

Programarea orientată obiect (*Object Oriented Programming* - OOP) reprezintă o tehnică ce s-a impus în ultimii ani, dovedindu-se benefică pentru realizarea sistemelor software de mare complexitate. Noțiunea de obiect datează din anii '60, o dată cu apariția limbajului Simula. Există limbaje - ca Smalltalk și Eiffel - care corespund natural cerințelor programării orientate obiect, fiind concepute în acest spirit, alături de care o serie de alte limbaje procedurale, printre care C++ și Turbo Pascal, au fost dezvoltate astfel încât să ofere suport pentru această tehnică. În prezent există în funcțiune sisteme software de mare anvergură realizate în tehnica programării orientată obiect, principiile ei fiind suficient de bine clarificate, astfel încât să se treacă din domeniul cercetării în cel al producției curente de programe.

10.1 Modelul de date orientat pe obiecte

OOP reprezintă o abordare cu totul diferită față de programarea funcțională, devenită deja “clasică”. Dacă în programarea clasică programatorul era preocupat să răspundă la întrebarea “ce trebuie făcut cu datele?”, adică să definească proceduri care să transforme datele în rezultate, în **OOP** accentul cade asupra datelor și legăturilor care există între acestea, ca elemente prin care se modelează obiectele lumii reale. Se poate afirma, într-o primă analiză, că **OOP** organizează un program ca o colecție de obiecte, modelate prin date și legături specifice, care interacționează dinamic, adică manifestă un anumit “comportament”, producând rezultatul scontat. În general, pentru modelul de date orientat pe obiect, se consideră definiții următoarele concepte: obiect, încapsulare, moștenire și polimorfism.

1. Obiectul este modelul informațional al unei entități reale, care posedă, la un anumit nivel, o mulțime de proprietăți și care are, în timp, un anumit comportament, adică manifestă o reacție specifică în relațiile sale cu alte obiecte din mediul său de existență. Ca model, un obiect este o unitate individualizabilă prin nume, care conține o mulțime de date, proceduri și funcții. Datele descriu

proprietățile și nivelul acestora, iar funcțiile și procedurile definesc comportamentul.

Având în vedere proprietățile comune și comportamentul similar ale entităților pe care le modelează, obiectele pot fi clasificate, adică împărțite în mulțimi. O mulțime de obiecte de același fel constituie o clasă, care poate fi descrisă prin modelul comun al obiectelor sale.

De exemplu, în figura 10.1, numerele raționale, ca perechi de numere întregi de forma (Numarator, Numitor) pot fi descrise printr-un model comun, denumit ClasaRational. Modelul arată că orice obiect de acest fel se caracterizează printr-o pereche de numere întregi și că pe această mulțime sunt definite operații unare și binare, care arată cum “interacționează” obiectele în interiorul mulțimii: un număr rațional poate da naștere opusului și inversului său, două numere raționale pot produce un alt număr rațional ca sumă, diferență etc.

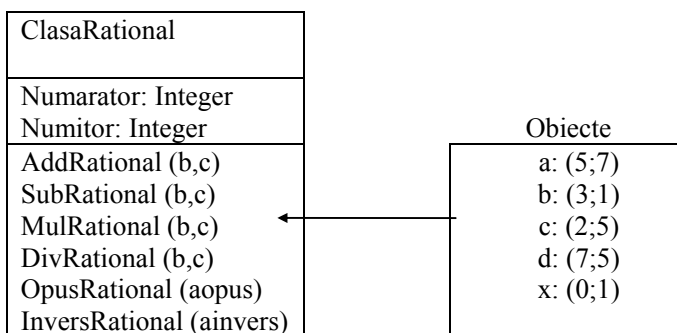


Fig. 10.1 Clasă și obiecte – mulțimea numerelor raționale

Numerele complexe (figura 10.2), descrise ca perechi de numere reale de forma (p_reală, p_imaginară) pot fi grupate într-un model comun, denumit Ccomplex:

CComplex	Exemplu de obiect
p_reală:real; p_imaginara:real;	a:(2, 7.5) a reprezintă numărul complex 2+7.5i
Conjugat(b)	
Suma(b,c)	
Inmultire(b,c)	
Modul:real;	
Diferenta(b,c)	
Impartire(b,c)	

Fig. 10.2 Clasă și obiecte – mulțimea numerelor complexe

Generalizând, se poate afirma că o clasă de obiecte se manifestă ca un tip obiect, iar modelul comun al obiectelor este modelul de definire a tipului obiect. Astfel, obiectele individuale apar ca manifestări, realizări sau instanțieri ale clasei, adică exemplare particulare generate după modelul dat de tipul obiect. Altfel spus, o clasă poate fi considerată ca un tip special de dată, iar obiectele sale ca date de acest tip.

Acceptarea acestei semnificații pentru clase de obiecte este de natură să simplifice descrierea obiectelor și să asigure un tratament al acestora similar tipurilor structurate de date din limbajele de programare: este suficientă o descriere a tipului obiect și apoi se pot declara constante și variabile de acest tip. Datele care reprezintă proprietățile obiectelor se numesc *attribute* și sunt de un anumit tip, de exemplu, întregi, real, boolean etc. Procedurile și funcțiile care definesc comportamentul obiectelor sunt cunoscute ca *metode ale clasei*. Împreună, attributele și metodele sunt *membrii clasei*, identificabili prin nume. Pentru a pune în evidență faptul că un membru aparține unui obiect se utilizează calificarea (notația cu punct): *nume_obiect.nume_membru*. În figura 10.1, **a.Numarator** referă valoarea 5, iar **a.AddRational(b,x)** referă metoda **AddRational** a obiectului *a* pentru a produce obiectul rezultat $x = a + b$.

Așa cum sugerează figura 10.1, obiectele trebuie să conțină valorile lor pentru attribute, deoarece definesc *starea obiectului* respectiv. Metodele, fiind comune, se specifică o singură dată. Despre o metodă, desemnată sau apelată cu un anumit obiect, se spune că se execută în context obiect, iar obiectul respectiv este numit curent. Apelul propriu-zis este considerat ca trimitere de mesaj la obiectul curent, iar execuția metodei reprezintă răspunsul obiectului curent la mesaj.

Faptul că o metodă se execută în contextul obiectului curent înseamnă că are, în mod implicit, acces la attributele și metodele obiectului, altele decât metoda respectivă. Pentru alte obiecte, din aceeași clasă sau din clase diferite, metoda trebuie să posede parametrii corespunzători. De asemenea, pentru a simplifica scrierea, în interiorul unei metode referirea la membrii obiectului curent se face fără calificare.

Pe baza acestor convenții, în procedurile *AddRational* și *OpusRational*, scrise în pseudocod, s-a specificat cu un parametru mai puțin decât numărul de operanzi pe care îi presupune operația respectivă, deoarece un operand este obiectul curent. Referirea la obiectul curent se distinge de celelalte prin lipsa calificării.

```

Procedure AddRational (b,c);
  c.Numărător:= Numarator * b.Numitor + Numitor * b. Numarator;
  c.Numitor:= Numitor * b.Numitor;
End;
Procedure OpusRational (aopus);
  aopus.Numarator:= -Numarator;
  aopus.Numitor:= Numitor;
End;
```

Descrierea în pseudocod a metodelor Conjugat, Suma și Modul din clasa Ccomplexe (figura 10.2) poate fi făcută astfel:

```

procedure Conjugat(b);
begin
    b.p_reala:=p_reala;
    b.p_imaginara:=-p_imaginara;
end;
procedure Suma(b,c);
begin
    c.p_reala:=p_reala+b.p_reala;
    c.p_imaginara:=-p_imaginara+b.p_imaginara;
end;
function Modul:real;
begin
    Modul:=sqrt(sqr(p_reala)+sqr(p_imaginara));
end;
    
```

Deoarece o clasă este un tip de dată, în definirea unei clase **B** se pot declara atribute de tip **A**, unde **A** este la rândul ei o clasă. Mai mult, o clasă **A** poate defini atribute de tip **A**. De exemplu, clasa **Carte** din figura 10.3 are atributul *Autor* de tipul *Persoana* care este, de asemenea, o clasă. Mai mult, *Persoana* are atributul *Sef*, care este de același tip (*Persoana*).

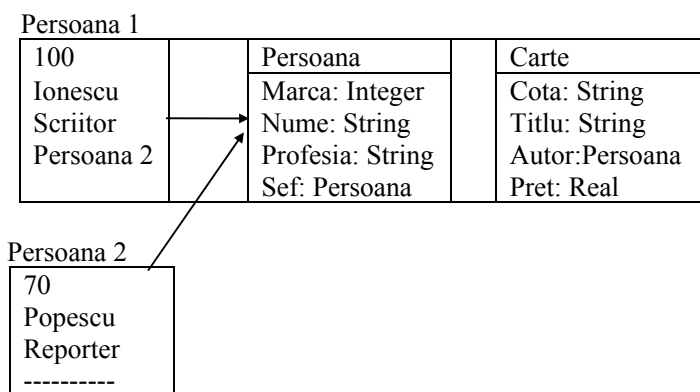


Fig. 10.3 Atribute de tip

Definirea atributelor unei clase ca tipuri ale altei clase pune în evidență o relație între clase și, deci, între obiectele acestora.

Din punct de vedere funcțional, metodele unei clase au destinații diverse. În multe cazuri și depinzând de limbaj, unei clase i se poate defini o metodă *constructor* și o metodă *destructor*. Un *constructor* este o metodă care creează un

obiect, în sensul că îi alocă spațiu și/sau inițializează atributele acestuia. *Destructorul* este o metodă care încheie ciclul de viață al unui obiect, eliberând spațiul pe care acesta l-a ocupat.

2. Încapsularea exprimă proprietatea de opacitate a obiectelor cu privire la structura lor internă și la modul de implementare a metodelor. Ea este legată de securitatea programării, furnizând un mecanism care asigură accesul controlat la starea și funcționalitatea obiectelor. Se evită astfel modificări ale atributelor obiectelor și transformări ale acestora care pot să le “deterioreze”. Potrivit acestui mecanism, o clasă trebuie să aibă membrii împărțiți în două secțiuni: partea publică și partea privată.

Partea publică este constituită din membri (atribute și metode) pe care obiectele le oferă spre utilizare altor obiecte. Ea este interfața obiectelor clasei respective cu “lumea exterioară” și depinde de proiectantul clasei. Modalitatea extremă de constituire a interfeței este aceea a unei interfețe compusă numai din metode. Dacă se dorește ca utilizatorii obiectelor clasei să poată prelua și/sau stabili valorile unor atribute ale acestora, interfața trebuie să prevadă metode speciale, numite *accesorii*.

Partea privată cuprinde membri (atribute și/sau metode) care servesc exclusiv obiectelor clasei respective. De regulă, în această parte se includ atribute și metode care facilitează implementarea interfeței.

De exemplu, o stivă, ca tip de dată, poate fi descrisă de o clasă **stack** în care interfața este constituită din metodele **Push**, **Pop**, **Top**, **Empty**, în timp ce pointerul la capul stivei, *Cap* și numărătorul de noduri, *Contor*, ca atribute, sunt “ascunse” în partea privată. Ea se servește de obiectele altei clase, denumită **Nod**, ale cărei obiecte le înlanțuiește în stivă (figura 10.4).

Trebuie remarcat că încapsularea înseamnă și faptul că utilizatorul metodelor nu trebuie să cunoască codul metodelor și nici nu trebuie să fie dependent de eventuala schimbare a acestuia, interfața fiind aceea care îi oferă funcționalitate obiectelor în condiții neschimbate de apelare.

Stack	Partea privată
Cap: Nod	
Contor: Integer	
Push ()	Interfața(partea publică)
Pop ()	
Top ()	
Empty ()	

Fig. 10.4 Interfața obiectelor

3. Moștenirea reprezintă o relație între clase și este, probabil, elementul definitoriu al **OOP**. Relația permite constituirea unei noi clase, numită *derivată*, pornind de la clase existente, denumite *de bază*. Dacă în procesul de construire participă o singură clasă de bază, moștenirea este simplă, altfel este multiplă. În

continuare se vor aborda câteva aspecte legate de moștenirea simplă, singura implementată în Pascal.

Se spune că o clasă **D** moștenește o clasă **A**, dacă obiectele din clasa **D** conțin toate atributele clasei **A** și au acces la toate metodele acestei clase. Din această definiție, dacă **D** moștenește **A**, atunci obiectele din **D** vor avea toate atributele și acces la toate metodele lui **A**, dar în plus:

- **D** poate defini noi atribute și metode;
- **D** poate redefini metode ale clasei de bază;
- metodele noi și cele redefinite au acces la toate atributele dobândite sau definite.

În figura 10.5, clasa *Cerc* moștenește clasa *Point*, deci un obiect de tipul *Cerc* va avea ca membri coordonatele *x,y* moștenite și ca atribut propriu *Raza*. Funcția *Distanța*, definită pentru calculul distanței dintre punctul curent și punctul *p*, dat ca parametru, este accesibilă și pentru obiectele *Cerc* și va calcula distanța dintre două cercuri (distanța dintre centrele lor). Funcția *Arie* și procedura *Desenează* sunt redeclerate de clasa *Cerc*, ceea ce înseamnă redefinirea lor impusă de codul diferit pe care trebuie să-l aibă în funcție de tipul figurilor geometrice (cerc sau altă figură).

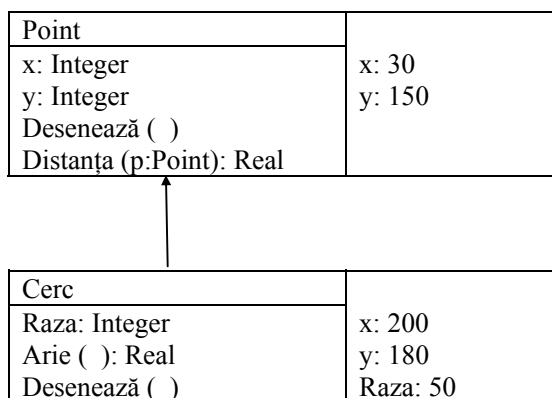


Fig. 10.5 Moștenirea simplă

Dacă se au în vedere mulțimi de clase, atunci se observă că relația de moștenire simplă induce un arbore ierarhic de moștenire pe această mulțime. Există o singură clasă inițială, și anume rădăcina arborelui, fiecare clasă are un singur ascendent (părinte) și orice clasă care nu este frunză poate avea unul sau mai mulți descendenți (fii). În fine, cu privire la moștenirea simplă se pot face următoarele observații:

- dacă se aduc modificări în clasa de bază, prin adăugarea de atribute și/sau metode, nu este necesar să se modifice și clasa derivată;
- moștenirea permite specializarea și îmbogățirea claselor, ceea ce înseamnă că, prin redefinire și adăugare de noi membri, clasa derivată are, în parte, funcționalitatea clasei de bază,

la care se adaugă elemente funcționale noi; • moștenirea este mecanismul prin care se asigură reutilizarea codului, sporind productivitatea muncii de programare.

4. Polimorfismul este un concept mai vechi al programării, cu diferite implementări în limbajele de programare care se bazează pe tipuri de date (limbaje cu tip). Ea și-a găsit extensia naturală și în *modelul orientat pe date*, implementat prin limbaje cu tip, în care clasa reprezintă tipul de date obiect.

- *Polimorfismul în limbajele de programare cu tip*. Noțiunea de polimorfism exprimă capacitatea unui limbaj de programare cu tip de a exprima comportamentul unei proceduri independent de natura (tipul) parametrilor săi. De exemplu, o procedură care determină cea mai mare valoare dintr-un șir de valori este polimorfică dacă poate fi scrisă independent de tipul acestor valori. În funcție de modul de implementare, se disting mai multe tipuri de polimorfism.

Polimorfismul ad-hoc se materializează sub forma unor proceduri care au toate același nume, dar se disting prin numărul și/sau tipul parametrilor. Polimorfismul este denumit și *supraîncărcare*, având în vedere semantica specifică fiecărei proceduri în parte.

Polimorfismul de incluziune se bazează pe o relație de ordine parțială între tipurile de date, denumită relație de incluziune sau inferioritate. Dacă un tip **A** este inclus (inferior) într-un tip **B**, atunci se poate pasa un parametru de tip **A** la o procedură care așteaptă un parametru de tip **B**. Astfel, o singură procedură definește funcțional o familie de proceduri pentru toate tipurile inferioare celor declarate ca parametri. Un exemplu clasic este cazul tipului întreg, inferior tipului real în toate operațiile de calcul.

Polimorfismul parametric constă în definirea unui model de procedură pentru care înseși tipurile sunt parametri. Polimorfismul, denumit și genericitate, presupune că procedura se generează pentru fiecare tip transmis la apel ca parametru.

Cele trei tipuri de polimorfism există (toate sau numai o parte din ele) în limbajele clasice de programare, dar unele pot să nu fie accesibile programatorului. Așa este cazul limbajului Pascal în care polimorfismul ad-hoc este implicit numai pentru operatorii limbajului (+, -, /, * etc), polimorfismul parametric nu este implementat, iar polimorfismul de incluziune este aplicabil numai funcțiilor și procedurilor de sistem.

- *Polimorfismul în limbajele orientate obiect*. Limbajele orientate obiect sau extensiile obiect ale unor limbaje cu tip oferă, în mod natural, polimorfismul ad-hoc și de incluziune. *Polimorfismul ad-hoc* intrinsec reprezintă posibilitatea de a defini în două clase independente metode cu același nume, cu parametri identici sau diferiți. Acest polimorfism nu necesită mecanisme speciale și decurge simplu, din faptul că fiecare obiect este responsabil de tratarea mesajelor pe care le primește. Polimorfismul este de aceeași natură și în cazul în care între clase există o relație de moștenire, cu precizarea că, în cazul în care o metodă din clasa derivată are parametri identici cu ai metodei cu același nume din clasa de bază, nu mai este

supraîncărcare, ci redefinire, după cum s-a precizat în paragraful anterior. *Polimorfismul de incluziune* este legat de relația de moștenire și de aceea se numește *polimorfism de moștenire*. Într-adevăr, relația de moștenire este de ordine parțială. Când clasa **D** moștenește direct sau indirect clasa **A**, atunci **D** este inferior lui **A**. În aceste condiții, orice metodă a lui **A** este aplicabilă la obiectele de clasă **D** și orice metodă, indiferent de context, care are definit un parametru de tip **A** (părinte) poate primi ca argument corespunzător (parametru actual) un obiect de clasă **D** (fiu).

Observație: un obiect de clasă **A** nu poate lua locul unui obiect de clasă **D**, deoarece **A** “acoperă” numai parțial pe **D**, care este o extensie și o specializare a lui **A**.

Limbajul Turbo Pascal, ca suport pentru programarea obiect, oferă ambele forme de polimorfism pentru clase.

- *Legarea statică și dinamică a metodelor*. Legarea statică a metodelor se regăsește atât în limbajele orientate obiect cât și în cele clasice. Compilatorul poate determina care metodă și din care clasă este efectiv apelată într-un anumit context și poate genera codul de apel corespunzător.

În plus, datorită polimorfismului și lucrului cu pointeri, în limbajele orientate obiect, unui obiect din clasa părinte, desemnat indirect prin referință (pointer) și nu prin nume, i se poate atribui un obiect fiu. În general, nu este posibil de determinat dacă, în contextul dat, metoda polimorfică trebuie apelată în varianta clasei de bază sau a celei derivate. De aceea, compilatorul generează un cod care, la momentul execuției, va testa tipul efectiv al obiectului și va realiza legarea metodei adecvate. În acest caz legarea este dinamică (sau la momentul execuției). Legarea dinamică este mai costisitoare decât cea statică, dar reprezintă o necesitate pentru a asigura elasticitatea necesară în realizarea programelor **OOP**, obiectele putând avea caracter de variabile dinamice.

10.2 Clase și obiecte în Pascal

Turbo Pascal implementează tehnica **OOP** pe fondul caracteristicilor sale de limbaj procedural, ceea ce înseamnă că lucrul cu clase și obiecte se realizează asemănător cu cel cu tipuri structurate de date, mecanismul de încapsulare fiind asigurat prin construirea unităților.

10.2.1 Specificarea claselor

În Pascal, clasele sunt tratate asemănător datelor de tip articol. Ținând seama de faptul că o clasă conține atât date cât și metode, specificarea unei clase presupune declararea structurii și definirea metodelor.

- **Declararea structurii.** Declararea unei clase, ca tip, se face în secțiunea

TYPE sub următoarea formă generală:

```
referinta_la_clasa = ^nume_clasa    {optional}
nume_clasa=OBJECT
    atribut_1;
    .....
    atribut_n;

    metoda_1;
    .....
    metoda_m;
{ PRIVATE
    atribut_1;
    .....
    atribut_p;

    metoda_1;
    .....
    metoda_q; }

END;
```

Fiecare declarație de atribut are forma: **nume_atribut:tip**, unde **tip** poate fi predefinit sau definit anterior de utilizator, inclusiv numele unui alt tip de clasă. La fel ca și la tipul RECORD, unele atribute pot fi referințe, chiar la tipul clasei care se definește, caz în care declararea tipului referință aferent trebuie să preceadă declararea clasei. O declarație de metodă cuprinde numai antetul unei funcții sau proceduri (signatura), adică numele, parametrii formali (dacă există) și tipul rezultatului, dacă este cazul. Apariția opțională a secțiunii PRIVATE în declarația de clasă anunță partea privată a clasei. Toate atributele și metodele care preced această secțiune sunt considerate publice. În fine, se remarcă prezența cuvântului OBJECT ca declarator de tip obiect. În Turbo Pascal nu există un declarator CLASS, pentru a introduce o clasă, așa cum există, de exemplu, în C++. Deci declaratorul, OBJECT, nu declară un obiect, ci o clasă de obiecte.

- **Definirea.** Definirea metodelor constă în construirea corpurilor subprogramelor a căror semnătură a fost precizată. Deoarece într-un program pot să existe mai multe clase, definirea trebuie să asigure recunoașterea metodelor diferitelor clase. Acest lucru se realizează prin prefixarea numelui metodei cu numele clasei. În rest, corpul metodelor se scrie în conformitate cu regulile generale ale limbajului și având în vedere că metoda respectivă se execută în contextul obiectului curent.

Exemplu:

10.1. Declararea și definirea unei clase

```
{ Declararea clasei Point}
Point = OBJECT
    Function Getx: Integer;
    Function Gety: Integer;
    Procedure Init (a, b: Integer);
    Function Distance (p: Point): Real;
Private
    x: Integer;
    y: Integer;
END;
{ Definirea metodelor}
Function Point.Getx: Integer;
Begin
    Getx: = x;
End;
Function Point.Gety: Integer;
Begin
    Gety: = y;
End;
Function Point.Distance (p: Point): Real;
Var
    dx, dy: Integer;
Begin
    dx: = x-p.x;
    dy: = y-p.y;
    Distance: = sqrt(dx*dx-dy*dy);
End;
Procedure Point.Init(a, b: Integer);
Begin
    x: = a;
    y: = b;
End;
```

Din exemplul, 10.1 se observă modul de declarare a unei clase, în care, pentru asigurarea încapsulării depline, comunicarea cu exteriorul se face numai prin interfață. Funcțiile *Getx* și *Gety* sunt necesare ca *accesorii* pentru a returna valorile atributelor *x* și *y*, iar procedura *Init* este un constructor care inițializează (schimbă) valorile celor două atribute. Funcția *Distance* ilustrează modul în care trebuie înțeleasă afirmația că o metodă se execută în context obiect. Se observă că, deși calculul distanței presupune două obiecte de tip *Point*, ca parametru figurează numai un punct, *p*. Unde este celălalt (primul)? În mod implicit, se consideră că primul punct este un obiect ascuns, obiectul de apel al funcției, numit curent, coordonatele sale fiind *x* și *y*. În relațiile de calcul, referirea coordonatelor celui alt punct, transmis ca argument, se va face prin notația cu calificări: **p.x**, **p.y**. Rezultă de aici că funcțiile *Getx*, *Gety* și procedura *Init* se referă la obiectul curent.

- **Unit de clasă.** Deoarece clasele trebuie să fie prin definiție entități reutilizabile, se încapsulează într-o unitate. Declararea claselor apare în secțiunea INTERFACE, iar definirea metodelor în secțiunea IMPLEMENTATION. Se recomandă ca fiecare clasă să fie o unitate.

10.2.2 Utilizarea obiectelor

Obiectele unei clase se comportă asemănător variabilelor de tip articol. Sintetizând aceasta, înseamnă că:

- Obiectele se declară, ca și variabilele, într-o secțiune VAR dintr-o entitate de program (program principal, funcție, procedură sau unitate). De aici rezultă că obiectele pot avea caracter *global* sau *local*, ceea ce determină, pe de o parte, spațiul de vizibilitate, iar, pe de altă parte, durata de viață și segmentul de alocare.

- Un obiect poate fi declarat *static* sau *dinamic*. Atributul de *static*, respectiv *dinamic*, se referă la momentul în care are loc alocarea spațiului pentru obiect (la compilare sau la execuție).

Deoarece declararea obiectelor dinamice va fi tratată într-un alt paragraf, în continuare se prezintă numai aspectele privind obiectele statice. Pentru obiectele statice declarația are forma: **nume_obiect:nume_clasa**. De exemplu: p1, p2 : Point. Ca și în cazul variabilelor, obiectele declarate trebuie să fie inițializate înainte de a fi utilizate. Se apelează, în acest sens, o metodă de tip constructor. De exemplu, p2.Init (30,100) inițializează punctul p2 cu x = 30 și y = 100.

Două obiecte de același tip pot participa într-o operație de atribuire. De exemplu, p1: = p2. Atribuirea între obiecte nu este întotdeauna corectă. Dacă obiectele au spațiu extins, atunci copierea nu este corectă. De aceea, atribuirile trebuie utilizate cu multă atenție, ele realizându-se prin copiere bit cu bit.

Un obiect poate participa ca argument, prin valoare sau referință, la apelul de procedură sau funcție. Deoarece transferul prin valoare corespunde unei copieri, pot apărea aceleași neajunsuri ca și la operația de atribuire. În plus, copierea poate fi și costisitoare ca spațiu și timp. De aceea se preferă transferul prin referință. O funcție nu poate returna ca rezultat un obiect, dar poate returna o referință la un obiect.

Orice metodă a unei clase, inclusiv constructorii, se apelează cu un anumit obiect care devine *acti* sau *curent* . Apelul are forma:

obiect.nume_metoda(alte_argumente).

De exemplu:

```
pr. Init( 70,80);  
d:= p1.Distance (p2).
```

Obiectul de apel este un parametru ascuns care se asociază automat cu un parametru formal denumit **Self**. Acest parametru formal prefixează, în mod

implicit, atributele și metodele utilizate de metoda apelată. Se poate spune că **Self** este un pseudonim al obiectului de apel și, de aceea, în metoda apelată se poate scrie: **Self.atribut** sau **Self.metoda**. Uneori **Self** se utilizează explicit pentru a evita ambiguitatea referirii sau atunci când este nevoie de a obține adresa unui obiect curent. În ultimul caz, adresa se desemnează prin **@Self**.

Exemplu:

10.2. Se declară și se definește o clasă de numere raționale ca perechi (Numarator, Numitor) de numere întregi. O astfel de clasă poate fi utilă în calcule cu fracții ordinale, atunci când transformarea acestora în fracții zecimale (numere reale) nu este de dorit din cauza erorilor de trunchiere.

```
UNIT ClasaRat;
INTERFACE
    Type
        Rational = OBJECT
            Procedure InitRat (x, y:Integer);
            Procedure AddRat (b: Rational; VAR c: Rational);
            Procedure SubRat (b: Rational; VAR c: Rational);
            Procedure MulRat (b: Rational; VAR c: Rational);
            Procedure DivRat (b: Rational; VAR c: Rational);
            Procedure OpusRat (VAR c: Rational);
            Procedure InversRat (VAR c: Rational);
            Function GetNumarator: Integer;
            Function GetNumitor: Integer;
            Procedure ReadRat;
            Procedure WriteRat;
            Function CompRat (b: Rational): Integer;
        PRIVATE
            Numarator, Numitor: Integer;
            Procedure SignRat;
        END;
IMPLEMENTATION
Function Cmmdc (x,y: Integer ):integer:FORWARD;
Procedure Rational. InitRat (x, y: Integer);
{Initializeaza in forma inductiva functia (numarator,
numitor)}
    Var d: Integer;
    Begin
        If (x=0) or (y=1)
            then y=1
            else begin
                If (abs(x)<>1) and (abs(y)<>1) then
                    begin
                        d:= Cmmdc (x,y)
                        x:= x/d;
                        y:= y/d;
                    end;
                SignRat;
            End;
        Numarator:= x;
        Numitor:= y;
    End;

Procedure Rational.AddRat (b: Rational; VAR c: Rational);
Var x, y: Integer;
Begin
    x:= Numarator*b.Numarator+Numitor*b.Numarator;
```

```
        y: = Numitor*b.Numitor;
        c.InitRat (x, y);
End;

Procedure Rational.SubRat (b:Rational; VAR c:Rational);
Var
    r: Rational;
Begin
    b.OpusRat ( r );
    AddRat (r, c);
End;

Procedure Rational.MulRat (b: Rational; VAR c: Rational);
Var x, y: Integer;
Begin
    x: = Numarator*b.Numarator;
    y: = Numitor*b.Numitor;
    c.InitRat (x, y);
End;

Procedure Rational.DivRat (b: Rational; VAR c: Rational);
Var r: Rational;
Begin
    b.InversRat ( r );
    MulRat (r, c);
End;

Procedure Rational.InversRat (VAR c: Rational);
Var d: Integer;
Begin
    d:= Numarator;
    if d=0 then d:=1;
    c.Numarator:= c.Numitor;
    c.Numitor:= d;
    c.SignRat;
End;

Procedure Rational.OpusRat (VAR c: Rational);
Begin
    c.Numarator:= - c.Numarator;
End;

Function Rational.GetNumarator: Integer;
Begin
    GetNumarator:= Numarator;
End;

Function Rational.GetNumitor: Integer;
Begin
    GetNumitor:= Numitor;
End;

Procedure Rational.SignRat;
Begin
    if (Numarator>0) and (Numitor<0) or
       (Numarator<0) and (Numitor<0)
    then Begin
        x:= -x;
        y:= -y;
    End;
End;

Function Cmmdc (x, y: Integer) : Integer;
```

```
Begin
    x:= abs (x); y:= abs (y);
    While x< >y Do
        if x>y then x:= x-y else y:= y-x;
    Cmmdc:= x;
End;

Procedure Rational.ReadRat;
Var
    txt: String [ 25];
    i, x, y: Integer;
Begin
    Read (txt);
    i:= Pos (txt, '/');
    If (i=0) then Begin {Numarator întreg}
        Val (txt, Numarator);
        Numitor:= 1;
        End
    else
        Begin
            Val(Copy (txt, 1, i-1), x);
            Val (Copy (txt, i, 25), y);
            InitRat (x, y);
        End;
    End;

Procedure Rational.WriteRat;
Begin
    Write (Numarator, '/', Numitor);
End;

Function Rational.CompRat (b: Rational): Integer;
Var d, n1, n2: Integer;
Begin
    d:= cmmdc (Numitor, b.Numitor);
    n1:= Numarator*(dDIVNumitor);
    n2:= b.Numarator*(dDIV b.Numitor);
    if (n1<n2)
        then CompRat:=-1
        else if (n1 = n2)
            then CompRat:= 0
            else CompRat:= 1;
    End;
End.
```

Clasa construită poate fi testată printr-un program multifuncțional, *TestE2*. Operațiile care se testează se codifică prin litere (**A**, **S**, **M** etc) la care se adaugă o operație, **T**, pentru terminare. Pe baza unei variabile **Op** (pentru operația curentă) se realizează o structură repetitivă, în corpul căreia se alege modul de tratare printr-o structură CASE OF.

```
Program TestE3;
Uses ClasRat;
Var
    a,b,c: Rational;
    Op: Char; r: Integer;
Procedure Meniu;
Begin
    WriteLn ('    Meniu');
    WriteLn ('A/a - Adunare');
    WriteLn ('S/s - Scadere');
```

```
        WriteLn ('M/m - Inmulțire');
        WriteLn ('D/d - Impartire');
        WriteLn ('C/c - Comparare');
        WriteLn ('G/g - Elemente');
        WriteLn ('T/t - Terminare');
        WriteLn ('    Alege operatia:');
        ReadLn ( Op );
        Op:= UpCase ( Op );
End;
BEGIN
Meniu;
While Op <>'T' Do
    Begin
        If (Op = 'G') then
            Begin
                a.ReadRat;
                WriteLn ('Numarator=', a.GetNumarator);
                WriteLn ('Numitor=', a.GetNumitor);
            End
        else If (Op = 'C') then
            Case a.CompRat (b) of
                -1: WriteLn (' a<b');
                0: WriteLn (' a=b');
                1: WriteLn ('a>b');
            End
        else
            Begin
                CaseOp of
                    'A': a.AddRat (b,c);
                    'S': a.SubRat (b,c);
                    'M': a.MulRat (b,c);
                    'D': a.DivRat (b,c);
                End;
                C.WriteRat;
            End;
        Meniu;
    End;
END.
```

La fel ca în exemplul 10.1, și în acest caz atributele sunt private, iar accesul la valorile lor se asigură prin accesoriile *GetNumarator*, *GetNumitor*. În partea privată apare o metodă care este utilizată numai intern (în clasă), în scopul asocierii semnului numărului rațional la numărător. De asemenea, se remarcă prezența funcției *Cmmdc* (cel mai mare divizor comun) care nu ține de clasă, dar este necesară implementării unor metode ale clasei.

La stabilirea numărului de parametri, în conceperea metodelor s-a avut în vedere că un operand, eventual unicul pentru operații unare, este obiectul curent. Atunci când, în contextul obiectului curent, s-a apelat o metodă care se referă la un alt obiect, acesta este precizat explicit, (altfel s-ar considera același context). De exemplu, în operațiile de scădere și împărțire se aplică o reducere la operațiile de adunare cu opusul, respectiv înmulțire cu inversul. Acest rezultat intermediar se obține prin *b.OpusRat(r)*, respectiv *b.InversRat(r)*, după care, în contextul definit la apelul lui *SubRat*, respectiv *DivRat* se invocă *AddRat(r,c)* și *MulRat(r,c)*, ceea ce

înseamnă că ele vor fi de fapt apelate ca *Self.AddRat(r,c)* și *Self.MulRat(r,c)*, adică **a** va fi prim operand.

O mențiune aparte trebuie făcută cu privire la constructorul **InitRat** care inițializează în formă ireductibilă fracția (Numarator, Numitor) a unui obiect. De aceea, ea este apelată ori de câte ori se creează o astfel de fracție, fie în programul principal, fie într-o altă metodă.

Metodele *ReadRat* și *WriteRat* realizează legătura obiectelor cu tastatura și monitorul. Procedura *ReadRat* oferă o alternativă de inițializare a obiectelor, prin citirea fracțiilor de la tastatură sub forma *x/y* sau *x*, dacă numitorul este unu (număr întreg). Procedura *WriteRat* afișează pe monitor un număr rațional sub formă de fracție *x/y*, în poziția curentă de scriere.

În cazul claselor numerice (dar nu numai), pe mulțimea obiectelor unei clase se poate defini o relație de ordine. Deoarece limbajul nu permite supraîncărcarea operatorilor, este necesar să se introducă o metodă, de regulă o funcție, care să definească relația. Este cazul funcției *CompRat* care compară două numere raționale **a**, **b** și returnează **-1**, **0**, **1**, după cum între cele două numere există una din relațiile: **a<b**; **a=b**; **a>b**.

Legat de precizările referitoare la supraîncărcarea operatorilor, se observă o lipsă de naturalețe în scrierea operațiilor definite pe mulțimea obiectelor. De exemplu, se scrie **a.AddRat(b, c)** în locul formei **c:=a+b** sau **a.CompRat (b)** în loc de **a<b** etc., așa cum ar fi posibil dacă operatorii +, < etc. ar putea fi supraîncărcați (cum este posibil în C++).

În fine, se face observația că limbajul nu prevede facilități de tratare a cazurilor de eroare care pot apărea în legătură cu obiectele unei clase. Se presupune că proiectantul clasei introduce eventuale măsuri de tratare a erorilor sau că utilizatorul clasei evită apelul unor metode în condițiile în care acestea pot produce erori. În exemplul considerat, procedurile *InitRat* și *InversRat* tratează și cazurile de eroare, prin transformarea corespunzătoare a obiectelor. În acest fel, programele nu riscă apariția erorii de împărțire la zero, dar pot furniza rezultate neașteptate. Această modalitate poate semnală apariția unei erori, dar nu este forma cea mai elaborată de semnalare și de prelucrare a erorii.

10.2.3 Constructori și destructori

De cele mai multe ori, spațiul unui obiect este compact, constituit din spațiile necesare atributelor sale. În cazul în care unele atribute sunt de tip referință, spațiul obiectelor clasei respective are o componentă dinamică denumită *spațiu extins* (figura 10.6).

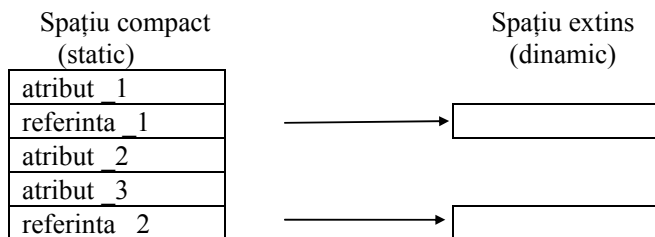


Fig. 10.6 Spațiul extins al unui obiect

Atunci când se declară o variabilă obiect dintr-o astfel de clasă, compilatorul poate alocă spațiul compact necesar. Spațiul extins, la acest moment, este necunoscut deoarece el trebuie să fie alocat în zona memoriei dinamice, la momentul execuției. Rezultă de aici două aspecte importante pe care trebuie să le aibă în vedere proiectantul clasei.

- În primul rând, trebuie să se prevadă un *constructor* capabil să trateze nu numai inițializarea atributelor statice, dar și a celor care ocupă un spațiu dinamic. Constructorul trebuie să utilizeze procedura *New* sau *GetMem* pentru a obține un bloc de mărimea necesară, a cărui adresă trebuie încărcată în câmpul referință al obiectului, iar blocul respectiv trebuie apoi inițializat.

Într-o clasă pot fi prevăzuți mai mulți constructori specializați, dacă există mai multe câmpuri de tip referință la un obiect, dacă un singur constructor ar fi incomod de utilizat sau ar avea o listă prea mare de parametri. Mai mult, limbajul prevede o posibilitate specială de a declara un constructor, utilizând declaratorul *Constructor* în loc de *Procedure*. Printr-o astfel de declarare, compilatorul are posibilitatea să distingă din mulțimea metodelor pe cele care au acest rol special, lucru necesar în cazul moștenirii, dar util și programatorului pentru a putea verifica mai ușor dacă a rezolvat corect problema inițializării obiectelor.

- În alt doilea rând, atunci când un obiect trebuie să-și încheie existența (de exemplu, la ieșirea dintr-o funcție/procedură în care a fost creat), este necesar să se prevadă o metodă *destructor* care să elibereze spațiul extins. Destructorul trebuie să apeleze procedura *Dispose* sau *FreeMem* pentru a returna acest spațiu zonei *heap*. În lipsa unui destructor rămân blocuri de memorie ocupate, fără ca ele să mai poată fi referite ulterior. Declarația destructorului se poate face prin declaratorul *Destructor* în loc de *Procedure*.

Constructorii și destructorii pot avea orice utilitate pe care o consideră programatorul, atunci când se creează sau se distrug obiecte.

Atribuirea între obiectele care au spațiu extins are unele particularități. Astfel, dacă se specifică atribuirea **a:=b**, atunci se realizează o copiere bit cu bit a obiectului **b** în **a**. După operație, **a** și **b** au același conținut în spațiul compact și, deci, referă același spațiu extins (figura 10.7).

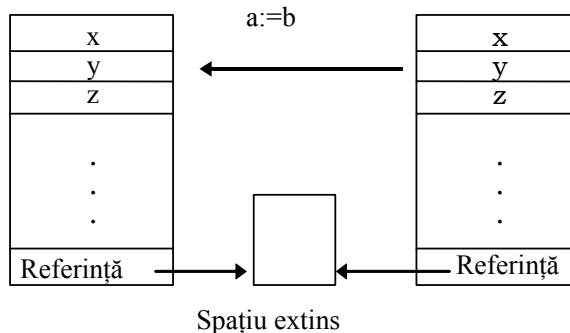


Fig. 10.7 Atribuirea în cazul obiectelor cu spațiu extins

Ulterior, modificarea realizată la unul din obiecte în spațiul extins afectează starea și pentru celălalt obiect. Dacă un obiect dispăre, atunci celălalt obiect va rămâne fără spațiu extins, iar când și acesta va dispărea, destructorul său va încerca să elibereze un spațiu care nu mai există, fiind deja eliberat. În legătură cu astfel de cazuri, deoarece operatorul de atribuire nu poate fi supraîncărcat, este necesar să se evite operația și să se prevadă o metodă proprie de copiere prin care să se inițializeze **a** cu valorile lui **b**, dar în spațiu complet separat. De asemenea, se recomandă ca obiectele de acest fel, utilizate ca argumente, să fie transmise, prin referință și nu prin valoare, având în vedere că ultima metodă este echivalentă cu o atribuire.

Exemplu:

10.3. Se ilustrează modul de utilizare a referințelor în declararea claselor, de realizare a constructorilor multipli și a destructorului de spațiu extins. Deși simplificat, exemplul sugerează modul în care se poate construi o clasă *matrice* cu elemente de tip *Rational*, prin utilizarea clasei din exemplul 10.2.

```
UNIT ClasaMRat;
INTERFACE
  uses ClasaRat;
  Const
    MaxLin = 10; MaxCol = MaxLin;
    ZeroRat : Rational = (Numarator:0; Numitor:1);
Type
  DimLin = 1..MaxLin;
  DimCol = 1..MaxCol;
  Matrice = array [DimLin, DimCol] of Rational;
  MatriceRat = OBJECT
    Constructor SetNulMat (m, n:Integer);
    Constructor ReadMat;
    Procedure AddMat (b:MatriceRat; VAR c:MatriceRat);
    Procedure WriteMat;
    Destructor FreeExtMat;
  PRIVATE
```

```

        Nlin, Ncol:Integer;
        PMat:^Matrice;
    END;
    Var
        ErrMat:Integer;
IMPLEMENTATION
Constructor MatriceRat.SetNulMat (m, n:Integer);
    Var i, j:Integer;
    Begin
        m:=Max (m, MaxLin);
        n:=Max (n, MaxCol);
        GetMem (Pmat, m*n*SizeOf (Rational));
        For i:=1 to m Do
            For j:=1 to n Do
                P.Mat^[i, j]:=ZeroRat;
            Nlin:= m; Ncol:=n;
        End;

Constructor MatriceRat.ReadMat;
Var m, n, i, j:Integer;
Begin
    Repeat
        Write ('Numărul de linii:'); ReadLn (m);
        Until m<=MaxLin;
    Repeat
        Write ('Numărul de coloane:');
        ReadLn (n);
        Until n<=MaxCol;
    WriteLn('Elementele rationale ale matricei, pe
linii!');
    GetMem (Pmat, m,*n*SizeOf (Rational));
    For i:=1 To m Do
        For j:=1 To n Do
            Begin
                Write ('x(', i:2,',', j:2,')=');
                PMat^[i, j].ReadRat;
                WriteLn;
            End;
        Nlin:=m; Ncol:=n;
    End;

Destructor MatriceRat.FreeExtMat;
Begin
    FreeMem (PMat, Nlin*Ncol*SizeOf (Rational));
End;

Procedure MatriceRat.AddMat (b:MatriceRat; VAR c:MatriceRat);
Var i, j:Integer;
Begin
    ErrMat:=0;
    If (Nlin=b.Nlin) and (Ncol=b.Ncol)
        then
            For i:=1 To Nlin Do
                For j:=1 To Ncol Do
                    PMat^[i, j].AddRat(b.PMat^[i, j],c.PMat^[i, j]);
                else ErrMat:=1;
    End;

Procedure MatriceRat.WriteMat;
Var i, j:Integer;
Begin
    For i:=1 To Nlin Do

```

```
        Begin
            WriteLn ('Linia:', i);
            For j:=1 To Ncol Do
                Begin
                    PMat^[i, j].WriteRat;
                    If (i<>Ncol) Write (';');
                End;
            End;
        End;

Program Test;
Uses ClasMat;
Var
    A,B,C:MatriceRat;
Begin
    A.ReadMat;
    B.ReadMat;
    A.AddMat (B,C);
    If (ErrMat) then      WriteLn      ('**      Eroare:Dimensiuni
diferite')
        else C.WriteMat;
    A.FreeExtMat;
    B.FreeExtMat;
    C.FreeExtMat;
    ReadLn;
END.
```

Analiza exemplului pune în evidență posibilitățile tehnicii **OOP** în limbajul Pascal.

- Inițializarea unor variabile de tip obiect, în secțiunea CONST. Este cazul variabilei *ZeroRat*, pentru care se observă tratarea inițializării ca în cazul articolelor, definindu-se variabile pentru fiecare atribut, potrivit tipului său.
- Definirea structurii de tip *array*, la care tipul de bază este un tip obiect. Tipul *Rational* permite definirea unui tip nou, *Matrice*, ca un masiv bidimensional.
- Alocarea dinamică a spațiului pentru matrice drept spațiu extins, potrivit numărului de linii și coloane, dar fără a se depăși dimensiunile declarate ale tipului *Matrice*. Se elimină, astfel, risipa de spațiu prin alocarea statică, la dimensiuni constante, pentru structura de tip masiv. De observat modul de referire a elementelor matricei și utilizarea metodelor clasei *Rational* pentru a trata aceste elemente. Se spune că între clasa *MatriceRat* și clasa *Rational* există o relație de tip client-server.
- Prezența mai multor constructori și a unui destructor pentru inițializarea, inclusiv a spațiului extins și, respectiv, eliberarea acestuia. Destructorul este apelat la sfârșitul programului, pentru fiecare obiect. Spațiul compact al obiectelor face obiectul eliberării numai în cazul obiectelor locale, la ieșirea din procedura sau funcția în care obiectele au fost create, când este automat returnat stivei.
- Tratarea erorilor clasei prin intermediul unei variabile publice (din interfață). Aceasta este utilizată în cazul operației de adunare și returnează valoarea unu dacă matricele nu au aceeași dimensiune, respectiv zero, în caz contrar. Este sarcina programului care apelează metodele clasei să verifice, la reîntoarcerea

controlului, ce valoare are variabila publică. Variabila nu este legată de clasă, ci de unitatea în care este încorporată clasa.

10.2.4 Utilizarea obiectelor dinamice

Obiectele statice nu sunt, în general, reprezentative pentru tehnica **OOP**. Cel mai frecvent, obiectele au un comportament dinamic, adică în timp se nasc, trăiesc și dispar. În mod corespunzător, trebuie să existe posibilitatea gestionării dinamice a spațiului lor, astfel încât, cu aceleași rezerve de memorie, să poată fi utilizate obiecte multiple, eventual de tipuri diferite. În limbajele orientate obiect, obiectele sunt, prin definiție, dinamice și sistemul preia sarcina de a aloca și elibera spațiul de memorie pentru acestea. Tehnica **OOP** implementată în Pascal face uz de facilitățile de definire a variabilelor dinamice, dar gestiunea obiectelor este lăsată exclusiv în sarcina programatorului.

În aceste condiții, declararea obiectelor dinamice se poate face în una din formele:

```
nume_referinta_clasa : ^nume_clasa;  
nume_referinta_clasa : nume_tip_referinta;
```

Prima formă corespunde declarării cu tip anonim a variabilelor și este mai puțin indicată. A doua formă utilizează explicit un tip referință spre clasă și este posibilă numai dacă un astfel de tip a fost declarat. De regulă, pentru a da posibilitatea definirii obiectelor dinamice, orice clasă se declară în forma:

```
nume_tip-referinta : ^nume_clasa;  
nume_clasa =OBJECT.....END;
```

Aici se face uz de excepția introdusă de limbaj, de a declara o referință înainte de declararea tipului de date pe care îl referă. Această permisiune asigură referințe în interiorul obiectelor, la obiecte din propria clasă și posibilitatea practică de a construi liste înlănțuite de variabile și obiecte.

De exemplu, dacă se presupun declarații de tip de forma:

```
PPoint=^TPoint;  
TPoint=OBJECT.....END;  
PRational=^TRational;  
TRational=OBJECT....END;
```

atunci este posibil să se declare variabile obiect de forma:

```
PtrP1,PtrP2 : Ppoint;  
PtrRat1: PRational;
```

După o astfel de declarare, compilatorul alocă spațiu (static) pentru variabilele referință. Este sarcina programatorului să aloce spațiul dinamic și să apeleze constructorii/ destructorii de inițializare/eliberare. În acest scop se utilizează procedurile *New* și *Dispose*, care au fost extinse astfel încât să accepte drept parametru și apelul la un constructor sau destructor. Mai mult, procedurile posedă o replică sub formă de funcție. Noua sintaxă a acestora se prezintă astfel:

```
New(variabila_referinta);  
New(variabila_referinta, apel_constructor);  
Variabila_referinta := New(tip_referinta, apel_constructor);  
Dispose(variabila_referinta)  
Dispose(variabila_referinta, apel_constructor);
```

De exemplu, dacă *Init(x,y:Integer)* este un constructor pentru cazul clasei *Tpoint*, se poate scrie:

```
New (PtrP1); PtrP1^.Init (30,20);  
sau  
New (PtrP1, Init(30, 20));  
sau  
PtrP1:=New (Ppoint, Init (30, 20));
```

În acest mod se asigură o posibilitate în plus pentru evitarea omisiunii apelului constructorului.

Similar stau lucrurile și cu eliberarea spațiului. Dacă obiectele dinamice nu au spațiu extins, atunci eliberarea spațiului compact dinamic se va face prin apelul procedurii *Dispose*, în prima formă; de exemplu, *Dispose (PtrP1)*. Dacă obiectul dinamic are spațiu extins, atunci clasa prevede și un destructor pentru acesta (destructor de spațiu extins). În acest caz se poate utiliza procedura *Dispose* în a doua formă. De exemplu, dacă se presupune clasa *MatriceRat* (exemplul_3), declarată în forma: *MatriceRat=^TMatriceRat; TMatriceRat= OBJECT...End*; și declarații de obiecte de forma: *PtrA:TMatriceRat*, se poate apela destructorul *FreeExtMat* în forma: *Dispose (PtrA, FreeExtMat)*. Prin acest apel se execută mai întâi destructorul, deci are loc eliberarea spațiului extins și apoi se eliberează spațiul compact dinamic.

Referirea unui obiect dinamic are forma **referinta^** și accesul la membrii obiectelor va avea formele:

```
referinta^.nume_metoda(argumente)  
referinta^.atribut
```

Așa cum se pot construi masive de obiecte statice, tot așa se pot construi astfel de structuri având ca tip de bază referințe la obiecte dinamice. Accesul la membrii obiectului **k** se va face sub forma:

```
numearray[k]^.nume_metoda(argumente);  
numearray[k]^.atribut.
```

10.3 Moștenirea în Pascal

Moștenirea este o relație de descendență ierarhică între clase, care oferă mecanismele necesare pentru ca o clasă derivată să dobândească atributele

părinților și să primească dreptul de acces la metodele acestora. Deoarece permite reutilizarea de cod, moștenirea este aspectul definitoriu al metodei **OOP**.

10.3.1 Aspecte de bază

În limbajul Pascal, moștenirea este implementată sub forma simplă, ceea ce înseamnă că o clasă nouă poate fi derivată dintr-o singură clasă părinte (de bază). Pentru ca o clasă să fie recunoscută ca bază a unei clase noi, declarația acesteia trebuie să aibă forma:

```
nume_clasa_derivata = OBJECT(nume_clasa_baza)
                        declaratii membri;
```

End;

În Pascal, clasa derivată poate să definească membri noi sau să utilizeze supraîncărcarea unor metode ale părintelui. Dacă se are în vedere arborele de moștenire, clasa derivată poate supraîncărca metode ale oricărui ascendent. Totuși, supraîncărcarea în Pascal are unele limite față de alte implementări. Operatorii limbajului nu pot fi supraîncărcați, iar o metodă nu poate fi supraîncărcată de mai multe ori în cadrul aceleiași clase. Din punct de vedere sintactic și semantic, aceste limitări constituie neajunsuri, fiind necesară introducerea unor metode distincte pentru operații formal identice, dar care acționează asupra unor obiecte de tipuri diferite. Dacă este necesar ca într-o clasă derivată, care supraîncarcă o metodă a unui strămoș, să fie referită explicit această metodă, atunci, sintactic, numele clasei strămoș trebuie să prefixeze numele metodei:

```
nume_clasa_stramos.nume_metoda_supraincercata( )
```

Dacă într-un arbore de moștenire o clasă nu posedă o anumită metodă, dar o referă fără a preciza clasa strămoș căreia îi aparține, compilatorul caută ascendent până la prima apariție a ei. Limbajul prevede, de asemenea, o regulă adecvată pentru asigurarea inițializării corecte a obiectelor aparținând unui arbore de derivare. Se cere ca fiecare clasă să aibă un constructor care să apeleze constructorul clasei părinte pentru partea moștenită și apoi să inițializeze atributele proprii (figura 10.9).

O situație similară, dar reciprocă, este cea a destructorilor. Limbajul prevede că, în cazul obiectelor unei ierarhii, apelul ascendent al destructorilor este automat. Deci, dacă o clasă (finală) posedă un destructor pentru spațiul extins, atunci execuția acestuia declanșează procesul de eliberare, din aproape în aproape, a spațiilor extinse moștenite de la toți strămoșii, prin execuția destructorilor acestora. Pentru ca mecanismul să funcționeze corect și în cazul în care există clase strămoș care nu posedă spațiu extins, limbajul permite destructori cu corp vid, pe care compilatorul îi generează automat la clasele respective. Pentru mai multă claritate, este recomandabil ca programatorul să definească destructori cu corp vid pe ramurile care nu conțin cel puțin un destructor propriu-zis.

O mențiune aparte trebuie făcută pentru cazul obiectelor dinamice ale unei ierarhii de clase. Datorită polimorfismului, orice obiect este compatibil, la atribuire, cu oricare obiect al unui strămoș. În aceste condiții, o referință la un obiect strămoș poate conține adresa unui obiect descendent și apelul unei metode, de forma: *PtrStramos^.Numemetoda ()*, introducând un element de nedeterminare în privința obiectului de apel. Pentru astfel de cazuri, în limbajul Pascal s-a stabilit următoarea regulă care se aplică în lipsa altor informații adiționale: se apelează metoda clasei corespunzătoare tipului referinței, dacă o astfel de metodă există, sau metoda corespunzătoare primului ascendent care o posedă.

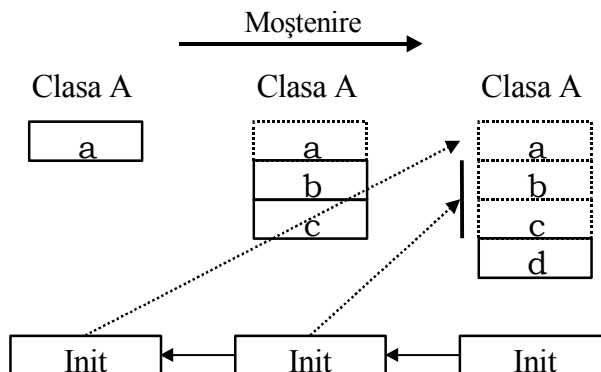


Fig. 10.9 Moștenirea și apelul

Exemplu:

10.4. În ramura $TX \leftarrow TY \leftarrow TZ \leftarrow TW$, definită în continuare, apare o procedură supraîncărcată, **p**.

```
PX=^TX;
TX=OBJECT
    Procedure p
END;
PY=^TY;
TY=OBJECT (TX)
    Procedure p
END;
PZ=^TZ;
TZ=OBJECT (TY)
    Procedure q
END;
PW=^TW;
TW=OBJECT (TZ)
    Procedure r
END;
```

Dacă se declară obiectele dinamice *PtrX:PX*; *PtrW:PW* și se face atribuirea și apelul:


```
.....
PtrX:=PtrW;
PtrX^.p;
```

se va apela procedura **p** a clasei TX, ghidându-se după referința PtrX care este asociată acestei clase. Dacă se declară PtrZ:PZ; PtrW:PW și se fac atribuirea și apelul:

```
.....
PtrZ:=PtrW;
PtrZ^.p;
```

atunci se va apela metoda **p** a clasei TY, primul ascendent al clasei TZ, deținătoarea referinței PtrZ, care posedă această metodă.

Ipostazele incluse în compilator, după regula de mai sus, permit *legarea statică* a metodelor, care este simplă și eficientă, dar care poate să nu fie convenabilă în multe cazuri.

10.3.2 Legarea dinamică. Metode virtuale

Legarea dinamică se referă la metode supraîncărcate, identice ca prototip, care sunt declarate într-o ramură a unei ierarhii de clase. În acest caz particular de supraîncărcare, se spune că metoda este *redefinită*, având în vedere funcționalitatea nouă (corpul nou) care i se asociază. Este de dorit ca metoda să fie apelată cu obiectul a cărui adresă o conține referința de apel, fapt ce nu poate fi stabilit decât la momentul execuției. Devine evidentă necesitatea înștiințării compilatorului asupra acestei intenții, pentru a nu aplica regula legării statice și, în consecință, pentru a genera codul necesar determinării obiectului real și metodei de apelat la momentul execuției. O astfel de tehnică de legare a unei metode cu obiectul dinamic este denumită dinamică (*late binding*).

Declararea unei metode din această categorie se realizează prin declaratorul VIRTUAL, inclus în linia de definire a prototipului:

procedure nume (parametri); VIRTUAL;
function nume (parametri): tip_rezultat; VIRTUAL;

Metodele redefinite sunt denumite *virtuale*. Dacă o metodă a fost declarată virtuală într-o clasă, atunci trebuie să fie declarată la fel în toate clasele derivate care au această clasă la rădăcină și redefinesc metoda în cauză. Totuși, un descendent nu este obligat să redefinească o metodă virtuală a unui ascendent, subînțelegându-se, în acest caz, că se aplică regula căutării ascendente, dacă un obiect al său o referă.

O mențiune aparte trebuie făcută cu privire la constructori și destructori. În primul rând, trebuie remarcat faptul că obiectele care posedă metode virtuale trebuie să fie inițializate prin constructor, dar constructorii nu pot fi declarați ca metode virtuale. Regula este de natură să asigure o corectă inițializare, prin evitarea situațiilor necontrolabile de inițializare parțială, posibile în cazul legării dinamice. În al doilea rând, se menționează, ca regulă, posibilitatea de a virtualiza destructorii. Spre deosebire de constructori, virtualizarea destructorilor este, în

multe cazuri, absolut necesară, deoarece se recomandă ca procedeu general. Ce se poate întâmpla în cazul în care destructorii nu sunt virtualizați rezultă și din exemplul care urmează.

Exemplu:

10.5. Se presupune derivarea $TA \leftarrow TB \leftarrow TC$ și destructorul virtual **D**. Se declară un masiv de referințe la obiecte de clasă TA care se inițializează cu adresele unor obiecte. Se execută apoi ștergerea obiectelor dinamice, cu apelul destructorului **D**, pentru a elibera spațiul extins al acestor obiecte.

```

PA = ^ TA;
TA = OBJECT
.....
Destructor D ; VIRTUAL;
End;
PB = ^ TB;
TB = OBJECT (TA)
.....
Destructor D ; VIRTUAL;
End;
PC = ^TC;
TC = OBJECT (TB)
.....
Destructor D; VIRTUAL;
End;
p:array [0..2] of PA;
.....
..
p[0]: = new (PB,...); {contine obiect TB}
p[1]: = new (PC,...); {contine obiect TC}
p[2]: = new (PA,...); {contine obiect TA}
.....
dispose (p[0], D); {apel destructor TB, TC}
dispose (p[1], D); {apel destructor TC, TB, TA}
dispose (p[2], D); {apel destructor TA}

```

Se observă că **p** conține referințe la diferite obiecte, în virtutea polimorfismului. Ștergerea obiectelor ale căror adrese sunt în componentele lui **p** implică o legare dinamică a destructorului potrivit. Astfel, pe baza regulii de apel automat al destructorilor în sens ascendent, se eliberează corect, în fiecare caz, spațiul extins propriu și cel moștenit. Dacă **D** nu ar fi fost declarat virtual, deoarece **p** are tipul de bază corespunzător referinței la clasa TA, în fiecare caz s-ar fi apelat numai destructorul acestei clase, ceea ce ar fi însemnat neeliberarea întregului spațiu extins al obiectelor respective.

Mecanismul metodelor virtuale este implementat prin intermediul tabelii de metode virtuale VMT (**V**irtual **M**ethods **T**able) a clasei care se atașează părții de date a obiectelor din acea clasă (figura 10.10).

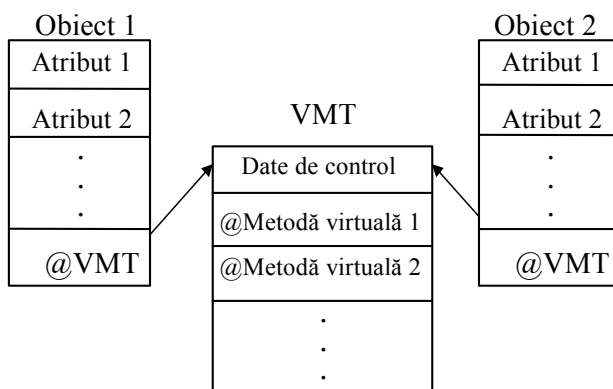


Fig. 10.10 Tabela de metode virtuale

Tabela de metode virtuale este atașată la obiect de unul din constructori, care este completat corespunzător de către compilator și este apelat înainte de apelul oricărei metode virtuale a obiectului respectiv.

10.3.3 Moștenirea și instanțele

Un tip obiect descendent dintr-un altul moștenește, în particular, atributele ascendentului. El posedă, deci, cel puțin câmpurile ascendentului, plus, eventual, altele noi. O instanță a unui tip ascendent poate fi, deci, atribuită cu o instanță a descendentului (*obiect_părinte:=obiect_fiu*). O eventuală atribuire inversă este incorectă, pentru că ar avea ca efect neinițializarea atributelor noi ale descendentului. De asemenea, în cazul unui transfer prin parametri valoare, o instanță părinte poate fi substituită cu una fiu.

Dacă, în schimb, obiectele conțin spațiu extins și unuia dintre obiecte îi este disponibilizat spațiul extins, atunci și celălalt își va pierde spațiul extins și, dacă el își va apela destructorul, acesta va încerca eliberarea unui spațiu de memorie inexistent. De asemenea, dacă tipurile de obiecte conțin metode virtuale, o atribuire nu este în general suficientă în inițializarea corectă a unei instanțe.

O instrucțiune de atribuire utilizează dimensiunea unei instanțe pentru transferul de date. În cazul claselor ce conțin metode virtuale, dimensiunea este conținută într-un câmp al VMT, deci poate fi cunoscută numai după apelul constructorului respectiv. Efectul unei instrucțiuni de atribuire este, deci, consistent doar în funcție de dimensiuni, acestea putând fi determinate numai după apelul constructorului.

Exemplu:

10.6. Lucrul cu metode virtuale

```
program virt;
uses crt;
type
  parinte=object
    a,b:word;
    constructor init_p(a1,b1:word);
    function suma:word;virtual;
  end;

  fiu=object(parinte)
    c:word;
    constructor init_f(a1,b1,c1:word);
    function suma:word;virtual;
  end;
var
  p:parinte;
  f:fiu;
constructor parinte.init_p;
begin
  a:=a1;  b:=b1;
end;
function parinte.suma:word;
begin
  suma:=a+b;
end;
constructor fiu.init_f;
begin
  a:=a1;  b:=b1;  c:=c1;
end;
function fiu.suma:word;
begin
  suma:=a+b+c;
end;
procedure afis;
begin
  writeln('Instanta parinte      a=',p.a,' b=',p.b, ' dimensiunea
',sizeof(p));
  writeln('Instanta fiu        a=',f.a,' b=',f.b,' c=',f.c,'
dimensiunea ', sizeof(f));
  readln;
end;

begin
  clrscr;
  fillchar(p,sizeof(parinte)+sizeof(fiu),#0);
  afis; {1}
  f.init_f(1,2,3);
  afis;{2}
  p:=f;
  afis;{3}
  p.init_p(4,5);
  afis;{4}
  p:=f;
  afis;{5}
end.
```

Rezultatul execuției programului

{1} parinte fiu	a=0 a=0	b=0 b=0	c=0	dimensiune 0 dimensiune 0
{2} parinte fiu	a=0 a=1	b=0 b=2	c=3	dimensiune 0 dimensiune 8
{3} parinte fiu	a=0 a=1	b=0 b=2	c=3	dimensiune 0 dimensiune 8
{4} parinte fiu	a=4 a=1	b=5 b=2	c=3	dimensiune 6 dimensiune 8
{5} parinte fiu	a=1 a=1	b=2 b=2	c=3	dimensiune 6 dimensiune 8

10.4 Proiectarea ierarhiilor de clase

Tehnica **OOP** se bazează pe date și pe clasificarea corespunzătoare a acestora. De aceea, pentru a asigura reutilizarea de cod, obiectivul principal al proiectării este construirea unui arbore de moștenire cât mai judicios. În continuare se vor face câteva recomandări de proiectare, ilustrate pe baza unui exemplu didactic de obiecte grafice, redat în detaliu la sfârșitul capitolului.

10.4.1 Clase abstracte

Dacă în procesul de analiză se privilegiază datele, atunci primul lucru care trebuie pus în evidență sunt mulțimile de obiecte care trebuie tratate. Pentru cazul obiectelor vizuale, acestea sunt: puncte, linii, dreptunghiuri, cercuri etc. În continuare trebuie căutate acele date (caracteristici) care sunt definitorii pentru diferitele tipuri de obiecte și trebuie puse în evidență acele elemente care sunt comune. Datele comune se constituie în clase și reprezintă acele date care pot fi moștenite. În cazul figurilor geometrice, se poate constata că orice obiect se identifică printr-un punct de coordonate (x,y) reprezentând începutul liniei, colțul stânga sus al dreptunghiului, centrul cercului etc, că orice figură presupune o curbă de contur trasată cu o anumită culoare, că la figurile închise se pune în evidență o suprafață care poate fi umplută (hașurată și colorată) într-un anumit mod etc.

Se poate ușor constata că, aplicând principiul grupării datelor comune în clase, se ajunge în situația de a obține clase pentru care nu are sens să se declare obiecte. Astfel, de exemplu, ar fi clasa care descrie curba de contur a obiectelor (*Contur*) sau cea care conține elementele definitorii pentru suprafața interioară a obiectelor (*Suprafața*). Astfel de clase se numesc abstracte (figura 10.11). Ele nu pot fi instanțiate, adică nu pot genera obiecte de acest tip. Rolul lor în ierarhii este

acela de clase de bază care grupează date și metode comune. Din ele, prin specializare, se pot construi clase noi, care vor poseda caracteristicile clasei pe care o moștenesc și, în plus, vor defini elemente (membri) specifice. Așa de exemplu, clasa *Contur* poate da naștere unei clase *Linie*, de obiecte de tip linie dreaptă, iar clasa *Suprafața* conduce la clasele *Dreptunghi* și *Cerc*, ca obiecte cu suprafață specifică.

Clasele abstracte pot fi definite la diferite niveluri în arbore, fiind posibil ca o clasă abstractă să fie părinte pentru o altă clasă abstractă mai specializată. De regulă, o clasă abstractă este rădăcină în orice ierarhie. Deși clasele abstracte nu se instanțiază, este permis să se declare referințe către acestea. Acest lucru asigură polimorfismul și, la limită, o referință de clasă rădăcină poate recepționa adresa oricărui obiect din ierarhie și, în consecință, poate asigura tratarea dinamică a obiectelor polimorfe.

Uneori, clasele abstracte sunt utilizate pentru a grupa clase care nu au date comune, dar participă împreună la realizarea programelor. Aceste clase sunt atât de generale încât, de regulă, se reduc la un constructor vid și, eventual, la un destructor vid și virtual. Ele se justifică prin aceea că, în virtutea polimorfismului, permit aplicarea unui cod comun și general la obiectele acestor clase, cum ar fi constituirea de liste eterogene.

Definirea unei clase abstracte drept clasă de bază a unei ierarhii conduce la posibilitatea de a utiliza aceeași clasă pentru toate ierarhiile. Principiul este aplicat, în general, la construirea bibliotecilor de clase pe care limbajele orientate obiect le oferă. De exemplu, în Turbo Pascal, biblioteca denumită Turbo Vision are drept clasă primordială pentru toate ierarhiile clasa *TObject*. Astfel se “leagă” între ele diferitele ierarhii și se poate utiliza facilitatea de polimorfism.

10.4.2 Definirea metodelor

După conturarea claselor, din punctul de vedere al membrilor de tip dată, următoarea etapă a proiectării este definirea metodelor, ținând cont de comportamentul dorit al diferitelor tipuri de obiecte. În cazul considerat, obiectele se creează, se afișează, se ascund, își schimbă culoarea de contur ori modul de umplere etc. (figura 10.11).

În legătură cu definirea metodelor se fac următoarele precizări:

- Fiecare clasă trebuie să posede un constructor. După modelul Turbo Vision, se recomandă a fi denumit **INIT**. Constructorul unei clase trebuie să apeleze constructorul clasei părinte, pentru a se asigura inițializarea părții moștenite.
- Fiecare clasă care utilizează alocarea dinamică pentru spațiu extins trebuie să posede un destructor, denumit **DONE**, declarat virtual.
- Vor fi declarate virtuale, la nivelul claselor strămoș, toate metodele care urmează să fie redefinite în clasele derivate. Aceste metode trebuie să fie redeclarate ca virtuale în toți descendenții.
- Dacă o clasă instanțiabilă conduce la obiecte vizuale (grafice, texte), atunci clasa trebuie să posede o metodă proprie de afișare (se spune că obiectele se

autoafișează). Metoda de afișare se recomandă să fie definită ca virtuală, pentru a se asigura o legare dinamică adecvată.

- Pentru a asigura satisfacerea deplină a principiului abstractizării și încapsulării datelor, se recomandă utilizarea secțiunii **PRIVATE** pentru datele specifice clasei și care nu intră direct în interfață. În acest sens trebuie prevăzute, pentru fiecare câmp, o metodă de consultare și o metodă de modificare a valorii.

- Deoarece programele realizate prin tehnica **OOP** sunt de talie mare, se recomandă utilizarea memoriei dinamice, care trebuie eliberată sistematic, eventual prin destructori adecvați.

10.4.3 Tratarea erorilor

În realizarea metodelor unei clase apar, de regulă, două tipuri de erori: de domeniu și de alocare.

- *Erorile de domeniu* sunt generate de nerespectarea domeniului de definire a metodei (de exemplu, pentru o clasă stivă, operațiile de citire și ștergere nu sunt definite în cazul în care lista este vidă). Erorile nu sunt tratate de limbaj, fiind sarcina programatorului să prevadă mecanisme adecvate de comunicare între metodele claselor și utilizatorul acestora. După “agentul” care provoacă situațiile de eroare, se pot aplica, în general, două strategii:

- a) Dacă eroarea este determinată de starea obiectului curent, adică de obiectul cu care urmează să se apeleze o anumită metodă, atunci este detectabilă înainte de apel. Clasa prevede metode de detecție, ca funcții booleene, pe fiecare tip de eroare sau o funcție unică pentru a sesiza prezența erorii. Obiectul curent va utiliza o funcție de detecție adecvată și va proceda în consecință, adică va apela sau nu metoda respectivă. De exemplu, pentru un obiect de tip listă, o funcție *Empty* poate testa dacă pointerul pentru capul listei are valoarea NIL, întorcând valoarea *True*, dacă lista este vidă sau *False*, în caz contrar.

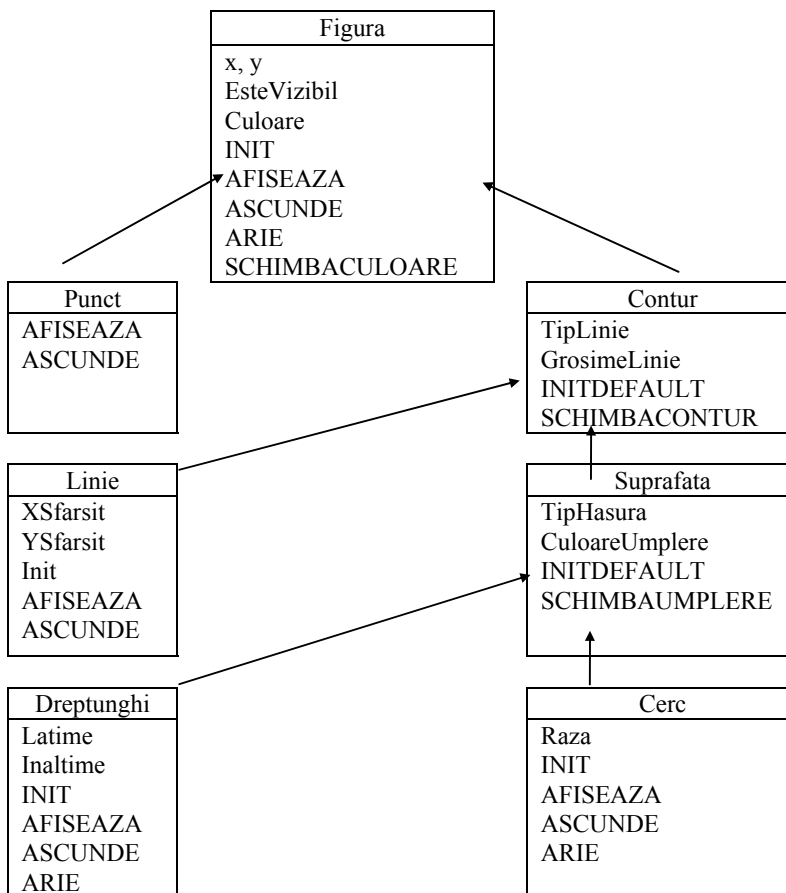


Fig. 10.11 Ierarhie de obiecte grafice

b) Dacă situațiile de eroare sunt datorate stării altor obiecte, care se primesc ca parametri în metoda aplicată, atunci strategia anterioară nu este aplicabilă, deoarece numai metoda în cauză este în măsură să sesizeze astfel de cazuri prin testarea parametrilor săi. În consecință, modul în care s-a terminat execuția metodei se transmite ca o informație apelatorului. Ea trebuie tratată ca o informație globală metodei sau chiar clasei, deoarece este independentă de obiectul cu care s-a apelat metoda. Având în vedere încapsularea claselor în unități de program, astfel de informații pot fi valori ale unor variabile globale, declarate în secțiunea INTERFACE. Variabila globală poate fi de tip enumerativ, tip în care constantele definesc situațiile de eroare. O declarație pentru aceste elemente poate avea, de exemplu, forma:


```

INTERFACE
  Type
    TipError=(OK, ZeroNum, TooBigNum);
  Var
    classErr:TipError;

```

La începutul metodelor implicate, variabilele de eroare se pun pe constanta care semnifică succes și, după testările parametrilor, se modifică adecvat. Este sarcina apelatorului metodei ca, la reîntoarcerea controlului din metodă, să testeze dacă execuția s-a realizat cu succes.

- *Erori de alocare.* Alocarea spațiului dinamic pentru obiecte sau a spațiului extins al acestora poate eșua. O astfel de situație conduce la abandonarea programului. Programatorul trebuie să construiască un mecanism propriu de tratare a situației. El se bazează pe utilizarea variabilei *HeapError*, procedura *Fail* și posibilitatea de a inițializa o unitate. Se procedează astfel:

◊ În fiecare constructor sau metodă care utilizează alocarea dinamică trebuie să se testeze valoarea referinței returnată de funcția sau procedura de alocare. Dacă a apărut o situație de eroare de alocare, procedura poate prevedea cod pentru a recupera eroarea, dacă acest lucru este posibil sau poate să încheie în ordine execuția apelând procedura *Fail*. De exemplu:

```

Constructor TA.Init;
Begin

    .....
    new (Ptr1); {alocare de spatiu dinamic}
    new(Ptr2);

    .....
    if (Ptr1=Nil) or (Ptr2=Nil).....
        Then
            begin
                {cod pentru recuperare}
                sau
                {cod de terminare + TA.DONE + Fail;}
            end;
        {continuare constructor}
End;

```

◊ În partea de implementare a unității de program a clasei se definește o funcție proprie, având un argument de tip WORD, care este utilizată în locul funcției standard ce se apelează în caz de lipsă de spațiu:

```

{$F+}           {adresare far}
Function fallocerr (a:Word): Integer;
Begin
    fallocerr:=1
End;

```

◊ În partea de inițializare a unității de program se introduce secvența:

```
Begin  
  HeapError:=@fallocerr;  
End.
```

În acest mod se înlocuiește funcția spre care punctează variabila *HeapError*, care întoarce valoarea zero în caz de lipsă de spațiu, cu o funcție proprie care returnează unu. Aceasta face ca sistemul să continue execuția programului și în cazul de eroare de spațiu, făcând posibilă tratarea proprie a situației.