Capitolul 13. Modelul Relaţional-Obiectual în Oracle9i2

Când vorbim de relaţional versus obiectual în bazele de date facem trimitere, vrând-nevrând, la cel mai surd război al ultimelor două decenii în această zonă a informaticii. Pe la mijlocul deceniului opt se părea că în circa un deceniu obiectualul, în calitate de element nou, progresist, va îngenunchia modelul relaţional. Dincolo de răceala relaţională în a gestiona obiecte complexe (altele decât simple numere, şiruri de caractere şi date calendaristice), cel mai bun argument era palmaresul orientării pe obiecte ce devenea, încet-încet, tehnologia cheie în limbajele/mediile de programare şi pătrundea tot mai temeinic în tot ceea ce ţine de analiza şi proiectarea de sisteme informatice.

Ei bine, la nivelul anului 2003, chiar dacă orientarea pe obiecte reprezintă ingredientul fundamental al mediilor de programare şi al metodelor, metodologiilor şi instrumentelor de analiză şi proiectare, în lumea bazelor de date relaţionalul continuă să ridiculeze obiectualul, dacă e să vorbim de segmentul de piaţă deţinut. Deşi, SGBD-urile "curat" obiectuale sunt încă în căutarea consacrării, marii producători de instrumente pentru aplicaţii cu baze de date au încercat un mariaj între relaţional şi obiectual, folosind obiectualul pentru partea de modelare/proiectare şi păstrând relaţionalul pentru stocarea datelor. Din acest punct de vedere, Oracle este unul dintre cele mai generoase produse.

Oracle oferă în prezent suport deplin pentru tehnologia orientată-obiect sub forma unui nivel de abstractizare plasat deasupra modelului relaţional. Tipuri abstracte noi (numite şi tipuri definite de utilizator) pot fi declarate fie pornind de la structurile de date predefinite, fie pe baza unor alte tipuri abstracte, referinţe spre obiecte sau colecţii. Metadatele referitoare la tipurile abstracte sunt păstrate în schema bazei, fiind astfel disponibile în SQL, PL/SQL, Java şi alte limbaje sau aplicaţii client. În spatele acestui model obiectual datele sunt în continuare stocate sub formă de tabele şi atribute. Utilizarea tipurilor abstracte în modelarea datelor poate oferi următoarele avantaje majore:

- Pot fi încapsulate operaţii. Dacă tabelele pot stoca doar date, obiectele oferă posibilitatea definirii unor funcţii (denumite metode) aplicabile asupra datelor acelor obiecte. Spre exemplu, pentru o factură, privită ca obiect ce cuprinde toate datele unui document real, se poate defini o metodă care să calculeze suma totală şi TVA-ul aferent.
- Obiectele oferă eficienţă sporită în dezvoltarea aplicaţiilor datorită faptului că pot fi „memorate" în baza de date, includ operaţiile ce se pot efectua asupra lor şi pot fi accesate de orice alt modul-program-client. Ca urmare, programatorii nu sunt nevoiţi să reconstruiască structuri proprii pentru manipularea datelor în fiecare aplicaţie-client.

- Obiectele oferă o perspectivă de "întreg" asupra datelor (părţile componente), în mod similar modului uman de abordare a lucrurilor. Ca urmare, sunt mai uşor de reprezentat şi de gestionat.

În cele ce urmează vom prezenta mecanismul de modelare orientată-obiect a datelor oferit de Oracle9i2.

13.1. Tipuri abstracte de date. Atribute şi metode

În Oracle, pentru a defini un nou tip utilizăm instrucţiunea `CREATE TYPE` astfel:

```
CREATE TYPE numeTip AS OBJECT (
   <atribut_1 Tip>, …, <atribut_n Tip>,
    MEMBER FUNCTION Metoda1 [(p1 tip,…)] RETURN Tip],
   [ MEMBER PROCEDURE Metoda2 [(p1 tip, …)]
                                )
```

Fiecare membru-atribut poate îmbrăca forma unui tip scalar sau colecţie predefinit (`NUMBER`, `VARCHAR2`, `INTEGER` etc.) sau poate fi declarat ca aparţinând unui tip abstract definit anterior. Definirea comportamentului obiectelor (operaţiile pe care le pot realiza acestea asupra datelor proprii) se realizează prin implementarea funcţiilor-membru specificate la crearea tipului, astfel:

```
CREATE TYPE BODY numeTip AS
   MEMBER FUNCTION Metoda1 IS
   … <variabile locale>…
   BEGIN
   … <Corpul metodei>…
   END Metoda1;
   …
END;
```

Spre exemplu, pentru a gestiona angajaţii unei unităţi se poate crea tipul `Angajat`, care să definească, în plus faţă de atributele specifice, două metode: `calculSalariu()` pentru operaţii asupra datelor stocate la un moment dat de un obiect de tip `Angajat` şi `setOre()` pentru modificarea stării obiectelor (modificarea datelor). În exemplul redat mai jos salariul se calculează pornind de la premisa că pentru concedii medicale se acordă un procent de 85% din valoarea salariului tarifar.

Listing13.1 Crearea tipului abstract `Angajat`

```
CREATE TYPE Angajat as OBJECT(
        Marca INTEGER,
        Nume VARCHAR2(30),
        Prenume VARCHAR 2(30),
        OreLucrate INTEGER,
        OreCO INTEGER,
        OreCM INTEGER,
```

```
        SalOrar NUMBER(16,2),
MEMBER FUNCTION CalculSalariu RETURN NUMBER,
MEMBER PROCEDURE SetOre(p_OreLucr INTEGER, p_OreCO INTEGER, p_OreCm INTEGER)
    ) ;
    /

CREATE OR REPLACE TYPE BODY Angajat AS
    MEMBER FUNCTION CalculSalariu RETURN NUMBER IS
    BEGIN
        RETURN (SELF.OreLucrate+SELF.OreCo + SELF.OreCM*0.85)*SELF.SalOrar;
    END CalculSalariu;
    MEMBER PROCEDURE SetOre(p_OreLucr INTEGER,
            p_OreCO INTEGER, p_OreCm INTEGER)
    IS
    BEGIN
       SELF.OreLucrate:=p_Orelucr;
       SELF.OreCO:=p_OreCO;
       SELF.OreCM:=p_OreCM;
END SetOre;
END;
    /
```

La o primă şi sumară analiză a listingului 13.1 pot fi desprinde două chestiuni interesante:

- Orice metodă primeşte un parametru implicit `SELF` care specifică obiectul curent (instanţa tipului `Angajat` asupra căreia se efectuează operaţia). Astfel, prin sintaxa `SELF.OreLucrate` se accesează datele stocate de atributul `OreLucrate` pentru obiectul curent. Utilizarea acestui parametru este opţională (`SELF.OreLucrate` este echivalent cu `OreLucrate`), dar recomandată pentru lizibilitatea codului sau în cazul operaţiilor complexe care necesită şi câteva variabile locale.
- Metoda `calculSalariu()` este definită fără paranteze pentru că nu acceptă nici un parametru. Parantezele sunt însă obligatorii pentru apelul metodei (vezi şi listing 13.2).

13.2. Crearea şi manipularea obiectelor în PL/SQL. Transferul prin valoare

Listing-ul 13.2 prezintă un bloc PL/SQL anonim prin care se construieşte un obiect de tip `Angajat`, se afişează salariul, apoi se modifică numărul de ore lucrate şi petrecute în concediu medical prin intermediul metodei `setOre`, şi se afişează din nou salariul. Construirea unui obiect de un anumit tip se realizează după o schemă familiară dezvoltatorilor care au avut de-a face cu un limbaj orientat-obiect: se apelează un constructor implicit căruia îi sunt furnizate valori pentru fiecare atribut în parte:

```
NEW  DenumireTip (val1, val2, .. valn)
```

Listing 13.2 Crearea unui obiect şi apelul metodelor sale într-un bloc PL/SQL

```
DECLARE
    obj_ang1 Angajat;
BEGIN
    obj_ang1:= NEW Angajat(111,'Asaftei','M.',160,0,0,100000);
    DBMS_OUTPUT.PUT_LINE(obj_ang1.Nume||' ' || obj_ang1.Prenume ||
                                        ' are sal:'  ||obj_ang1.calculSalariu());
    -- modificăm starea obiectului:
    obj_ang1.setOre(100,0,60);
    DBMS_OUTPUT.PUT_LINE(obj_ang1.Nume || ' ' || obj_ang1.Prenume ||
        ' are salariul dupa modificare oreCM=60:' || obj_ang1.calculSalariu());
END;
/
```

Notă. În listing-ul 13.2 invocarea metodei `setOre()` (care modifică starea obiectului) poate fi înlocuită de următoarea secvență de cod:

```
obj_ang1.oreLucrate:=100;
obj_ang1.oreCM:=60;
```

```
SQL> /
Asaftei M. are sal:16000000
Asaftei M. are sal dupa modficare oreCM=60:15100000
```

Figura 13.1. Rezultatul execuţiei listingului 13.2

Un aspect deosebit de important în ceea ce priveşte manipularea obiectelor într-un limbaj de programare se referă la modul de transfer al acestora (între variabile sau între un modul-program şi altul). În Oracle **obiectele se transmit prin valoare, şi nu prin referință** (aşa cum se întâmplă, spre exemplu, în Java). Listing-ul 13.3 şi figura 13.2 ilustrează acest aspect.

Listing13.3. Transferul obiectelor între variabile prin valoare

```
DECLARE
    obj_ang1 Angajat;
    obj_ang2 Angajat;

BEGIN
    obj_ang1:= NEW Angajat(111,'Asaftei','M.',160,0,0,100000);
    obj_ang2:=NEW Angajat(111,'Gologan','P.',0,0,0,150000);
    DBMS_OUTPUT.PUT_LINE('Pasul 1: doua obiecte diferite: doua variabile');
    DBMS_OUTPUT.PUT_LINE(obj_ang1.Nume||' '|| obj_ang1.Prenume ||
        ' are sal:'||obj_ang1.calculSalariu());
    DBMS_OUTPUT.PUT_LINE(obj_ang2.Nume||' '|| obj_ang2.Prenume ||
         ' are sal:'||obj_ang2.calculSalariu());

    -- transfer de OBIECT : se crează automat unul NOU identic !!!!!! :
    obj_ang1:=obj_ang2;
    DBMS_OUTPUT.PUT_LINE('Pasul 2 dupa atribuire: doua obiecte identice : doua variabile');
    DBMS_OUTPUT.PUT_LINE(obj_ang1.Nume||' '|| obj_ang1.Prenume ||
        ' are sal:'||obj_ang1.calculSalariu());
    DBMS_OUTPUT.PUT_LINE(obj_ang2.Nume||' '|| obj_ang2.Prenume || '
        are sal:'||obj_ang2.calculSalariu());

    -- se modifică doar obiectul 2 :
    obj_ang2.OreLucrate:=200;
    DBMS_OUTPUT.PUT_LINE('Pasul 3: modificam unul din obiecte (orelucrate =200) :
```

```
        celelalt ramane neschimbat');
    DBMS_OUTPUT.PUT_LINE(obj_ang1.Nume||' '|| obj_ang1.Prenume ||
        ' are sal:'||obj_ang1.calculSalariu());
    DBMS_OUTPUT.PUT_LINE(obj_ang2.Nume||' '|| obj_ang2.Prenume ||
        ' are sal:'||obj_ang2.calculSalariu());
END;
/
```

Rezultatul execuției blocului PL/SQL de mai sus îl regăsim în figura 13.2

```
Pasul 1: doua obiecte diferite: doua variabile
Asaftei M. are sal:16000000
Gologan P. are sal:0
Pasul 2 dupa atribuire: doua obiecte identice : doua variabile
Gologan P. are sal:0
Gologan P. are sal:0
Pasul 3: modificam unul din obiecte (orelucrate =200) : celelalt ramane neschimbat
Gologan P. are sal:0
Gologan P. are sal:30000000

PL/SQL procedure successfully completed.

SQL>
```

Figura 13.2. Rezultatul execuţiei listingului 13.3

Aşadar, operația de atribuire `obj_ang1:=obj_ang2` va determina construirea unui obiect complet diferit dar cu stare identică (valorile atributelor sunt identice). În mod similar se va realiza şi transferul de obiecte între parametrii unor funcții sau proceduri. Figura 13.3 schematizează transferul prin valoare.

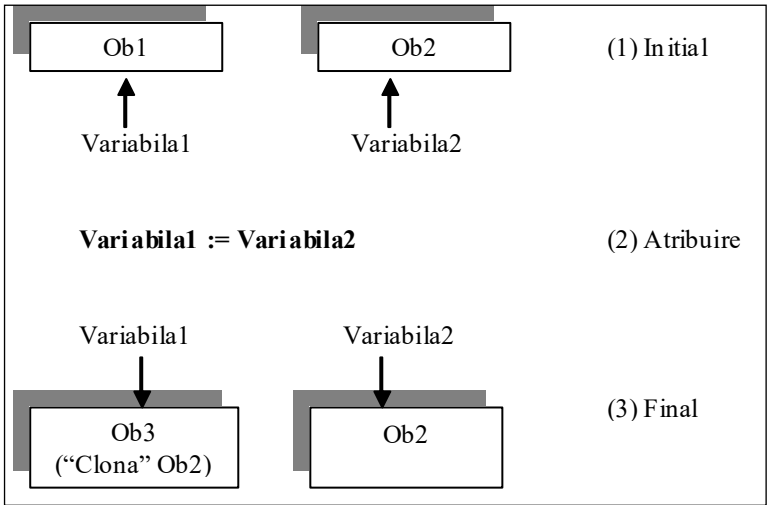

Figura 13.3. Transferul obiectelor între variabile se realizează prin valoare

Pentru transferul obiectelor prin referință, Oracle oferă operatorul `REF` despre care vom discuta însă ceva mai târziu.

13.3. Stocarea şi gestionarea obiectelor în baza de date

Oracle furnizează două tehnologii de stocare a obiectelor în baza de date:

- ***obiecte linie*** – sunt stocate sub formă de linii ale unei tabele construită pe baza tipului căruia îi aparţin respectivele obiecte; atributele unei asemenea tabele vor fi tocmai atributele din definiţia tipului iar un obiect va constitui o înregistrare;
- ***obiecte coloană*** – persistenţa se asigură sub formă de valori ale unui atribut declarat într-o tabelă relaţională clasică; pentru fiecare înregistrare, respectivul atribut va conţine un obiect de tipul declarat.

Dezavantajul imposibilităţii gestionării "native" a referinţelor este diluat printr-un pachet special PL/SQL căruia îi va fi dedicată ultima parte a acestui paragraf.

13.3.1. Crearea şi actualizare

Corespunzător celor două tehnologii de stocare a obiectelor, listingul 13.4 conţine două comenzi `CREATE TABLE` urmate de comenzi `INSERT` pentru popularea acestora cu date.

Listing 13.4. Două modalitaţi de stocare a obiectelor în BD

```
CREATE TABLE Angajati_ObjTbl OF ANGAJAT;
        -- sau :
CREATE TABLE Angajati_RelTbl (id INTEGER, pers ANGAJAT);

    -- adaugam date in tabela obiectuala ca în orice alta tabela relaţionala:
INSERT INTO Angajati_ObjTbl VALUES (111,'Asaftei','V.', 0,0,0,200000);
    --sau prin crearea în mod explicit a unui obiect nou:
INSERT INTO Angajati_ObjTbl VALUES (NEW Angajat (112,'Mandache','M.',0,0,0,150000));

    -- inserare date in tabela relational-obiectula:
INSERT INTO  Angajati_RelTbl VALUES (1,NEW Angajat(111,'Asaftei','V.', 0,0,0,200000)) ;
INSERT INTO  Angajati_RelTbl VALUES (2, NEW Angajat (112,'Mandache','M.',0,0,0,150000));
```

Prima linie de cod generează o tabelă obiectuală ale cărei înregistrări vor fi obiecte de tip `Angajat`, iar a doua linie de cod defineşte o tabelă relaţională în care atributul `pers` este de tip `Angajat` şi va stoca obiecte. Tot aici observăm că operaţia INSERT-SQL pentru tabela obiectuală nu este cu nimic diferită de modelul relaţional clasic. Acest aspect se manifestă şi în cazul efectuării unor modificări sau interogări (vezi figura 13.4).

Lucrurile se schimbă însă în cazul tabelei relaţionale `Angajati_RelTbl` - vezi figura 13.5 - în privinţa a două aspecte:

- pentru a extrage datele stocate de atributele obiectelor vom utiliza notaţia specifică modelului de programare orientat-obiect (cu punct);
- **este obligatorie utilizarea unui alias pentru tabela** în cauză, pentru a manipula apoi obiectele după şablonul: `AliasTabela.AtributTabe-`

`la.AtributObiect`. În caz contrar obţinem eroarea ORA-00904, ca în figura 13.5.

```
SQL> select nume from Angajati_ObjTbl;

NUME
------------------------------
Asaftei
Mandache

SQL> update Angajati_ObjTbl set salorar=230000 where marca=111;

1 row updated.

SQL> select * from Angajati_ObjTbl;

     MARCA NUME                           PRE ORELUCRATE      ORECO      ORECM    SALORAR
---------- ------------------------------ --- ---------- ---------- ---------- ----------
       111 Asaftei                        V.           0          0          0     230000
       112 Mandache                       M.           0          0          0     150000

SQL> |
```

Figura 13.4. Interogarea şi actualizarea tabelei obiectuale ANGAJATI_OBJTBL

```
SQL> select pers.nume from  Angajati_RelTbl;
select pers.nume from  Angajati_RelTbl
       *
ERROR at line 1:
ORA-00904: "PERS"."NUME": invalid identifier


SQL> select A.pers.nume from  Angajati_RelTbl A;

PERS.NUME
--------------------------------
Asaftei
Mandache

SQL> update Angajati_RelTbl A set A.pers.salorar=230000 where A.pers.marca=111;

1 row updated.

SQL> select * from  Angajati_RelTbl;

        ID
----------
PERS(MARCA, NUME, PRENUME, ORELUCRATE, ORECO, ORECM, SALORAR)
--------------------------------------------------------------------------------
         1
ANGAJAT(111, 'Asaftei', 'V.', 0, 0, 0, 230000)

         2
ANGAJAT(112, 'Mandache', 'M.', 0, 0, 0, 150000)


SQL>
```

Figura 13.5. Interogarea şi actualizarea tabelei relaţionale ANGAJATI_RELTBL

Este important de sesizat faptul că până acum s-au prezentat tehnici de actualizare şi interogare a ***datelor*** obiectelor (valorile stocate de atribute). De multe ori însă este necesar ca în urma unei interogări să obţinem chiar ***obiecte*** sau să

înlocuim în tabela obiectuală un obiect cu altul complet nou. În acest scop este disponibilă funcţia `VALUE(alias_tabela_ob)` ce returnează *obiecte noi,* identice cu obiectele din tabela-obiectuală specificată prin `alias_tabela_ob`. Funcţia poate fi folosită numai în fraze SQL şi numai pentru tabele obiectuale.

Afirmaţiile de mai sus sunt puse în aplicare de figura 13.5 bis, astfel:

- prin intermediul primei interogări obţinem obiecte şi nu valori ale atributelor lor;
- variabila `obj` din blocul PL/SQL preia o *copie* a obiectului cu `marca`=112 din tabela `Angajati_ObjTbl`; tot aici se modifică valoarea atributului `nume` al *noului obiect* (`obj.nume='test'`)
- ultima interogare denotă faptul că obiectul cu `marca`=112 din tabela `Angajati_ObjTbl` a rămas neschimbat.

```
SQL> SELECT VALUE(A) FROM angajati_objtbl A ;

VALUE(A)(MARCA, NUME, PRENUME, ORELUCRATE, ORECO, ORECM, SALORAR)
--------------------------------------------------------------------
ANGAJAT(111, 'Asaftei', 'V.', 0, 0, 0, 230000)
ANGAJAT(112, 'Mandache', 'M.', 0, 0, 0, 150000)

SQL> DECLARE
  2  obj ANGAJAT ;
  3  BEGIN
  4  SELECT VALUE (A) INTO obj FROM angajati_objtbl A
  5  WHERE A.marca=112 ;
  6  obj.nume := 'test' ;
  7  END ;
  8  /

PL/SQL procedure successfully completed.

SQL> SELECT VALUE(A) FROM angajati_objtbl A ;

VALUE(A)(MARCA, NUME, PRENUME, ORELUCRATE, ORECO, ORECM, SALORAR)
--------------------------------------------------------------------
ANGAJAT(111, 'Asaftei', 'V.', 0, 0, 0, 230000)
ANGAJAT(112, 'Mandache', 'M.', 0, 0, 0, 150000)
```

Figura 13.5 Bis. Funcţia VALUE() returnează “clone” ale obiectelor stocate într-o tabelă obiectuală

Cu ajutorul aceleiaşi funcţii putem înlocui obiectul pentru care `marca` este 112 cu alt obiect:

```
UPDATE angajati_objTbl  A
     SET VALUE(A)=NEW Angajat(115,'Filimon','A.',0,0,0,0)
      WHERE  A.marca=112;
```

Important! O modificare de genul celei de mai sus va lăsa intactă adresa logică a obiectului (vezi paragrafele referitoare la referinţe spre obiecte).

Pentru oricare din cele două tabele se poate defini o cheie primară, relativ la atributul `marca`:

```
ALTER TABLE Angajati_ObjTbl ADD PRIMARY KEY (marca);
ALTER TABLE Angajati_RelTbl ADD PRIMARY KEY (pers.marca);
```

Figura 13.6 demonstrează funcționalitatea cheii primare în ambele cazuri.

```
SQL> insert into Angajati_ObjTbl values (new Angajat (112,'Mandache','M.',0,0,0,150000));
insert into Angajati_ObjTbl values (new Angajat (112,'Mandache','M.',0,0,0,150000))
*
ERROR at line 1:
ORA-00001: unique constraint (LIVIU.SYS_C001554) violated


SQL>  insert into Angajati_RelTbl values (3, new Angajat (112,'Mandache','M.',0,0,0,150000));
 insert into Angajati_RelTbl values (3, new Angajat (112,'Mandache','M.',0,0,0,150000))
*
ERROR at line 1:
ORA-00001: unique constraint (LIVIU.SYS_C001555) violated


SQL>
```

Figura 13.6 Chei primare definite pe baza atributelor obiectelor

13.3.2. Referințe spre obiecte. Tipul de date REF

În aplicațiile orientate-obiect există diverse situații în care este necesară o adresă logică (așa numitul "pointer") spre un obiect existent, adresă prin intermediul căreia să fie gestionat exact obiectul respectiv, şi nu o copie a acestuia. Mai mult, dacă privim lucrurile din perspectiva arhitecturii bazei de date, tabelele copil se construiesc pe baza referințelor spre obiectele din tabelele părinte (spre exemplu pentru tabela SPORURI, care gestionează diverse sporuri acordate angajaților, ar fi necesar un pointer spre obiectele stocate în tabela părinte ANGAJATI_OBJTBL definită anterior) .

În acest sens Oracle furnizează tipul de dată `REF` ce defineşte un **pointer spre un obiect-linie** existent, prin intermediul căruia putem modela relații între obiecte (în special relații de tip "one-to-many") şi care poate fi utilizat în diverse scopuri, după cum urmează:

- pentru a examina şi actualiza datele obiectului sursă;
- pentru a obține o copie a obiectului sursă;
- pentru a înlocui obiectul sursă spre care ține pointer-ul.

Important! Un `REF` poate defini un pointer numai spre un obiect stocat într-o tabelă de obiecte şi nu spre obiecte rezidente în memorie sau ca valori ale atributelor unei tabele relaționale clasice. De asemenea, tipul obiectului trebuie să fie obligatoriu cel declarat la definirea tipului `REF`[1].

Pentru exemplificare, vom porni de la definiția tabelei ANGAJATI_OBJTBL creată anterior şi vom construi tabela SPORURI_REFTBL pentru gestionarea orelor lucrate de angajați, astfel:

```
CREATE TABLE Sporuri_RefTbl (
  ref_angajat REF ANGAJAT REFERENCES Angajati_ObjTbl,
```

[1] Sau un subtip al său (vezi paragraful referitor la extinderea tipurilor)

```
an INTEGER,
luna INTEGER,
OreSporNoapte INTEGER,
OreSporNocive INTEGER
);
```

Aşadar, atributul `ref_angajat` al tabelei va stoca referinţe numai spre obiecte de tip `Angajat` existente în tabela ANGAJATI_OBJTBL, restricţie implementată în modelul relaţional prin cheie străină (restricţie referenţială).

Notă. Tipul `REF` poate fi utilizat şi la definirea atributelor unui tip abstract, însă **nu** poate fi utilizat pentru definirea cheilor primare.

Obţinerea unei referinţe spre un obiect se realizează cu ajutorul funcţiei `REF(obiect)` ce poate fi folosită doar în fraze SQL. Figura 13.7 ilustrează două aspecte esenţiale referitor la tipurile `REF`:

- o referinţă reprezintă adresa, identificatorul unic (`OID - Object IDentifier`) generat de sistem la crearea obiectului (acest identificator este atribuit automat la momentul iniţializării obiectului şi este unic în baza de date);
- prin intermediul referinţelor manipulăm obiectele în mod clasic orientat-obiect, folosind notaţia cu punct (în exemplul din figură, expresia `REF(A)).nume` returnează datele stocate de atributul `nume` al obiectelor din tabela ANGAJATI_OBJTBL.

```
  1* select REF(A) from angajati_objtbl A
SQL> /

REF(A)
--------------------------------------------------------------------------------
0000280209597 23BED4FCB43DBA0A738DC4E087CB6AEBE10029B3D4EFC92C14314011872E1004037
F70000

0000280209DB131B97134A49718E1BE037187CD4E1AEBE10029B3D4EFC92C14314011872E1004037
F70001


SQL> ed
Wrote file afiedt.buf

  1* select (REF(A)).nume from angajati_objtbl A
SQL> /

(REF(A)).NUME
------------------------------
Asaftei
Mandache

SQL>
```

Figura 13.7. Utilizarea funcţiei REF() pentru a obţine “pointeri” spre obiecte

Pentru că o referinţă se poate obţine numai prin intermediul unei fraze SELECT SQL, adăugăm înregistrări în tabela SPORURI_REFTBL după următorul model:

```
INSERT INTO  Sporuri_RefTbl
    (SELECT REF(A),2003,03,0,0 FROM Angajati_ObjTbl A);
```

Vor fi inserate atâtea înregistrări câte linii sunt în tabela ANGAJATI_OBJTBL. Figura 13.8 evidenţiază faptul că, în mod similar tabelelor obiectuale, pentru a manipula obiectele sursă ale unor referinţe este obligatorie utilizarea unui alias pentru tabelă.

```
SQL> select ref_angajat.nume, oreSporNoapte, oreSporNocive from sporuri_reftbl;
select ref_angajat.nume, oreSporNoapte, oreSporNocive from sporuri_reftbl
       *
ERROR at line 1:
ORA-00904: "REF_ANGAJAT"."NUME": invalid identifier


SQL> ed
Wrote file afiedt.buf

  1* select S.ref_angajat.nume, oreSporNoapte, oreSporNocive from sporuri_reftbl S
SQL> /

REF_ANGAJAT.NUME               ORESPORNOAPTE ORESPORNOCIVE
------------------------------ ------------- -------------
Asaftei                                    0             0
Mandache                                   0             0

SQL> update Sporuri_RefTbl S set oreSporNoapte=80 where S.ref_angajat.marca=112;

1 row updated.

SQL>
```

Figura 13.8. Acces la datele obiectelor prin intermediul referinţelor

13.3.3. Referinţe spre obiecte în PL/SQL. Pachetul UTL_REF

Posibilitatea obţinerii unei *referinţe* spre un obiect deja existent are o deosebită importanţă asupra codului unei aplicaţii orientate-obiect din perspectiva următoarelor aspecte:

- referinţă spre un obiect existent oferă posibilitatea manipulării acestuia de către mai multe module-program, în funcţie de necesităţi;
- eventuală modificare a stării obiectului s-ar efectua într-un singur loc şi ar fi imediat disponibilă tuturor modulelor ce fac referire la acesta;
- referinţele oferă o modalitate de gestionare mult mai riguroasă a memoriei disponibile.

PL/SQL nu suportă, la momentul actual (versiunea Oracle9i2), navigarea între obiecte prin intermediul referinţelor, deşi se pot declara variabile de tip `REF` care să conţină adrese de obiecte. Figura 13.9 demonstrează această afirmaţie.

```
SQL> ed
Wrote file afiedt.buf

  1  DECLARE
  2  angajat_ref REF ANGAJAT;
  3  begin
  4  Select REF(A) INTO angajat_ref from Angajati_ObjTbl A where A.marca=112;
  5  DBMS_OUTPUT.PUT_LINE( angajat_ref.Nume);
  6* END;
SQL> /
DBMS_OUTPUT.PUT_LINE( angajat_ref.Nume);
                                *
ERROR at line 5:
ORA-06550: line 5, column 35:
PLS-00536: Navigation through REF variables is not supported in PL/SQL.
ORA-06550: line 5, column 1:
PL/SQL: Statement ignored
```

Figura 13.9. Referinţele nu pot fi utilizate în PL/SQL pentru manipularea obiectelor

Oracle oferă totuşi o posibilitate de gestionare a referinţelor în PL/SQL prin intermediul pachetului `UTL_REF` ale cărui proceduri sunt următoarele:

a. `UTL_REF.SELECT_OBJECT(reference IN REF "<typename>", object IN OUT "<typename>");`
extrage obiectul adresat de parametrul `reference` şi pasează valoarea variabilei `object`. Aşadar s*e va obţine un obiect nou, copie identică a obiectului-sursă*. Fraza SQL echivalentă este:

```
SELECT VALUE(t)
INTO object
FROM object_table t WHERE REF(t) = reference;
```

b. `UTL_REF.UPDATE_OBJECT( reference IN REF "<typename>", object IN "<typename>");`
înlocuieşte obiectul de la adresa specificată prin parametrul `reference` cu unul nou. Cu alte cuvinte, adresa (OID) rămâne neschimbată doar că va fi un alt obiect identificat prin intermediul ei. Fraza SQL echivalentă este:

```
UPDATE object_table t
SET VALUE(t) = object WHERE REF(t) = reference;
```

c. `UTL_REF.DELETE_OBJECT( reference IN REF "<typename>")`
şterge obiectul identificat prin referinţa `reference;`

d. `UTL_REF.LOCK_OBJECT( reference IN REF "<typename>")`

e. `UTL_REF.LOCK_OBJECT( reference IN REF "<typename>", object IN OUT "<typename>");`
blochează obiectul identificat prin referinţa `reference` pentru a nu fi modificat de altă tranzacţie. Fraza SQL echivalentă este:

```
SELECT VALUE(t)
```

```
INTO object
FROM object_table t WHERE REF(t) = reference
FOR UPDATE;
```

Aşadar, pentru fiecare dintre proceduri se poate construi o frază SQL echivalentă. Avantajul utilizării pachetului `UTL_REF` constă în faptul că, odată ce obţinem o referinţă, obiectul poate fi manipulat fără a cunoaşte exact tabela în care se află. Listingul 13.5 oferă o secvenţă de cod prin care se testează fiecare din procedurile de mai sus (cu excepţia `Lock_Object`).

Listing 13.5. O procedură de folosire a pachetului UTL_REF

```
CREATE OR REPLACE PROCEDURE test_utl_ref (v_ref IN REF ANGAJAT) IS
    obj_clona_angajat ANGAJAT;
BEGIN

    -- pas 1 – obţinere obiect
    UTL_REF.SELECT_OBJECT(v_ref,obj_clona_angajat);
    DBMS_OUTPUT.PUT_LINE('Pas1/ variabila locala detine o copie a ob. din tabela:                    '
                    ||obj_clona_angajat.Marca || '  ' || obj_clona_angajat.Nume);

    -- pas 2 – înlocuire obiect
    UTL_REF.UPDATE_OBJECT(v_ref, NEW Angajat(120,'TEST2',null,0,0,0,0));
    DBMS_OUTPUT.PUT_LINE
        ('Pas2/ dupa modificare variabila locala detine in continuare copia veche:          '
            || obj_clona_angajat.Marca || '  ' || obj_clona_angajat.Nume);

    -- pas 3 – obţinerea obiectul după modificare
    UTL_REF.SELECT_OBJECT(v_ref,obj_clona_angajat);
    DBMS_OUTPUT.PUT_LINE(
        'Pas3/ Variabila locala a preluat valoarea obiectului nou de la aceeasi referinta:'
        || obj_clona_angajat.Marca || '  ' || obj_clona_angajat.Nume);

    -- pas 4 – stegere obiect
    UTL_REF.DELETE_OBJECT(v_ref);

END;
```

Logica modulului `test_utl_ref` (vezi si rezultatul executiei lui din figura 13.10) este următoarea:

- Pasul 1 – într-o variabilă locală - `obj_clona_angajat` - de tip `Angajat` - se preia valoarea obiectului a cărui referinţă a fost obţinută prin parametrul `v_ref`;
- Pasul 2 – se modifică obiectul identificat prin referinţa `v_ref` prin înlocuirea lui (înlocuirea se va petrece in tabela sursă) cu altul complet nou cu `marca`=120; textul afişat la acest pas demonstrează faptul că variabila locală deţine o copie;
- Pasul 3 – se reiniţializează variabila locală cu obiectul nou de la aceeaşi referinţă;
- Pasul 4 – obiectul se şterge din tabela sursă (se observă la interogarea finală din figura 13.10 că au rămas doar două obiecte în tabela `Angajati_ObjTbl`).

```
SQL>  select * from angajati_objtbl;

     MARCA NUME                           PRE ORELUCRATE      ORECO      ORECM    SALORAR
---------- ------------------------------ --- ---------- ---------- ---------- ----------
       111 Asaftei                        V.           0          0          0     230000
       112 Mandache                       M.           0          0          0     150000
       119 Test                           T.           0          0          0          0

SQL> declare
  2  ref_a REF ANGAJAT;
  3  BEGIN
  4  select ref(A) into ref_a from angajati_objTbl A where A.marca=119;
  5  test_UTL_REF(ref_a);
  6  end;
  7
  8  /
Pas1/ variabila locala detine o copie a ob. din tabela:                        119  Test
Pas2/ dupa modificare varibila locala detine in continuare copia veche:        119  Test
Pas3/ Variabila locala a preluat valoarea obiectului nou de la aceeasi referinta:120  TEST2

PL/SQL procedure successfully completed.

SQL> select * from angajati_objtbl;

     MARCA NUME                           PRE ORELUCRATE      ORECO      ORECM    SALORAR
---------- ------------------------------ --- ---------- ---------- ---------- ----------
       111 Asaftei                        V.           0          0          0     230000
       112 Mandache                       M.           0          0          0     150000
```

Figura 13.10. Rezultatul execuţiei procedurii TEST_UTL_REF

13.4. Extinderea tipurilor. Moştenire şi polimorfism

Început timid în Oracle 8, suportul pentru toate ingredintele orientării pe obiecte s-a ameliorat de la versiune la versiune, iar astăzi dezvoltatorii de aplicaţii Oracle nu mai au frisoane (mari) când este vorba de moştenire, suprascriere, supraîncărcare sau polimorfism.

13.4.1. Ierarhii de obiecte

Un concept esenţial al "filosofiei" orientate-obiect este reprezentat de *moştenire*, ca proces de specializare a tipurilor prin crearea unor *subtipuri* derivate din cel de bază, numit *supertip*. Subtipurile *moştenesc* toţi membrii supertipului(date şi metode) şi prezintă cel puţin un element de noutate pentru a particuza obiectele (entităţile din lumea reală). Elementul de noutate se poate concretiza în:

- atribute şi/sau metode noi;
- modificarea comportamentului – o altă implementare a metodelor moştenite.

Ansamblul alcătuit din tipul de bază (cel mai abstract (scuzaţi exprimarea !)) şi toate subtipurile sale specializate este numit *ierarhie*. Figura 1 ilustrează o ierarhie simplificată a persoanelor ce-şi desfăşoară activitatea în mediul universitar. Specializarea subtipului poate însemna: adăugarea unor noi membri (atribute,

metode) sau modificarea comportamentului uneia din metodele moştenite (proces numit *suprascriere*). Suprascrierea metodelor face posibil *polimorfismul* (două obiecte de acelaşi supertip dar de subtipuri diferite se comportă diferit la execuția metodei suprascrise).

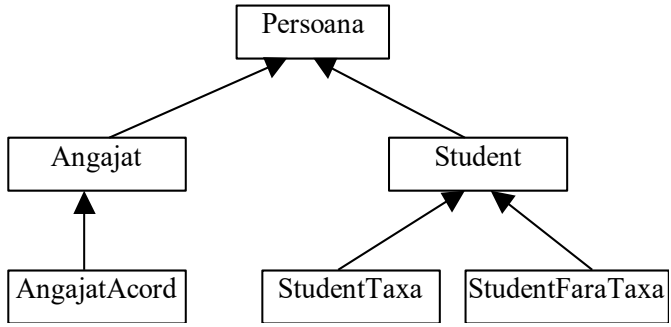

Figura 13.11. O ierarhie de tipuri

Începând cu versiunea 9i2, Oracle oferă suport deplin pentru toate conceptele descrise anterior. Astfel, definirea unui subtip derivat dintr-un supertip poate fi realizată numai dacă acesta din urmă a fost declarat `NOT FINAL` după cum urmează:

```
CREATE TYPE supertip AS OBJECT (…) NOT FINAL ;
CREATE TYPE subtip UNDER supertip
                    (…atribute şi/sau metode noi…);
```

De cele mai multe ori, un modul de cod nu lucrează cu obiecte instanțiate din supertip, ci cu obiecte specializate. Supertipurile se utilizează pentru a trata unitar toate obiectele din tipurile derivate. Astfel, conform ierarhiei definite în figura 13.11, nu vor exista cazuri în care vom utiliza obiecte de tip `Persoana` sau `Student` ci doar obiecte de tip `Angajat`, `AngajatAcord`, `StudentTaxa` şi `StudentFaraTaxa`. Pentru a asigura însă flexibilitatea aplicației şi pentru a suporta o eventuală dezvoltare a modelului, aceste obiecte vor fi *tratate* în anumite faze ale prelucrărilor ca fiind de tipul de bază, `Persoana`. În aceeaşi ordine de idei, la nivelul supertipului se pot defini metode care nu implementează nici un comportament, datorită faptului că acest comportament va fi diferit pentru fiecare subtip în parte. Ca urmare, proiectantul supertipului va delega responsabilitatea definirii acțiunii de executat către proiectanții subtipurilor care vor avea în vedere cerințele proprii. Spre exemplu, înscrierea unui student la un examen presupune verificarea îndeplinirii unor condiții prealabile diferite pentru studenții cu taxă față de cei fără taxă (primii trebuie să-şi plătească taxa de şcolarizare în întregime pentru a putea participa la examen). Iată de ce proiectantul tipului `Student` va putea recurge la definirea unei eventuale metode `inscrieEx()` ca fiind abstractă, urmând ca subtipurile `StudentTaxa` şi `StudentFaraTaxa` să implementeze fiecare propriul mod de evaluare a condițiilor de intrare în examen.

Paradigma orientată-obiect oferă posibilitatea creării unor tipuri ce nu pot fi instanţiate (nu putem obţine obiecte de tipul respectiv) şi a unor metode fără implementare ce vor trebui obligatoriu suprascrise de tipurile derivate. În acest scop, Oracle9i2 furnizează următoarea sintaxă:

```
CREATE TYPE supertip AS OBJECT (
   … lista atribute…,
   NOT INSTANTIABLE MEMBER FUNCTION metoda1 RETURN NUMBER
                          )
         NOT INSTANTIABLE NOT FINAL;
```

iar pentru crearea subtipului:

```
CREATE TYPE subtip UNDER supertip (
   … atribute şi/sau metode noi…,
OVERRIDING MEMBER FUNCTION metoda1 RETURN NUMBER);

CREATE TYPE BODY subtip AS
OVERRIDING MEMBER FUNCTION metoda1 RETURN NUMBER IS
BEGIN
   …
END metoda1;
END;
```

Există, totuşi câteva restricţii în ceea ce priveşte suprascrierea metodelor:

- metodele pot fi declarate de tip `FINAL`, caz în care nu mai pot fi suprascrise de subtipuri;
- metodele de ordonare (de tip ORDER[2]) pot apare doar în tipul rădăcină al unei ierarhii şi nu pot fi suprascrise în tipurile derivate;
- dacă o metodă furnizează valori implicite pentru anumiţi parametri, în tipul derivat, la suprascrierea metodei, se vor specifica exact aceleaşi valori pentru aceeaşi parametri.

Un alt concept esenţial îl reprezintă **supraîncărcarea metodelor** - posibilitatea existenţei simultan a mai multor metode cu acelaşi nume dar cu un număr diferit de parametri sau cu acelaşi număr de parametri dar de tipuri diferite. Acest aspect prezintă o deosebită importanţă într-o ierarhie. În exemplul următor subtipul `AngajatAcord` defineşte prin metoda `getSalariu(procRealiz NUMBER)` o supraîncărcare (şi **nu** o suprascriere) a metodei `getSalariu()` moştenită din supertip. Cu alte cuvinte, pentru un obiect `obj` de tip `AngajatAcord` următoarele două apeluri sunt valide, fiecare executând altceva: `obj.getSalariu()` şi `obj.getSalariu(0.8)`.

```
CREATE TYPE Angajat AS OBJECT (
   …
```

[2] vezi paragrafele următoare pentru metode de tip `MAP` şi `ORDER`

```
    MEMBER FUNCTION getSalariu RETURN NUMBER
                                    )

CREATE TYPE AngajatAcord UNDER Angajat (
    MEMBER FUNCTION getSalariu(procRealiz NUMBER)
          RETURN NUMBER
          ...
                                          )
```

Exemplificarea conceptelor prezenate mai sus poate fi foarte bine realizată dacă încercăm elaborarea unei soluţii pentru una din problemele uzuale ale unei aplicaţii de salarizare: calculul drepturilor salariale aferente fiecărui angajat având în vedere diversitatea tipologiei acestor drepturi (sporuri, concedii de odihnă şi medicale, premii etc.). În acest sens, aşa cum reiese şi din figura 13.12, putem considera că sumele aferente diverselor sporuri, concedii, ore lucrate sau premii sunt, în esenţă, speţe ale unui singur tip, `Drepturi`.

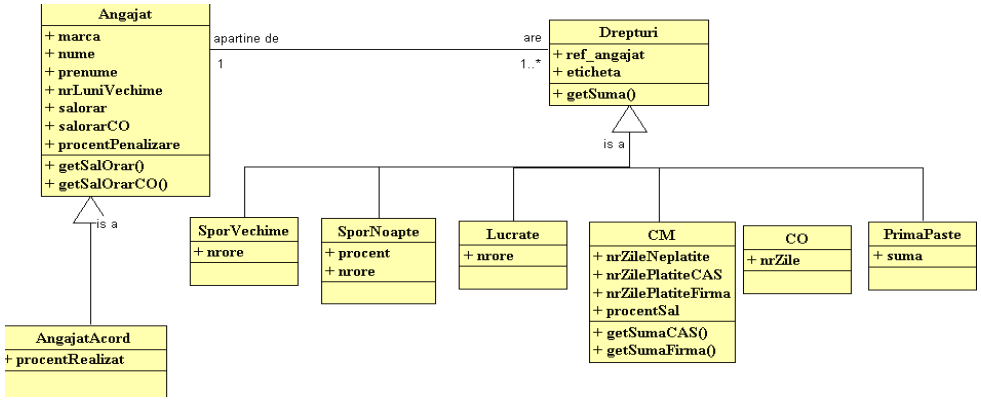

Figura 13.12. O ierarhie de tipuri pentru calculul drepturilor salariale

Acest tip defineşte:

- metoda `getSuma()` ce va returna suma aferentă dreptului respectiv (pentru că este vorba de o metodă abstractă (fără implementare), toate subtipurile vor fi obligate să implementeze un comportament specific prin suprascrierea acestei metode);
- atributul `ref_Angajat` ca referinţă spre o tabelă de obiecte de tip `Angajat` pentru a identifica persoana pentru care este acordat dreptul respectiv;
- atributul `eticheta` ce are drept scop stocarea unei denumiri pentru dreptul respectiv, denumire ce va fi utilizată la afişarea sumei în diverse rapoarte;

Tipul `Angajat` conţine şi două metode pentru obţinerea salariului real (în eventualitatea că salariatul ar "beneficia" de o penalizare). Observăm că şi tipul `Angajat` a fost derivat, prin crearea subtipului `AngajatAcord` ce prezintă un

atribut suplimentar, `procentRealizat`, utilizat la suprascrierea metodei `getSalOrar()` (salarizarea în acord înseamnă acordarea tuturor drepturilor în funcţie de realizări).

Generosul listing 13.6 îşi propune crearea în Oracle a ierarhiei descrise.

Listing 13.6. Codul sursă pentru implementarea diagramei din figura 13.12

```
DROP TABLE angajati_objtbl CASCADE CONSTRAINTS
/
DROP TABLE angajati_reltbl CASCADE CONSTRAINTS
/
DROP TABLE Sporuri_RefTbl CASCADE CONSTRAINTS
/
DROP TYPE Drepturi FORCE
/
DROP TYPE Angajat FORCE
/
CREATE OR REPLACE TYPE Angajat AS OBJECT (
    marca INTEGER,
    nume VARCHAR2(50),
    prenume VARCHAR2(50),
    nrLuniVechime INTEGER,
    salorar NUMBER(16,2),
    salorarCO NUMBER(16,2),
    procentPenalizare NUMBER(5,2),
    MEMBER FUNCTION getSalOrar RETURN NUMBER,
    MEMBER FUNCTION getSalOrarCO RETURN NUMBER
) NOT FINAL
/

-- suprascriem doar metoda getSalOrar() deoarece drepturile pe concediul de odihnă
-- nu sunt în funcţie de realizări
CREATE OR REPLACE  TYPE AngajatAcord UNDER ANGAJAT (
    procentRealizat NUMBER(5,2),
    OVERRIDING MEMBER FUNCTION getSalOrar RETURN NUMBER

)
/

CREATE OR REPLACE  TYPE BODY Angajat AS
    MEMBER FUNCTION getSalOrar RETURN NUMBER IS
        BEGIN
        RETURN SELF.salOrar*(1-SELF.procentPenalizare);
    END getSalOrar;

    MEMBER FUNCTION getSalOrarCO RETURN NUMBER IS
        BEGIN
        RETURN SELF.salOrarCO*(1-SELF.procentPenalizare);
    END getSalOrarCO;
END;
/

CREATE OR REPLACE  TYPE BODY AngajatAcord AS
    OVERRIDING MEMBER FUNCTION getSalOrar RETURN NUMBER IS
        BEGIN
        RETURN SELF.salOrar*(1-SELF.procentPenalizare)*SELF.procentRealizat;
    END getSalOrar;

END;
```

```
/

CREATE OR REPLACE TYPE DREPTURI AS OBJECT (
     ref_Angajat REF ANGAJAT,
     eticheta VARCHAR2(20),
     NOT INSTANTIABLE MEMBER FUNCTION getSuma RETURN NUMBER
) NOT INSTANTIABLE NOT FINAL
/

CREATE OR REPLACE TYPE SporVechime UNDER DREPTURI (
     nrore NUMBER(5,2),
     OVERRIDING MEMBER FUNCTION getSuma RETURN NUMBER
     )
/

CREATE OR REPLACE TYPE SporNoapte UNDER DREPTURI (
     procent NUMBER(5,2),
     nrore NUMBER(5,2),
     OVERRIDING MEMBER FUNCTION getSuma RETURN NUMBER
     )
/

CREATE OR REPLACE TYPE Lucrate UNDER DREPTURI (
     nrore NUMBER(5,2),
     OVERRIDING MEMBER FUNCTION getSuma RETURN NUMBER
     )
/

CREATE OR REPLACE TYPE CO UNDER DREPTURI (
     nrZile NUMBER(5,2),
     OVERRIDING MEMBER FUNCTION getSuma RETURN NUMBER
     )
/

CREATE OR REPLACE TYPE CM UNDER DREPTURI (
     nrZileNeplatite INTEGER,
     nrZilePlatiteCAS INTEGER,
     nrZilePlatiteFirma INTEGER,
     procentSal NUMBER(5,2),
     OVERRIDING MEMBER FUNCTION getSuma RETURN NUMBER,
     MEMBER FUNCTION getSumaCAS RETURN NUMBER,
     MEMBER FUNCTION getSumaFirma RETURN NUMBER
     )
/

CREATE OR REPLACE TYPE PrimaPaste UNDER DREPTURI(
     suma NUMBER(16,2),
     OVERRIDING MEMBER FUNCTION getSuma RETURN NUMBER
)
/

 CREATE OR REPLACE TYPE BODY SporVechime AS
     OVERRIDING MEMBER FUNCTION getSuma RETURN NUMBER IS
          obj_angajat ANGAJAT;
          procent_ Transe_sv.Procent_sv%TYPE;
 BEGIN
      UTL_REF.SELECT_OBJECT(SELF.ref_angajat,obj_angajat);
      SELECT procent_sv INTO procent_ FROM transe_sv
           WHERE (obj_angajat.nrLuniVechime/12) BETWEEN ani_limita_inf AND ani_limita_sup;
     RETURN obj_angajat.getSalOrar() * SELF.nrore*procent_;
```

```
 EXCEPTION
      WHEN NO_DATA_FOUND THEN RETURN 0;
      WHEN OTHERS THEN RETURN -1;
 END getSuma;
 END;
/

CREATE OR REPLACE TYPE BODY SporNoapte AS
     OVERRIDING MEMBER FUNCTION getSuma RETURN NUMBER IS
          obj_angajat ANGAJAT;
     BEGIN
          UTL_REF.SELECT_OBJECT(SELF.ref_angajat,obj_angajat);
          RETURN obj_angajat.getSalOrar()*SELF.nrore*SELF.procent;
     EXCEPTION
          WHEN OTHERS THEN RETURN -1;
     END getSuma;
END;
/

CREATE OR REPLACE TYPE BODY Lucrate AS
     OVERRIDING MEMBER FUNCTION getSuma RETURN NUMBER IS
          obj_angajat ANGAJAT;
     BEGIN
          UTL_REF.SELECT_OBJECT(SELF.ref_angajat,obj_angajat);
          RETURN obj_angajat.getSalOrar()*SELF.nrore;
     EXCEPTION
          WHEN OTHERS THEN RETURN -1;
     END getSuma;
END;
/

CREATE OR REPLACE TYPE BODY co AS
     OVERRIDING MEMBER FUNCTION getSuma RETURN NUMBER IS
          obj_angajat ANGAJAT;
     BEGIN
          UTL_REF.SELECT_OBJECT(SELF.ref_angajat,obj_angajat);
-- considerăm ziua de CO de 8 ore
          RETURN obj_angajat.getSalOrarCO()*8*SELF.nrZile;
     EXCEPTION
          WHEN OTHERS THEN RETURN -1;
     END getSuma;
END;
/

CREATE OR REPLACE TYPE BODY co AS
     OVERRIDING MEMBER FUNCTION getSuma RETURN NUMBER IS
          obj_angajat ANGAJAT;
     BEGIN
          UTL_REF.SELECT_OBJECT(SELF.ref_angajat,obj_angajat);
          RETURN obj_angajat.getSalOrarCO()*SELF.nrZile;
     EXCEPTION
          WHEN OTHERS THEN RETURN -1;
     END getSuma;
END;
/

CREATE OR REPLACE TYPE BODY PrimaPaste AS
     OVERRIDING MEMBER FUNCTION getSuma RETURN NUMBER IS
     BEGIN
          RETURN SELF.suma;
```

```
    END getSuma;
END;
/

CREATE OR REPLACE TYPE BODY CM AS
MEMBER FUNCTION getSumaCAS RETURN NUMBER IS
        obj_angajat ANGAJAT;
    BEGIN
        UTL_REF.SELECT_OBJECT(SELF.ref_angajat,obj_angajat);
    -- considerăm ziua de CM de  8 ore
        RETURN obj_angajat.getSalOrar()*8*SELF.procentSal*SELF.nrZilePlatiteCAS;
    EXCEPTION
        WHEN OTHERS THEN RETURN -1;
    END getSumaCAS;
MEMBER FUNCTION getSumaFirma RETURN NUMBER IS
        obj_angajat ANGAJAT;
    BEGIN
        UTL_REF.SELECT_OBJECT(SELF.ref_angajat,obj_angajat);
        RETURN obj_angajat.getSalOrar()*8*SELF.procentSal*SELF.nrZilePlatiteFirma;
    EXCEPTION
        WHEN OTHERS THEN RETURN -1;
    END getSumaFirma;


OVERRIDING MEMBER FUNCTION getSuma RETURN NUMBER IS
        obj_angajat ANGAJAT;
    BEGIN
        UTL_REF.SELECT_OBJECT(SELF.ref_angajat,obj_angajat);
        RETURN SELF.getSumaCAS()+SELF.getSumaFirma();
    EXCEPTION
        WHEN OTHERS THEN RETURN -1;
    END getSuma;
END;
/
```

În listingul de mai sus, pentru calculul sporului de vechime este necesară tabela `TRANSE_SV` a cărei structură a fost definită în capitolul 4. De asemenea se observă utilizarea pachetului `UTL_REF` pentru a obţine unui obiect de tip `Angajat` pe baza referinţei stocate de atributul `ref_Angajat` (aşa cum am precizat în unul din paragrafele anterioare, PL/SQL nu oferă deocamdată suport direct pentru navigarea între obiecte prin referinţe). În plus, ne luăm rămas bun de la vechiul tip `Angajat` şi tabelele care l-au folosit.

Listing 13.7. Tabele pentru stocarea obiectelor de tip Angajat şi Drepturi

```
CREATE TABLE angajati_objtbl OF ANGAJAT;

CREATE TABLE drepturi_tbl (
 Luna INTEGER,
 An INTEGER,
 dreptAcordat DREPTURI
 );

INSERT INTO angajati_objtbl VALUES (
            NEW Angajat(111,'Angajat',' 1',140,100000,120000,0));
INSERT INTO angajati_objtbl VALUES (
            NEW Angajat(112,'Angajat',' 2',18,80000,100000,0.15));
```

```
INSERT INTO angajati_objtbl VALUES (
            NEW AngajatAcord(113,'Angajat_Acord',' 3',18,100000,120000,0,1.2));

INSERT INTO angajati_objtbl VALUES (
            NEW AngajatAcord(114,'Angajat_Acord',' 4',18,80000,100000,0.1,0.8));

-- drepturi pentru angajatul 111
INSERT INTO drepturi_tbl (SELECT 03,2003,NEW Lucrate(REF(A),'ore lucrate',100)
                                FROM angajati_objtbl A WHERE A.marca=111);
INSERT INTO drepturi_tbl (SELECT 03,2003,NEW SporVechime(REF(A),'sp vech',160)
                                FROM angajati_objtbl A WHERE A.marca=111);

-- drepturi pentru angajatul 112
INSERT INTO drepturi_tbl (SELECT 03,2003,NEW CM(REF(A),'c. med',3,7,10,0.85)
                                FROM angajati_objtbl A WHERE A.marca=112);

-- drepturi pentru angajatul 113
INSERT INTO drepturi_tbl (SELECT 03,2003,NEW Lucrate(REF(A),'ore lucrate',100)
                                FROM angajati_objtbl A WHERE A.marca=113);
INSERT INTO drepturi_tbl (SELECT 03,2003,NEW SporNoapte(REF(A),'sp noapte',0.1,80)
                                FROM angajati_objtbl A WHERE A.marca=113);
INSERT INTO drepturi_tbl (SELECT 03,2003,NEW CO(REF(A),'zile CO',8)
                                FROM angajati_objtbl A WHERE A.marca=113);
```

Listingul 13.7 evidenţiază un alt aspect esenţial legat de ierarhiile de tipuri şi anume: ***o tabelă obiectuală sau un atribut declarate de un anumit tip abstract vor permite şi stocarea de obiecte aparţinând tipurilor derivate***. Se observă că în tabela `Angajati_ObjTbl` au fost adăugate două obiecte de tip `AngajatAcord` iar în tabela `Drepturi_Tbl` valorile atributului `DreptAcorda`t reprezintă obiecte aparţinând subtipurilor tipului `Drepturi`. Acesta este cât se poate de natural pentru că tipul `SporNoapte`, spre exemplu, nu este altceva decât un alt fel (o speţă) de `Drepturi`.

```
  1  select D.dreptacordat.ref_angajat.nume ||  D.dreptacordat.ref_angajat.prenume as nume,
  2         D.dreptacordat.eticheta as drepturi, D.dreptacordat.ref_angajat.getSalOrar() as salOrar,
  3         D.dreptacordat.getSuma() as suma
  4*     from drepturi_tbl D
SQL> /

NUME                 DREPTURI              SALORAR       SUMA
-------------------- -------------------- ---------- ----------
Angajat 1            ore lucrate              100000   10000000
Angajat 1            sp vech                  100000          0
Angajat 2            c. med                    68000    7860800
Angajat_Acord 3      ore lucrate              120000   12000000
Angajat_Acord 3      sp noapte                120000     960000
Angajat_Acord 3      zile CO                  120000    7680000

5 rows selected.
```

Figura 13.13. Comportament polimorfic al metodelor getSuma() şi getSalOrar()

Figura 13.13 ilustrează comportamentul polimorfic al metodei `getSuma()` (şi în mod indirect şi al metodei `getSalOrar()` în cazul angajatului 3 care este de tip `AngajatAcord`). Astfel, fiecare obiect din subtipurile derivate din `Drepturi` va calcula suma aferentă după algoritmul propriu definit la suprascrierea metodei `getSuma()`. Pentru că polimorfismul în cazul tipului `Angajat` nu este la fel de evident, să analizăm mai atent angajatul 1 şi angajatul 3 care au acelaşi salariu şi

acelaşi număr de ore lucrate dar angajatul 3 beneficiază de o sumă mai mare. Această situaţie se dotorează faptului că angajatul 3 este de tip `AngajatAcord` şi beneficiază de un coeficient de realizare de 1,2 (conform valorilor din listing 13.7). Metoda `getSuma()`, definită în tipul `Lucrate`, invocă metoda `getSalOrar()` a obiectului adresat prin atributul `ref_angajat` (în cazul nostru angajatul 3). Astfel se va obţine un salariu orar calculat diferit (înmulţit cu procentul de realizare) faţă de cazul unui obiect de tip `Angajat` (unde metoda `getSalOrar()` returnează salariul orar mai puţin o eventuală penalizare).

13.4.2. Identificarea tipurilor în momentul execuţiei. Funcţiile TREAT, SYS_TYPEID şi IS OF

O atenţie deosebită trebuie acordată faptului că obiectele stocate în tabele sau atribute sunt *tratate* în mod implicit ca fiind de tipul declarat al tabelei sau atributului respectiv. Astfel, o interogare de genul celei din figura 13.14 va genera o eroare prin care suntem înştiinţaţi că atributul `Nrore` nu este recunoscut ca aparţinând tipului `Drepturi`. Cu alte cuvinte, motorului SQL trebuie să i se precizeze dacă obiectul stocat trebuie tratat ca fiind de tipul declarat în tabelă (supertipul) sau de tipul său original (în exemplul din figură, tipul original ar fi `Lucrate`).

```
SQL> select D.dreptacordat.nrore from drepturi_tbl D;
select D.dreptacordat.nrore from drepturi_tbl D
       *
ERROR at line 1:
ORA-00904: "D"."DREPTACORDAT"."NRORE": invalid identifier
```

Figura 13.14. Incercare de acces la valoarea unui atribut ce aparţine unui obiect de un tip derivat al celui declarat în tabelă

Pentru identificarea tipului original al obiectelor şi manipularea lor în consecinţă, Oracle furnizează câteva funcţii, după cum urmează:

- `TREAT (obiect AS Tip)` – specifică tipul ;
- `IS OF (Tip)` - verifică tipul de care aparţine un obiect;
- `SYS_TYPEID (obiect)` – returnează identificatorul unic al tipului (fiecărui tip îi este asociat un identificator unic în baza de date). O variabilă PL/SQL ce ar trebui să preia acest identificator trebuie declarată tip `RAW(16)`.

În figura 13.15, ce ilustrează valorificarea acestor funcţii, observăm că în cazul tabelelor obiectuale (`Angajat_ObjTbl`) trebuie folosită funcţia `VALUE()`[3] pentru a obţine obiecte „pasate" apoi ca argument al funcţiei `TREAT`.

[3] vezi paragraful referitor la stocarea şi gestionarea obiectelor în BD pentru detalii despre funcţia `Value()`

```
SQL>  select (TREAT(D.dreptacordat AS Lucrate)).nrore from drepturi_tbl D
  2      where D.dreptacordat IS OF (Lucrate);

(TREAT(D.DREPTACORDATASLUCRATE)).NRORE
--------------------------------------
                                   100
                                   100

SQL>  select D.Dreptacordat.eticheta, SYS_TYPEID(D.dreptacordat) from drepturi_tbl

DREPTACORDAT.ETICHET SYS_TYPEID(D.DREPTACORDAT)
-------------------- ------------------------------
ore lucrate          04
sp vech              02
c. med               08
ore lucrate          04
sp noapte            03
zile CO              05

6 rows selected.

SQL> select TREAT(A as AngajatAcord) from Angajati_objTbl A where A IS OF (AngajatAcord);
select TREAT(A as AngajatAcord) from Angajati_objTbl A where A IS OF (AngajatAcord)
                                                             *
ERROR at line 1:
ORA-00904: "A": invalid identifier

SQL>  select TREAT(VALUE(A) as AngajatAcord) from Angajati_objTbl A
  2     where VALUE(A) IS OF (AngajatAcord);

TREAT(VALUE(A)ASANGAJATACORD)(MARCA, NUME, PRENUME, NRLUNIVECHIME, SALORAR, SALORARCO, PROCENTPENALI
-----------------------------------------------------------------------------------------------
ANGAJATACORD(113, 'Angajat_Acord', ' 3', 18, 100000, 120000, 0, 1.2)
ANGAJATACORD(114, 'Angajat_Acord', ' 4', 18, 80000, 100000, .1, .8)
```

Figura 13.15. Funcţii pentru identificarea tipurilor.

În cazul tipurilor-referinţă (`REF`) sunt necesare prelucrări suplimentare (vezi şi figura 13.16) după cum urmează:

- funcţia `TREAT` va trebui să returneze tot o referinţă, astfel că vom folosi sintaxa: `TREAT (referinta_Tip AS REF Tip);`
- operatorul `IS OF` "lucrează" numai cu obiecte, astfel că vom utiliza funcţia `DEREF(referinta)` pentru a obţine obiectul adresat prin `referinta`.

```
SQL>  select (TREAT(D.dreptacordat.ref_angajat AS AngajatAcord)).procentRealizat as procRealiz
  2   from drepturi_tbl D where D.dreptacordat.ref_angajat IS OF (AngajatAcord);
 select (TREAT(D.dreptacordat.ref_angajat AS AngajatAcord)).procentRealizat as procRealiz
         *
ERROR at line 1:
ORA-00932: inconsistent datatypes: expected REF LIVIU2.ANGAJAT got LIVIU2.ANGAJATACORD


SQL>   select (TREAT(D.dreptacordat.ref_angajat AS REF AngajatAcord)).procentRealizat as procRealiz

  2   from drepturi_tbl D where DEREF(D.dreptacordat.ref_angajat) IS OF (AngajatAcord);

PROCREALIZ
----------
       1.2
       1.2
       1.2

SQL>
```

Figura 13.16. Tipurile REF necesită un tratament special în funcţiile TREAT şi IS OF

13.4.3. Constructori

Paradigma orientată-obiect defineşte constructorul ca fiind un tip special de funcţie-membru(metodă), *cu acelaşi nume ca şi al tipului*, ce se declanşează la momentul instanţierii unui tip, returnează un obiect de tipul respectiv şi prin intermediul căreia pot fi efectuate operaţiuni de iniţializare a obiectului. La fel ca în multe limbaje de programare orientată-obiect, în Oracle există un constructor implicit pentru fiecare tip declarat, constructor ce este apelat la crearea unui obiect nou după următorul format:

```
NEW DenumireTip(val_atrib1, …, val_atribn)
```

unde `val_atrib1..n` reprezintă valori pentru *fiecare* atribut al tipului.

Există însă suficient de multe situaţii în care, fie se doreşte iniţializarea unui obiect cu valori implicite, fie sunt necesare operaţii suplimentare înainte de iniţializarea obiectului. Aceste deziderate pot fi rezolvate prin definirea unor *constructori proprii*, prin supraîncărcarea constructorului implicit, astfel:

```
CREATE TYPE DenumireTip AS OBJECT (
…
CONSTRUCTOR FUNCTION DenumireTip(<lista_parametri>)
    RETURN SELF AS RESULT
…          )
/

CREATE TYPE BODY DenumireTip IS
…
CONSTRUCTOR FUNCTION DenumireTip(<lista_parametri>)
                      RETURN SELF AS RESULT IS
BEGIN
          …
          RETURN;
END DenumireTip;
END;
```

Se observă că în definiţia corpului funcţiei-constructor instrucţiunea `RETURN` nu este urmată de nimic, nici măcar de parametrul implicit `SELF`.

Important! Constructorii nu sunt moşteniţi de subtipuri, aşa încât va trebui să definim câte unul pentru fiecare subtip în parte.

Spre exemplu, pentru tipul `Drepturi` definit anterior am dori ca la instanţierea unui obiect nou :

- atributul `eticheta` să preia o valoare implicită;
- pentru atributul `ref_angajat` să furnizăm doar marca angajatului iar extragerea referinţei spre obiectul corespunzător din tabela `Angajati_objTbl` să se realizeze automat.

Listingul 13.8 prezintă secvența de cod ce modifică[4] tipul `Lucrate`, în sensul celor prezentate mai sus, prin adăugarea unui constructor cu doi parametri (număr ore şi marca angajat).

Listing 13.8. Adăugarea unei funcţii-constructor

```
ALTER TYPE Lucrate  ADD
    CONSTRUCTOR FUNCTION Lucrate(pNrOre INTEGER, pMarca INTEGER)
        RETURN SELF AS RESULT
        CASCADE INCLUDING TABLE DATA;

CREATE OR REPLACE TYPE BODY Lucrate AS
    OVERRIDING MEMBER FUNCTION getSuma RETURN NUMBER IS
        obj_angajat ANGAJAT;
    BEGIN
        UTL_REF.SELECT_OBJECT(SELF.ref_angajat,obj_angajat);
        RETURN obj_angajat.getSalOrar()*SELF.nrore;
    EXCEPTION
        WHEN OTHERS THEN RETURN -1;
    END getSuma;

CONSTRUCTOR FUNCTION Lucrate(pNrOre INTEGER, pMarca INTEGER)
    RETURN SELF AS RESULT
IS
    V_ref REF ANGAJAT;
BEGIN
    -- extragem referinţa necesară pentru atributul ref_angajat
    SELECT REF(A) INTO v_ref FROM Angajati_ObjTbl  A WHERE A.marca=pMarca;
    SELF.eticheta:= 'Ore lucrate' ;
    SELF.nrOre:=pNrOre;
    SELF.ref_Angajat:=v_ref;
    RETURN;
END Lucrate;
END;
/
```

Funcţionalitatea constructorului nou definit este redată în figura 13.17 (observaţi diferenţele între tehnica de iniţializare a unui obiect `Lucrate` din figură şi cea utilizată în listingul 13.7).

```
SQL>  insert into drepturi_tbl values (03,2003,new Lucrate(160,114));

1 row created.

SQL> column nume format A30
SQL> select D.dreptacordat.ref_angajat.Nume ||  D.dreptacordat.ref_angajat.Prenume AS nume,
  2         D.dreptacordat.eticheta as drepturi, D.dreptacordat.getSuma() as suma
  3         from drepturi_tbl D where D.dreptacordat.ref_angajat.marca=114;

NUME                           DREPTURI                  SUMA
------------------------------ -------------------- ----------
Angajat_Acord 4                Ore lucrate             9216000

SQL>
```

Figura 13.16. Obiect de tip `Lucrate` creat prin intermediul unui constructor propriu definit în listingul 13.8

[4] Pentru detalii referitoare la modificarea definiţiei tipurilor - vezi paragraful următor

13.5. Modificarea definiției tipurilor

Tipurile definite pot evolua în timp, impunându-se necesitatea modificării definiției lor prin: adăugarea de noi atribute sau metode, modificarea sau ştergerea de membri. Pentru modificarea tipurilor utilizăm instrucțiunea SQL, `ALTER TYPE`, cu următoarele clauze:

- `ADD ATTRIBUTE <atribut_nou Tip>` - adaugă atribute noi. Exemplu: `ALTER TYPE ADD ATTRIBUTE (a INTEGER, b VARCHAR2(5))`
- `ADD <specificatii_metoda>` - adaugă o metodă nouă. Exemplu: `ALTER TYPE ADD MEMBER PROCEDURE test`
- `DROP ATTRIBUTE <atribut1>`
- `DROP <specificatii_metoda>`
- `MODIFY ATTRIBUTE <atribut TipNou>`
- `COMPILE` – recompilează pentru validare un tip dependent de un alt tip care s-a modificat cu opțiunea `INVALIDATE` (vezi paragraful următor)
- `FINAL / NOT FINAL` – converteşte un tip pentru a putea fi derivat sau nu.

Datorită faptului că în majoritatea situațiilor există dependențe (moştenire, compunere) între tipuri sau între tipuri şi tabele de obiecte, câteva clauze ale comenzii `ALTER TYPE` specifică modalitatea de propagare a modificărilor spre tipurile dependente sau spre obiectele deja existente, astfel:

- `CASCADE` – determină propagarea modificării în toate tipurile dependente (atât tipurile derivate sau cele care definesc atribute de tipul respectiv, cât şi tabele);
- `INVALIDATE` – toate tipurile şi tabelele dependent for fi marcate ca invalide;
- `INCLUDING TABLE DATA` – clauză ce poate fi folosită numai în combinație cu opțiunea `CASCADE` şi prin care se asigură „upgrade-ul" obiectelor stocate conform noii definiții a tipului (atributele noi vor avea valoarea implicită declarată la definirea tipului sau valoarea `null`);
- `NOT INCLUDING TABLE DATA` – similară celei anterioare doar că obiectele deja stocate nu-şi modifică structura şi sunt marcate ca invalide; ulterior se va recurge la comanda `ALTER TABLE tabela_de_obiecte UPGRADE INCLUDING DATA` pentru a modifica structura obiectelor stocate;
- `CONVERT TO SUBSTITUTABLE` - este utilizată la conversia tipurilor în `NOT FINAL` şi asigură convertirea atributelor tabelelor dependente astfel

încât să poată stoca şi obiecte aparţinând subtipurilor tipului respectiv; dacă tipul este convertit la `NOT FINAL` fără această opţiune, tabelele dependente nu vor putea stoca obiecte aparţinând subtipurilor.

Notă. Pentru a extrage informaţii despre relaţiile de dependenţă între tipuri sau între tipuri şi tabele vom interoga tabela-sistem `USER_DEPENDENCIES`.

Aşadar, pentru a modifica un tip (spre exemplu `Angajat`) se poate recurge la una din următoarele alternative:

1. Se propagă modificările în tipurile dependente şi se actualizează obiectele din tabele:

```
ALTER TYPE Angajat
ADD ATTRIBUTE test INTEGER CASCADE INCLUDING TABLE DATA;
```

2. Se propagă modificările în tipurile dependente şi se actualizează obiectele ulterior:

```
ALTER TYPE Angajat ADD ATTRIBUTE test INTEGER
            CASCADE NOT INCLUDING TABLE DATA;
ALTER TABLE Angajati_ObjTbl UPGRADE INCLUDING DATA;
ALTER TABLE Drepturi_tbl UPGRADE INCLUDING DATA;
```

3. Fără a propaga nici o modificare, se recompilează toate tipurile dependente şi actualizează obiectele stocate în tabele la un moment ulterior:

```
ALTER TYPE Angajat ADD ATTRIBUTE test INTEGER INVALIDATE;
ALTER TYPE AngajatAcord COMPILE;
ALTER TYPE Drepturi COMPILE;
ALTER TYPE Lucrate COMPILE;
.... (procedăm similar cu toate subtipurile)

ALTER TABLE Angajati_ObjTbl UPGRADE INCLUDING DATA;
ALTER TABLE Drepturi_tbl UPGRADE INCLUDING DATA;
```

Ştergerea atributului `Test`:

```
ALTER TYPE Angajat DROP ATTRIBUTE test
          CASCADE INCLUDING TABLE DATA;
```

Important! Schema de lucru cu numărul 3 o vom utiliza atunci când variantele anterioare nu funcţionează, şi obţinem o eroare precum cea din figura 13.18. La adăugarea unei metode noi *întregul corp al tipului va trebui recompilat* prin `CREATE OR REPLACE TYPE BODY` (vezi listing 13.8 din paragrafele precedente). De asemenea trebuie acordată atenţie faptului că, odată tipurile modificate, va fi necesară o nouă sesiune de SQL*PLUS.

```
SQL> alter type Angajat add attribute test Integer CASCADE INCLUDING TABLE DATA;
alter type Angajat add attribute test Integer CASCADE INCLUDING TABLE DATA
*
ERROR at line 1:
ORA-22324: altered type has compilation errors
ORA-00600: internal error code, arguments: [4883], [0x78D19FF8], [0x7888D088],
[], [], [], [], []


SQL> alter type Angajat add attribute test Integer CASCADE NOT INCLUDING TABLE DATA;
alter type Angajat add attribute test Integer CASCADE NOT INCLUDING TABLE DATA
*
ERROR at line 1:
ORA-22324: altered type has compilation errors
ORA-00600: internal error code, arguments: [4883], [0x78D19FCC], [0x7888D088],
[], [], [], [], []
```

Figura 13.18. Erorile la modificarea tipului impun utilizarea schemei de lucru 3 prezentată mai sus

13.6. Metode de ordonare a obiectelor

Nu puține sunt situațiile în care se utilizează clauze de ordonare a datelor în frazele Select SQL sau se compară variabile în blocuri PL/SQL. În mod similar datelor scalare, obiectele pot fi ordonate după valorile conținute de atributele lor sau după rezultatul returnat de metodele-membre, aşa cum se poate observa şi în următorul exemplu:

```
SELECT * FROM angajati_objtbl A
      ORDER BY (VALUE(A)).Nume||(VALUE(A)).prenume;
```

sau

```
SELECT * FROM drepturi_tbl D
    ORDER BY D.dreptacordat.getSuma();
```

Oracle oferă însă şi posibilitatea definirii unei metode speciale prin care să se specifice un criteriu de ordonare (comparație) a obiectelor ca atare. Această metodă poate fi:

- de tip `MAP` – definită fără parametri şi care returnează o valoare scalară (`NUMBER, DATE, VARCHAR2`) ce va fi utilizată în operațiile de ordonare;
- de tip `ORDER` – defineşte un singur parametru ce preia un obiect de acelaşi tip şi returnează o valoare numerică pozitivă, zero sau negativă cu semnificația: obiectul curent este mai mare, egal sau mai mic decât cel preluat prin parametru. Recurgem la acest tip de metodă atunci când criteriul de ordonare este mai complex şi nu poate fi asociat cu un tip scalar.

Spre exemplu, pentru a ordona obiectele de tip `Angajat` după nume şi prenume se defineşte metoda `MAP` din listing 13.9.

Listing 13.9. Un exemplu de metodă de tip MAP

```
ALTER TYPE Angajat ADD MAP MEMBER FUNCTION map_nume_pren RETURN VARCHAR2
   CASCADE INCLUDING TABLE DATA;

CREATE OR REPLACE  TYPE BODY Angajat AS
      ...  -- metodele iniţiale
    MAP MEMBER FUNCTION map_nume_pren RETURN varchar2 IS
    BEGIN
        RETURN SELF.Nume || SELF.Prenume;
    END map_nume_pren;
END;
/
```

După modificare, interogarea din exemplul de mai sus poate fi rescrisă astfel:

```
SELECT * FROM angajati_objtbl A ORDER BY VALUE(A)
```

Listingul 13.10 prezintă şi un exemplu de metodă `ORDER` creată în scopul ordonării obiectelor de tip `Drepturi` în funcţie de sumă.

Listing 13.10. Metodă de tip ORDER

```
ALTER TYPE Drepturi
    ADD ORDER MEMBER FUNCTION compara (obj Drepturi)
    RETURN INTEGER
    CASCADE INCLUDING TABLE DATA;

CREATE OR REPLACE TYPE BODY Drepturi IS
     ORDER MEMBER FUNCTION compara(obj Drepturi) RETURN INTEGER IS
      BEGIN
          IF SELF.getSuma() < obj.getSuma() THEN
                    RETURN -1;          - - ar fi fost valabil orice număr întreg negativ
          ELSEIF   SELF.getSuma()=obj.getSuma() THEN
                    RETURN 0;
                    ELSE
                      RETURN 1;
          END IF;
      END compara;
 END;
```

Astfel, va fi posibilă o interogare pe tabela `Drepturi_tbl` de genul:

```
SELECT D.DreptAcordat.getSuma()
FROM drepturi_tbl D order by D.DreptAcordat;
```

Implementarea acestor metode într-o ierarhie de tipuri necesită câteva precizări suplimentare:

- pentru un anumit tip definit, poate fi declarată doar una dintre metode (fie `MAP`, fie `ORDER`), şi nu ambele;
- sortările pe bază de metode `MAP` sunt mult mai rapide pentru că sunt, în esenţă, sortări ale unor tipuri scalare, pe când metodele `ORDER` sunt apelate de multiple ori ;

- numai metodele `MAP` pot fi suprascrise de subtipuri (şi numai dacă supertipul a definit la rândul lui o asemenea operație);
- metodele `ORDER` pot fi definite numai pentru supertipurile rădăcină (cele care nu au ascendenți pe arborele ierarhiei) şi nu pot fi suprascrise de subtipuri.