja bilokateri avto, objekt pa je točno določen avto (FIAT – natanko določena številka in specifične lastnosti).

## Dedovanje:

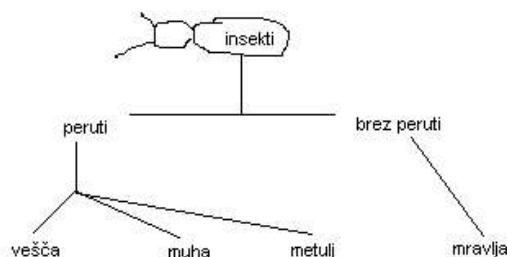
primer insketov: - dve vrsti insektov

- z peruti
- brez peruti

vsaka skupina ima ogromno podskupin (vešče, metulji, muhe, mravlje,...)

Kako klasificiramo/najdemo razliko med skupinami:

začnemo pri deblu (korenu) in drevo nato nadgrajujemo. Na nižjih nivojih je problem manj specifičen (vprašanja so enostavna – Ima peruti ali ne?). Vsak naslednji nivo je bolj specifičen in tudi vprašanja postajajo vedno bolj specifična – kompleksna. Kako globoko v drevo/hierarhijo gremo, je odvisno od nas samih/problema/zahtev.



Dobra praksa je poimenovati definirane tipe z odpsredaj črko T:

najprej pogledjmo kako bi realizirali delavca in njegovo urno postavko z podatkovnim tipom zapis.

type

```
Tdelavec=record
    ime:string[25];
    naslov:string[25];
    cena:real;
end;
Tpostavka=record
    ime:string[25];
    naslov:string[25];
    cena:real;
    ure:integer;
end;
```

ali pa kombiniramo in izpeljemo hierarhijo:

```
Tpostavka=record
    delavec:Tdelavec;
    ure:integer;
end;
```

Z objekti bi pa realizirali na naslednji način:

type

```
Tdelavec=object
    ime:string[25];
    naslov:string[25];
    cena:real;
end;
Tpostavka=object(Tdelavec)
    ure:integer;
end;
```

Objekt Tpostavka podeduje od objekta Tdelavec.

### **Lastnosti:**

Dostopanje do lastnosti objekta:

objekt je potrebno najprej kreirati

```
var
```

```
  a:Tpostavka;
```

```
begin
```

```
  a.ime := '....';
```

```
  a.naslov := '....';
```

```
...
```

```
end.
```

Dobra praksa:

čeprav lahko dostopamo do lastnosti objekta direktno je dobra praksa taka, da dostopamo do lastnosti preko metod.

Metoda je procedura ali funkcija, deklarirana znotraj objekta in povezana z lastnostmi objekta.

```

Tdelavec = record
  ime:string[25];
  naslov:string[25];
  cena:real;
end;
var
  NasDelavec:Tdelavec;
begin
  with NasDelavec do
    begin
      ime := '...';
      naslov := '...';
      cena := 5.5;
    end;
  end.

```

Problem takega prirejanja je v tem, da deluje samo na enim zapisom. Takoj ko imamo še en enak objekt je potrebno enako zapisati še enkrat. Za prireditvene potrebe izdelamo inicializacijsko proceduro (procedura, ki nam postavi vrednosti v spremenljivke).

```

Procedure initDelavec(var delavec:Tdelavec;Aime,Anaslov:string;Acena:real);
begin
  with delavec do
    begin
      ime := Aime;
      naslov := Anaslov;
      cena := Acen;
    end;
  end;
end;

```

Problem podvajanja kode smo odpravili, ampak obstaja še elegantnejša rešitev z uporabo objektov.

```

Type
  Tdelavec = object
    ime,naslov:string[25];
    cena:real;
    procedure init(novoime,novnaslov:string[25];novacena:real);
  end;

procedure Tdelavec.init(novoime,novnaslov:string[25];novacena:real);
begin
  ime := novoime;
  naslov := novnaslov;
  cena := novacena;
end;

var
  NasDelavec:Tdelavec;
begin
  nasdelavec.init('Janez','Tehnični delavec',300);
end.

```

Funkcije in procedure/metode, ki so napovedane v definiciji razreda, realiziramo posebej. Pri tem moramo povedati, kateremu razredu pripada metoda, saj lahko metoda z enakim imenom pripada več objektom. Razred določimo tako, da pred ime metode napišemo ime razreda in ga od imena metode ločimo z piko.

Pri deklaraciji objekta lahko razdelimo podatke in metode na javne in privatne. Da so javni pomeni, da jih lahko direktno uporabljamo iz programa, če so pa privatni, jih lahko uporabljamo/dostopamo samo preko metod iz našega razreda.

Type

```
Tdelavec = object
  procedure init(Aime, Anaslov: string[25]; Acena:real);
  procedure setIme(Aime:string[25]);
  procedure setNaslov(Anaslov:string[25]);
  procedure setCena(Acena:real);
  function getIme:string[25];
  function getNaslov:string[25];
  function getCena:real;
private
  ime, naslov:string[25];
  cena:real;
end;
```

```
noviObjekt = object
  lastnosti |
  metode   |   javne metode in lastnosti
private
  lastnosti |
  metode   |   privatne metode in lastnosti
end;
```

```
procedure Tdelavec.init(Aime, Anaslov: string[25]; Acena:real);
begin
  ime := Aime;
  naslov := Anaslov;
  cena := Acena;
end;
procedure setIme(Aime:string[25]);
begin
  ime := Aime;
end;
procedure setNaslov(Anaslov:string[25]);
begin
  naslov := Anaslov;
end;
procedure setCena(Acena:real);
begin
  cena := Acena;
end;
function getIme:string[25];
begin
  getIme := ime;
end;
```

```

function getNaslov:string[25];
begin
  getNaslov := naslov;
end;
function getCena:real;
begin
  getCena := cena;
end;

```

Objekti so podobni zapisom (record), zato lahko uporabljamo with stavek, tako za lastnosti, kot za metode.

```

Var
  a:Tdelavec;
begin
  with a do
  begin
    init('...','...',5);
    writeln(getIme); writeln(getNaslov);
  end;
end.

```

### Enkapsulacija

Enkapsulacija je združevanje lastnosti in metod v eno strukturo. Da objekt temeljito izdelamo, moramo izdelati za vsako lastnost tudi metodo za branje in metodo za pisanje. Če imamo samo metodo branje, lahko imenujemo lastnost “read-only” - samo za branje, kar pomeni, da je iz programa ne moremo spreminjati (spreminja se lahko samo znotraj metod).

Navadno se v pascalu dodeli lastnosti pod privatne, metode pa pod javne:

```

type
  Tinsekt = object
    ime:string[20];
    x,y:integer;
  end;
  Tperuti = object(Tinsekt)
    procedure init(ax,ay:integer);
    procedure show;
    procedure hide;
    procedure moveto(novix,noviy:integer);
  end;
  Tcebela=object(Tperuti)
  ...
  procedure init(ax,ay:integer);
  procedure show;
  procedure hide;
  procedure moveto(novix,noviy:integer);
  end;

```

Oba objekta Tperuti in Tcebela imata štiri metode. Tperuti.init in Tcebela.init za inicializacijo spremenljivk. Tperuti.show metoda nam izriše insekt z perutmi na zaslon, medtem ko metoda Tcebela.show zna izrisati čebelo na zaslon. Ravno tako velja za Tperuti.hode metoda zna izbrisati insekt z perutmi, medtem ko Tcebela.hode zna izbrisati čebelo iz zaslona. Metode show in hide se med objektoma razlikujeta.

Metodi Tperuti.moveto in Tcebela.moveto sta pa čisto enaki, kateri nam najprej pobrišeta objekt iz

zaslona, nato določita novi koordinati x in y in na koncu objekt na novo prikažeta.

```
Procedure Tperuti.moveto(novix,noviy:integer);  
begin  
  hide;  
  x := novix;  
  y := noviy;  
  show;  
end;
```

```
Procedure Tcebela.moveto(novix,noviy:integer);  
begin  
  hide;  
  x := novix;  
  y := noviy;  
  show;  
end;
```

Razlike med metodami ni, le prepisali smo metodo in ji zamenjali ime objekta. Tukaj se pojavi vprašanje, zakaj nebi kar podedovali moveto metodo iz Tperuti v Tcebela.

Vse metode do sedaj so bile statične. Omenjeni problem nam rešijo virtualne metode. Najprej si pogledjmo problem, da bomo lažje razumeli.

Problem:

Vse dokler ne vnesemo v Tcebela metode moveto, ta metoda ne deluje pravilno (podedovana). Če Tcebela pokliče podedovano metodo Tperuti.moveto, vse kar se premakne je samo objekt Tperuti. (zakaj? Ker dejansko v metodi se kličejo metode show in hide, ki pa dejansko pripadata podedovani metodi Tperuti. Zato se pokličejo tudi hide in show iz objekta Tperuti).

Ko smo izdelali metodo Tcebela.moveto in iz metode pokličemo show in hide, se pravilno čebela prestavi (pokličejo se pravilne metode za prikaz in brisanje).

Vse skupaj izvira iz problema, kako prevajalnik izdelava kodo. Če pokličemo metodo in ta kliče druge metode in če prevajalnik klicane metode ne najde, gre po hierarhiji objektov navzgor, vse dokler ne najde metode in jo izvrši. Seveda v metodi operira z spremenljivkami, ki so lokalne v tisti metodi.

## **Polimorfizem in virtualne metode**

Opisane metode so statične in včasih delajo preglavice, ker se ne obnašajo pravilno.

Virtualne metode implementirajo/nam omogočajo zelo pogljlivo orodje za generalizacijo tako imenovani polimorfizem. Polimorfizem je grška beseda in pomeni več oblik, skratka način, kako imenujemo metode z enim imenom in po hierarhiji se drugače obnašajo. Vsak objekt v hierarhiji insektov se drugače izriše na zaslon: drugi primeri, kako izrišemo kvadrat, kako izrišemo krog, kako izrišemo pravokotnik na zaslon.

Torej vsak naslednji objekt, ki ga kreiramo ima svoj način risanja in skrivanja iz zaslona, tora metoda za premikanje po zaslonu ostane vedno enaka.

## **Polimorfni objekti**

Polimorfni objekti nam omogočajo procesiranje objektov, katerih tip ni znan ob prevajanju. To ni v skladu s pascalovo logiko, ampak z logiko razmišljanja človeka.

Primer:

če imamo metode za prikaz, skrivanje in premikanje in imamo 10 objektov (muha, čebela, komar, vešča,...) in sedaj hočemo izdelati metodo, da z miško premikamo te objekte.

Po starem bi bilo potrebno za vse te objekte izdelati metodo za premikanje. Spretni programerji bi

mogoče izdelali metodo z uporabo spremenljivke, ki pove za kateri tip gre in nato klicali pravilno proceuro (npr. Case stavek). Kaj pa če bi imeli 1000 insektov?  
Prava in najbolj elegantna rešitev je uporaba polimorfizma in virtualnih metod.

### Virtualne metode (omogočajo polimorfizem)

metodo naredimo virtualno, če v deklaraciji metode dodamo rezervirano besedo `virtual`; Paziti je potrebno, če eno metodo naredimo virtualno, jo je potrebno v vseh objektih v hierarhiji definirati kot virtualno (drugače pride do napake!!)

Type

```
Tperuti=object(Tinsekt)
  constructor init(ax,ay:integer);
  procedure show; virtual;
  procedure hide; virtual;
  procedure moveto(novix,noviy:integer);
end;
Tcebela=object(Tperuti)
  constructor init(ax,ay:integer);
  procedure show; virtual;
  procedure hide; virtual;
end;
```

Kot prvo metode `moveto` ne potrebujemo več v `Tcebela`, ker sedaj se uporabljajo virtualne metode. Metodi `show` in `hide` se znotraj metode `moveto` kličejo pravilno glede na tip objekta. Ni več klica metod `show` in `hide` v objektu, kjer je metoda `moveto` implementirana – kot v statičnem objektu!

Druga novost je uporaba rezervirane besede constructor za `Tperuti.init` in `Tcebela.init`. Konstruktor je poseben tip metode, ki inicializira objekt za uporabo virtualnih metod.

Vsak objekt, ki vključuje virtualne metode MORA vsebovati konstruktor. Konstruktor je potrebno klicati preden pokličemo bilokatero virtualno metodo. Drugače pride do napak pri izvajanju!

Vsak objekt posebej je potrebno inicializirati. Ni dovolj, da neinicializiranemu objektu priredimo vrednost inicializiranega objekta.

Var

```
Mcebela,Ncebela:Tcebela;
begin
  Mcebela.init(10,10);
  Ncebela := Mcebela;
end.
```

NAROBE vsak objekt je potrebno inicializirati posebej.

```
Mcebela.init(10,10);
```

```
Ncebela.init(10,10);
```

Kaj konstruktor izdelava? Vsak objekt ima tabelo virtualnih metod(t.i. VMT – Virtual Method Table). Objekti v TVM vsebujejo virtualne metode in povezave na dejansko implementacijo. Konstruktor v TVM postavi metode in jih pravilno poveže z implementacijo. Objekt ima samo eno TVM, zato ne moremo prijeti vsebine drugega objekta, ker se pri prireditvi ne skopira TVM. Zato je potrebno objekt najprej inicializirati – poklicati konstruktor.

Statične ali virtualne metode? Uporabniško gledano naj bi bile vse metode virtualne. Torej zakaj nimamo vseh metod virtualnih? Če hočemo hitrost in optimalno razporejen pomnilnik posegamo po statičnih metodah.

## Dinamični objekti

Vsi do sedaj naštetih objekti so bili statični (kot podatkovni tip integer, real,...). Vsak objekt lahko kreiramo tudi dinamično. Pascal pozna nekaj "trikov" za lažje delo z dinamičnimi objekti. Kot že poznamo kazalci se deklarirajo z  $\wedge$ (puščico) pred podatkovnim tipom.

```
Var
  P: $\wedge$ Tcebel;
begin
  new(p);
```

Tako kot pri zapisih, se v pomnilniku kreira dovolj prostora za delo tudi z objekti. Takoj ko ima objekt virtualne metode, ga pred uporabo moramo inicializirati.

```
P $\wedge$ .init(10,10);
```

Metode se ravno tako pokliče z imenom spremenljivke in kazalcem in za piko ime metode z argumenti.

Pascal nam nudi tudi nov način za inicializacijo kazalcev nad objekti.

```
New(p,init(10,10));
```

Najprej podamo ime spremenljivke in nato ime konstruktorja z atributi.

New lahko uporabimo tudi na naslednji način:

```
type
  Pinsekt= $\wedge$ TInsekt;
var
  v:Pinsekt;
begin
  p := new(Pinsekt);
  ali
  p := new(Pinsekt,init(10,10));
end.
```

Vsak dinamično kreirani objekt je potrebno tudi izbrisati iz pomnilnika. Kot vse kazalce se tudi pri kazalcih na objekte kliče proceduro dispose:

```
dispose(p);
```

Glede na to, da objekt sam lahko vključuje več kazalcev se izdelava metodo, ki vse kazalce v objektu najprej pobriše in šele nato pobriše še samega sebe.

```
P $\wedge$ .done;
```

Metoda done naj bi vsebovale vse potrebno za brisanje iz pomnilnika (deinicializacija – vse kar je konstruktor inicializiral naj bi destruktore deinicializiral – uničil). Pascal nam ponuja posebno metodo – destructor, ki nam služi za brisanje iz pomnilnika. Destruktor je tako kot konstruktor, le konstruktor izdelava vse potrebno za operiranje z objektom, destruktore pa poskrbi za čiščenje – torej po izvedbi destruktora je kot da objekta nikoli nebi izdelali (za to poskrbi programer!!).

```
Type
Tdelavec = object
  ime:string[25];
  naslov:string[25];
  cena:real;
  constructor init(aime, anaslov:string[25]; acena:real);
  destructor done;virtual;
  function getIme:string[25];
  function getNaslov:string[25];
  function getCena:real; virtual;
end;
```



Zaradi različnih podatkovnih tipov in različnih dedovanj je dobro, da se destruktorja izdelata virtualno.

Destruktorja ni potrebno za vsak objekt, čeprav vsebuje virtualne metode. Destruktor se potrebuje samo, da se sprostijo kazalci iz pomnilnika. Lahko pa ga izdelamo v vseh objektih.

Destruktor torej uporabljamo na dinamično izdelanih objektih in virtualnimi metodami, kjer dejansko ne vemo koliko pomnilnika objekt potrebuje. Dispose lahko kličemo tako kot new, torej da povemo katera metoda je destruktor.

```
Dispose(p,done);
```