Teil IV

Objektorientierung in Oracle

15 OBJEKT-RELATIONALE ERWEITERUNGEN

Objekt-Relationale Datenbanksysteme – u. a. ORACLE – vereinen relationale Konzepte und Objektorientierung. Dabei werden die folgenden Konzepte angeboten [Vos94]:

- Komplexe und Abstrakte Datentypen. Dabei werden *Value Types* und *Object Types* unterschieden. Erstere erweitern nur das Domain-Konzept von SQL2, letztere unterstützen Objekt-Identität und Kapselung interner Funktionalität.
- Spezialisierung: ein Typ kann als Subtyp eines oder mehrerer anderer Typen vereinbart werden. Dabei hat jeder Subtyp *genau* einen (direkten oder indirekten) maximalen Supertyp (der keinen weiteren Supertyp besitzt).
- Tabellenverbände als Form der Spezialisierung von Relationen: Eine Tabelle kann als Subtabelle einer oder mehrerer anderer Tabellen definiert werden.
- Funktionen als Bestandteile eines ADT's oder von Tabellen, oder freie Funktionen.
- Methoden- und Funktionsaufrufe im Rahmen von `SELECT`-Ausdrücken.

15.1 Objekte

Objektorientierung bedeutet, dass zwischen dem *Zustand* und dem *Verhalten* eines *Objektes* unterschieden wird. Objektorientierung in ORACLE 8 bedeutet, dass es neben Tabellen, deren Inhalt aus Tupels besteht, auch *Object Tables* gibt, deren Inhalt aus Objekten besteht. Ebenso können einzelne Spalten einer Tupeltabelle *objektwertig* sein. Im Gegensatz zu einem *Tupel* besitzt ein Objekt *Attribute*, die den inneren Zustand des Objektes beschreiben, sowie *Methoden*, mit denen der Zustand abgefragt und manipuliert werden kann. Einfache Objekttypen wurden bereits in Abschnitt 5.1 als *komplexe Attribute* behandelt: Komplexe Attribute sind Objekttypen, die nur *Wertattribute* und keine Methoden besitzen. Objekte können neben *Wertattributen* auch *Referenzattribute* besitzen: In diesem Fall enthält das Attribut eine Referenz auf ein anderes Objekt. Referenzattribute werden durch `REF <object-datatype>` gekennzeichnet.

Grob gesehen entsprechen Objekttabellen den Klassen in objektorientierten Programmiersprachen während Tupel den Instanzen entsprechen. Im Gegensatz zu der üblichen Vorstellung von Objektorientierung bietet ORACLE 8 keine Vererbung zwischen Klassen, Subklassen und Instanzen. Damit beschränkt sich die Klassen-/Tabellenzugehörigkeit darauf, gemeinsame Methoden aller zugehörigen Objekte zu definieren. Als Methoden stehen *Prozeduren* und *Funktionen* zur Verfügung, wobei eine Funktion durch `MAP` oder `ORDER` ausgezeichnet werden kann um eine Ordnung auf den Objekten einer Klasse zu definieren (anderenfalls sind nur Tests auf Gleichheit erlaubt). Die *Signaturen* der Methoden sowie ihre `READ/WRITE` Zugriffscharakteristik werden in der `TYPE`-Deklaration angegeben, die Implementierung der Methoden ist als PL/SQL-Programm im `TYPE BODY` gegeben.

Die allgemeine Form einer Objekttypdeklaration sieht damit – grob – so aus:

```
CREATE [OR REPLACE] TYPE <type> AS OBJECT
  (  <attr> <datatype>,
     ⋮
     <attr> REF <object-datatype>,
     ⋮
     MEMBER FUNCTION <func-name> [(<parameter-list>)] RETURN <datatype>,
     ⋮
     MEMBER PROCEDURE <proc-name> [(<parameter-list>)],
     ⋮
     [ MAP MEMBER FUNCTION <func-name> RETURN <datatype>, |
       ORDER MEMBER FUNCTION <func-name>(<var> <type>) RETURN <datatype>,]
     [ <pragma-declaration-list> ]
     );
```

Dabei muss `<object-datatype>` ebenfalls durch einen Befehl der Form `CREATE TYPE <object-datatype> AS OBJECT ...` als Objekttyp deklariert sein. `<parameter-list>` hat dieselbe Syntax wie bereits in Abschnitt 14.1 für Prozeduren und Funktionen angegeben. Der erste Teil der Definition, in dem die Attribute festlegt werden ist der Definition von Tabellen durch `CREATE TABLE` (vgl. Abschnitt 2.1) sehr ähnlich – es ist jedoch zu beachten, dass bei `CREATE TYPE` *keine* Integritätsbedingungen angegeben werden können (diese werden erst bei der (Objekt)tabellen-Definition angegeben).

`PRAGMA`-Klauseln. In `<pragma-declaration-list>` kann für jede Methode eine `PRAGMA`-Klausel der Form

```
PRAGMA RESTRICT_REFERENCES (<method_name>,<feature-list>));
```

angegeben werden, die spezifiziert, ob eine Prozedur/Funktion schreibend/lesend auf die Datenbank zugreifen darf. `<feature-list>` ist eine Komma-Liste mit den möglichen Einträgen

`WNDS`	Writes no database state,
`WNPS`	Writes no package state,
`RNDS`	Reads no database state,
`RNPS`	Reads no package state.

Diese Angabe ist insbesondere für Funktionen wichtig, da ORACLE diese nur ausführt, wenn *zugesichert* ist, dass sie den Datenbankzustand nicht verändern. Daher muss bei Funktionen zumindest

```
PRAGMA RESTRICT_REFERENCES (<function_name>,WNPS,WNDS);
```

gesetzt werden.

`MAP`- und `ORDER`-Funktionen erfordern

```
PRAGMA RESTRICT_REFERENCES (<function-name>,WNDS,WNPS,RNPS,RNDS)
```

d.h. sie dürfen *keinen Datenbankzugriff* enthalten (was sich als sehr strenge Einschränkung herausstellt).

Beispiel 20 (Objekttyp *GeoCoord*) Der komplexe Attributtyp *GeoCoord* (vgl. Beispiel 6) kann wie folgt um eine Methode *Distance* erweitert werden, die die Entfernung zu einem gegebenen zweiten *GeoCoord*-Wert berechnet. Außerdem wird die Entfernung von Greenwich als `MAP`-Methode deklariert:

```
CREATE OR REPLACE TYPE GeoCoord AS OBJECT
(Longitude NUMBER,
 Latitude NUMBER,
 MEMBER FUNCTION Distance (other IN GeoCoord) RETURN NUMBER,
 MAP MEMBER FUNCTION Distance_Greenwich RETURN NUMBER,
 PRAGMA RESTRICT_REFERENCES (Distance,WNPS,WNDS,RNPS,RNDS));
 PRAGMA RESTRICT_REFERENCES (Distance_Greenwich,WNPS,WNDS,RNPS,RNDS));
/
```

Der `TYPE BODY` enthält die Implementierung der Objektmethoden in PL/SQL. Für jedes Objekt ist automatisch die Methode `SELF` definiert, mit der das Objekt selber als Host-Objekt einer Methode angesprochen werden kann. `SELF` wird in Methodendefinitionen verwendet, um auf die Attribute des Host-Objektes zuzugreifen.

Die Definition des `TYPE BODY` muss der bei `CREATE TYPE` vorgegeben Signatur desselben Typs entsprechen. Insbesondere muss für *alle* deklarierten Methoden eine Implementierung angegeben werden.

```
CREATE [OR REPLACE] TYPE BODY <type>
AS
     MEMBER FUNCTION <func-name> [(<parameter-list>)] RETURN <datatype>
     IS
        [<var-decl-list>;]
        BEGIN <PL/SQL-code> END;
     ⋮
     MEMBER PROCEDURE <proc-name> [(<parameter-list>)]
     IS
        [<var-decl-list>;]
        BEGIN <PL/SQL-code> END;
     ⋮
     [ MAP MEMBER FUNCTION <func-name> RETURN <datatype> |
       ORDER MEMBER FUNCTION <func-name>(<var> <type>) RETURN <datatype>
      IS
       [<var-decl-list>;]
       BEGIN <PL/SQL-code> END;]
END;
/
```

Beispiel 21 (Objekttyp *GeoCoord* (Forts.)) Der `TYPE BODY` zu *GeoCoord* enthält nun die Implementierung der Methoden *Distance* und *Distance_ Greenwich*:

```
CREATE OR REPLACE TYPE BODY GeoCoord
AS
MEMBER FUNCTION Distance (other IN GeoCoord)
```

```
  RETURN NUMBER
  IS
  BEGIN
   RETURN 6370*ACOS(cos(SELF.latitude/180*3.14)*cos(other.latitude/180*3.14)
              *cos((SELF.longitude-other.longitude)/180*3.14)
              + sin(SELF.latitude/180*3.14)*sin(other.latitude/180*3.14));
  END;
MAP MEMBER FUNCTION Distance_Greenwich
  RETURN NUMBER
  IS
  BEGIN
   RETURN SELF.Distance(GeoCoord(0,51));
  END;
END;
/
```

Methoden eines Objektes werden mit

```
<object>.<method-name>(<argument-list>)
```

aufgerufen.

Zeilen- und Spaltenobjekte. Im weiteren unterscheidet man *Zeilenobjekte* und *Spaltenobjekte*: Zeilenobjekte sind Elemente von *Objekttabellen*. Spaltenobjekte erhält man, wenn ein Attribut eines Tupels (oder eines Objekts) objektwertig ist (vgl. Abschnitt 5.1). Während Zeilenobjekte eine eindeutige OID erhalten (siehe Abschnitt 15.3) über die sie *referenzierbar* sind, haben Spaltenobjekte *keine* OID, sind also *nicht* referenzierbar. Bei der Tabellendefinition werden wie üblich auch Integritätsbedingungen angegeben.

Wie bereits in Abschnitt 5.1 werden Zeilen- und Spaltenobjekte mit Hilfe der entsprechenden *Konstruktormethode* erzeugt. Zu beachten ist hierbei, dass die für einen n-stelligen Konstruktor auch n Argumente angegeben werden. Wenn die Werte zum Zeitpunkt der Objekterzeugung noch nicht bekannt sind, müssen `NULL`werte angegeben werden.

Bei `SELECT`-Statements müssen immer Tupel- bzw. Objektvariablen durch Aliasing verwendet werden, wenn ein *Zugriffspfad* `<var>.<method>*[.<attr>]` angegeben wird – also wenn Attribute oder Methoden von objektwertigen Attributen selektiert werden.

15.2 Spaltenobjekte

Spaltenobjekte erhält man, indem man ein Attribut einer Tabelle (Tupel- oder Objekttabelle) objektwertig definiert (auch komplexe Attribute wie in Abschnitt 5.1 definieren Spaltenobjekte).

Beispiel 22 (Spaltenobjekte) Der in Beispiel 20 erzeugte komplexe Attributtyp *GeoCoord* wird wie bereits in Beispiel 6 in der Definition der Tabelle *Mountain* verwendet, um *Spaltenobjekte* zu erzeugen:

```
CREATE TABLE Mountain
(Name VARCHAR2(20) CONSTRAINT MountainKey PRIMARY KEY,
```

```
Height NUMBER CONSTRAINT MountainHeight
    CHECK (Height > = 0),
Coordinates GeoCoord CONSTRAINT MountainCoord
    CHECK ((Coordinates.Longitude >= -180) AND
           (Coordinates.Longitude <= 180) AND
           (Coordinates.Latitude >= -90) AND
           (Coordinates.Latitude <= 90)));
```

Spaltenobjekte werden nun mit Hilfe der entsprechenden Konstruktormethode erzeugt:

```
INSERT INTO Mountain VALUES ('Feldberg', 1493, GeoCoord(8,48));
```

Die Entfernung eines Berges zum Nordpol erhält man dann mit

```
SELECT Name, mt.coordinates.Distance(GeoCoord(0,90))
FROM Mountain mt;
```

unter Verwendung der Tupelvariablen `mt` um den Zugriffspfad zu *coordinates.Distance* eindeutig zu machen. □

Enthält ein in einer objektwertigen Spalte `<attr>` verwendeter Objekttyp `<object-type>` eine tabellenwertiges Attribut `<tab-attr>`, so muss bei der `NESTED TABLE ... STORE AS`-Klausel im `CREATE TABLE`-Statement ebenfalls ein Pfadausdruck verwendet werden:

```
CREATE TABLE ...
  (  ...
     <attr> <object-type>
     ...)
  NESTED TABLE <attr>.<tab-attr> STORE AS <name>;
```

15.3 Zeilenobjekte

Objekttabellen enthalten im Gegensatz zu den "klassischen" relationalen Tupeltabellen keine Tupel, sondern Objekte. Innerhalb jeder Objekttabelle besitzt jedes Objekt eine eindeutige *OID (Objekt-ID)*, die dem *Primärschlüssel* im relationalen Modell entspricht. Dieser wird mit den (weiteren) Integritätsbedingungen bei der Tabellendefinition angegeben. Erwähnenswert ist hierbei die problemlose Integration referentieller Integritätsbedingungen von Objekttabellen zu bestehenden relationalen Tabellen durch die Möglichkeit, Fremdschlüsselbedingungen in der gewohnten Syntax anzugeben.

```
CREATE TABLE <name> OF <object-datatype>
[(<constraint-list>)];
```

`<constraint-list>` ist dabei eine Liste von attributbezogenen Bedingungen sowie Tabellenbedingungen – letztere haben dieselbe Syntax wie bei Tupeltabellen. Attributbedingungen entsprechen den Spaltenbedingungen `<column-constraint>` bei Tupeltabellen und sind von der Form

```
<attr-name> [DEFAULT <value>]
   [<colConstraint> ... <colConstraint>]
```

Beispiel 23 (Objekt-Tabelle *City*) Als Beispiel wird eine Objekttabelle definiert, die alle Städte enthält, und wie oben den Datentyp *GeoCoord* verwendet. Der Objekttyp *City_ Type* wird analog zu der Tupeltabelle *City* definiert:

```
CREATE OR REPLACE TYPE City_Type AS OBJECT
 (Name VARCHAR2(35),
  Province VARCHAR2(32),
  Country VARCHAR2(4),
  Population NUMBER,
  Coordinates GeoCoord,
 MEMBER FUNCTION Distance (other IN City_Type) RETURN NUMBER,
 PRAGMA RESTRICT_REFERENCES (Distance,WNPS,WNDS,RNPS,RNDS));
/
```

Die Methode *Distance(city)* dient dazu, die Entfernung zu einer anderen Stadt zu berechnen. Sie basiert auf der Methode *Distance(geo_ coord)* des Datentyps *Geo_ Coord*. Auch hier wird wieder `SELF` verwendet, um auf die Attribute des Host-Objekts zuzugreifen:

```
CREATE OR REPLACE TYPE BODY City_Type
AS
 MEMBER FUNCTION Distance (other IN City_Type)
 RETURN NUMBER
 IS
 BEGIN
   RETURN SELF.coordinates.Distance(other.coordinates);
 END;
END;
/
```

Die Objekttabelle wird nun entsprechend definiert, wobei der (mehrspaltige) Primärschlüssel als Tabellenbedingung angegeben werden kann. Der Primärschlüssel darf keine `REF`-Attribute enthalten (vgl. Abschnitt 15.4).[1] Die Fremdschlüsselbedingung für *Country* wird ebenfalls als Tabellenbedingung angegeben:

```
CREATE TABLE City_ObjTab OF City_Type
 (PRIMARY KEY (Name, Province, Country),
  FOREIGN KEY (Country) REFERENCES Country(Code));
```

Objekte werden unter Verwendung des Objektkonstruktors `<object-datatype>` in Objekttabellen eingefügt.

[1] Dies ist nicht so richtig gut, da *schwache Entitätstypen* eine Schlüsselattribut von einem anderen Entitätstyp importieren. Diese Beziehung wird i.a. durch Referenzen modelliert.

so, if you have a scoped ref – you can in fact index uniquely the columns. For some unknown reason to me – they will not let you create a primary key (simply a `NOT NULL` constraint plus a unique index), nor a `UNIQUE` constraint (simply a unique index). I cannot figure out why not. But anyway, except for losing the fact that name, a_ref, b_ref is a primary key in the data dictionary – the above lets you accomplish what you want. You cannot index an unscoped ref (why? good question – i don't know) and thats documented in the server concepts manual (page 11-8) □

Beispiel 24 (Objekt-Tabelle *City* (Forts.)) Die Objekttabelle *City_ ObjTab* wird wie folgt aus der Tupeltabelle *City* (z. B. mit allen deutschen Städten, deren geographische Koordinaten bekannt sind) gefüllt:

```
INSERT INTO City_ObjTab
SELECT City_Type(Name,Province,Country,Population,GeoCoord(Longitude,Latitude))
FROM City
WHERE Country = 'D'
  AND NOT Longitude IS NULL;
```

Will man ein selektiertes Zeilenobjekt *als Ganzes* ansprechen (etwa für einen Vergleich oder in einer `ORDER BY`-Klausel), so muss man `VALUE (<var>)` verwenden:

Beispiel 25 (Objekt-Tabelle *City* (Forts.)) Gibt man `VALUE (<var>)` aus, erhält man die bekannte Darstellung mit Hilfe des Objektkonstruktors:

```
SELECT VALUE(cty)
FROM City_ObjTab cty;
```

VALUE(Cty)(Name, Province, Country, Population, Coordinates(Longitude, Latitude))
City_Type('Berlin', 'Berlin', 'D', 3472009, GeoCoord(13, 52))
City_Type('Bonn', 'Nordrhein-Westfalen', 'D', 293072, GeoCoord(8, 50))
City_Type('Stuttgart', 'Baden-Wuerttemberg', 'D', 588482, GeoCoord(9, 49))
⋮

□

`VALUE` kann ebenfalls verwendet werden, um in der obigen Anfrage zwei Objekte auf Gleichheit zu testen, oder um ein Objekt als Argument an eine Methode zu geben. Im Beispiel kann man die Distanz zwischen je zwei Städten entweder über die Methode *Distance* des Attributs *coordinates* oder direkt über die Methode *Distance* des Objekttyps *City* berechnen. Wieder werden Objektvariablen `cty1` und `cty2` verwendet, um die Zugriffspfade eindeutig zu machen:

```
SELECT cty1.Name, cty2.Name, cty1.coordinates.Distance(cty2.coordinates)
FROM City_ObjTab cty1, City_ObjTab cty2
WHERE NOT VALUE(cty1) = VALUE(cty2);

SELECT cty1.Name, cty2.Name, cty1.Distance(VALUE(cty2))
FROM City_ObjTab cty1, City_ObjTab cty2
WHERE NOT VALUE(cty1) = VALUE(cty2);
```

Wird ein Objekt mit einem `SELECT INTO`-Statement einer PL/SQL-Variablen zugewiesen, wird ebenfalls `VALUE` verwendet.

```
SELECT VALUE(<var>) INTO <PL/SQL-Variable>
FROM <tabelle> <var>
WHERE ... ;
```

15.4 Objektreferenzen

Als Datentyp für Attribute sind neben den bereits bekannten skalaren Typen, Collection-Typen und Objekttypen auch *Referenzen* auf Objekte möglich. Eine entsprechende Attributdeklaration ist dann von der Form

```
<ref-attr> REF <object-datatype>
```

Dabei wird ein *Objekttyp* als Ziel der Referenz angegeben. Wenn ein referenzierter Objekttyp in verschiedenen Tabellen vorkommt, wird damit das Ziel der Referenz hier nicht auf eine bestimmte Tabelle beschränkt. Einschränkungen dieser Art werden bei der Deklaration der entsprechenden Tabelle als Spalten- oder Tabellenconstraints mit `SCOPE` angegeben:

- als Spaltenconstraint (nur bei Tupeltabellen):
  `<ref-attr> REF <object-datatype> SCOPE IS <object-table>`
- als Tabellenconstraint:
  `SCOPE FOR (<ref-attr>) IS <object-table>`

Hierbei ist zu berücksichtigen, dass nur Objekte, die eine OID besitzen – also Zeilenobjekte einer Objekttabelle – referenziert werden können. Die OID eines zu referenzierenden Objektes wird folgendermaßen selektiert:

```
SELECT ..., REF (<var>), ...
FROM <tabelle> <var>
WHERE ... ;
```

Beispiel 26 (Objekttyp *Organization*) Politische Organisationen können als Objekte modelliert werden: Eine Organisation besitzt einen Namen, eine Abkürzung, einen Sitz in einer bestimmten Stadt (die in Beispiel 23 ebenfalls als Objekt modelliert wird und damit als *Referenzattribut* gespeichert werden kann) und hat eine Liste von Mitgliedern (hier bezeichnet durch die jeweiligen Landeskürzel).

```
CREATE TYPE Member_Type AS OBJECT
 (Country VARCHAR2(4),
  Type VARCHAR2(30));
/
CREATE TYPE Member_List_Type AS
 TABLE OF Member_Type;
/
CREATE OR REPLACE TYPE Organization_Type AS OBJECT
 (Name VARCHAR2(80),
  Abbrev VARCHAR2(12),
  Members Member_List_Type,
  Established DATE,
  seated_in REF City_Type,
  MEMBER FUNCTION is_member (the_country IN VARCHAR2) RETURN VARCHAR2,
  MEMBER FUNCTION people RETURN NUMBER,
  MEMBER FUNCTION number_of_members RETURN NUMBER,
  MEMBER PROCEDURE add_member
       (the_country IN VARCHAR2, the_type IN VARCHAR2),
```

```
  PRAGMA RESTRICT_REFERENCES (is_member,WNPS,WNDS),
  PRAGMA RESTRICT_REFERENCES (people,WNDS,WNPS);
  PRAGMA RESTRICT_REFERENCES (number_of_members,WNDS,WNPS));
/
```

Die entsprechende Tabellendefinition sieht dann folgendermaßen aus:

```
CREATE TABLE Organization_ObjTab OF Organization_Type
 (Abbrev PRIMARY KEY,
  SCOPE FOR (seated_in) IS City_ObjTab)
 NESTED TABLE Members STORE AS Members_nested;
```

Das Einfügen in Objekttabellen geschieht nun wie üblich unter Verwendung des Objektkonstruktors:

```
INSERT INTO Organization_ObjTab VALUES
 (Organization_Type('European Community','EU',
  Member_List_Type(),NULL,NULL));
```

Die geschachtelte Tabelle *Members* wird ebenfalls wie bereits bekannt mit Werten gefüllt:

```
INSERT INTO
 THE(SELECT Members
     FROM Organization_ObjTab
     WHERE Abbrev='EU')
 (SELECT Country, Type
  FROM is_member
  WHERE Organization = 'EU');
```

Will man eine *Referenz* (also eine OID) auf ein Objekt, das durch eine Objektvariable beschrieben wird, verarbeiten, muss man `SELECT REF(<var>)` angeben (Man beachte, dass `<var>` selber also weder das Objekt, noch eine Referenz ist!). `SELECT REF(VALUE(<var>))` ergibt keine OID, sondern eine Fehlermeldung.

Beispiel 27 (Objekttyp *Organization* (Forts.)) Um das Referenzattribut *seated_in* auf eine Stadt zeigen zu lassen, muss die entsprechende Referenz in *City_ ObjTab* gesucht werden:

```
UPDATE Organization_ObjTab
SET seated_in =
   (SELECT REF(cty)
    FROM City_ObjTab cty
    WHERE Name = 'Brussels'
      AND Province = 'Brabant'
      AND Country = 'B')
WHERE Abbrev = 'EU';
```

Die Selektion von Wertattributen erfolgt ebenfalls wie bei gewöhnlichen relationalen Tabellen – Aliasing muss nur zur Selektion von Methoden oder objektwertigen Attributen verwendet werden:

```
SELECT Name, Abbrev, Members, seated_in
FROM Organization_ObjTab;
```

gibt *Members* in der bekannten Darstellung mit Hilfe des Objektkonstruktors aus:

Name	Abbrev	Members
European Community	EU	Member_List_Type(...)

Bei Referenzattributen bekommt man auf ein gewöhnliches `SELECT <ref-attr-name>` (erwartungsgemäß) eine Objekt-ID geliefert (die zum internen Gebrauch innerhalb von SQL oft benötigt wird, jedoch in der Benutzerausgabe nicht erwünscht ist):

```
SELECT Name, Abbrev, seated_in
FROM Organization_ObjTab;
```

Name	Abbrev	seated_in
European Community	EU	<oid>

Hat man eine solche Referenz gegeben, erhält man mit `DEREF(<oid>)` das zugehörige Objekt:

```
SELECT Abbrev, DEREF(seated_in)
FROM Organization_ObjTab;
```

Abbrev	seated_in
EU	City_Type('Brussels', 'Brabant', 'B', 951580, GeoCoord(4, 51))

Die einzelnen Attribute eines referenzierten Objekts kann man durch *Pfadausdrücke* der Form `SELECT <ref-attr-name>.<attr-name>` adressieren ("*navigierender Zugriff*"). Dabei muss Aliasing mit einer Variablen verwendet werden um den Pfadausdruck eindeutig zu machen:

```
SELECT Abbrev, org.seated_in.name
FROM Organization_ObjTab org;
```

Abbrev	seated_in.Name
EU	Brussels

`DEREF` wird insbesondere in PL/SQL-Routinen (z. B. `SELECT DEREF(...) INTO ...`) und bei der Definition von Objekt-Views(vgl. Abschnitt 16) verwendet.

Bemerkung 2 Mit `REF` und `DEREF` lässt sich `VALUE` ersetzen:

```
SELECT VALUE(cty) FROM City_ObjTab cty;
```

und

```
SELECT DEREF(REF(cty)) FROM City_ObjTab cty;
```

sind äquivalent. □

Zyklische Referenzen. Häufig kommen Objekttypen vor, die sich gegenseitig referenzieren sollen: Eine Stadt liegt in einem Land, andererseits ist die Hauptstadt eines Landes wieder eine Stadt. In diesem Fall benötigt die Deklaration jedes Datentypen bereits die Definition des anderen. Zu diesem Zweck erlaubt ORACLE die Definition von *unvollständigen* Typen (auch als *Forward*-Deklaration bekannt) durch

```
CREATE TYPE <name>;
/
```

Eine solche unvollständige Deklaration kann in `REF`-Deklarationen verwendet werden. Sie wird später durch eine komplette Typdeklaration ersetzt.

Beim Füllen dieser Tabellen muss man ebenfalls schrittweise vorgehen: Da jedes Tupel einer Tabelle eine Referenz auf ein Tupel einer anderen Tabelle enthält, muss man auch die Objekte zuerst "unvollständig" erzeugen, d.h., Objektfragmente anlegen, die bereits *eindeutig identifizierbar sind*, aber *keine Referenzen auf noch nicht angelegte Objekte* enthalten. Danach kann man mit `UPDATE` schrittweise die Tupel um Referenzen ergänzen.

Beispiel 28 (Zyklische Referenzen) Zwischen Städten, Provinzen, und Ländern bestehen zyklische Referenzen.

```
CREATE OR REPLACE TYPE City_Type
/
CREATE OR REPLACE TYPE Country_Type AS OBJECT
(Name VARCHAR2(32),
 Code VARCHAR2(4),
 Capital REF City_Type,
 Area NUMBER,
 Population NUMBER);
/
CREATE OR REPLACE TYPE Province_Type AS OBJECT
(Name VARCHAR2(32),
 Country REF Country_Type,
 Capital REF City_Type,
 Area NUMBER,
 Population NUMBER);
/
CREATE OR REPLACE TYPE City_Type AS OBJECT
 (Name VARCHAR2(35),
  Province REF Province_Type,
  Country REF Country_Type,
  Population NUMBER,
  Coordinates GeoCoord);
/
```

Beim Erzeugen der Tabellen mit Daten muss ebenfalls die Reihenfolge beachtet werden. Sind die zu referenzierenden Objekte beim Erzeugen des referenzierenden Objektes noch nicht bekannt, muss ein *NULL*wert anstelle der Referenz bei der Erzeugung angegeben werden, der später dann durch die Referenz ersetzt wird.

Wenn Objekttypen, die sich gegenseitig zyklisch referenzieren, gelöscht werden, muss man zumindest einmal ein Objekttyp löschen, der noch referenziert wird. Dazu muss

```
DROP TYPE <type> FORCE;
```

verwendet werden (vgl. Seite 5.2).

Unvollständige Datentypen können nur zur Definition von *Referenzen* auf sie benutzt werden, nicht zur Definition von Spalten oder in geschachtelten Tabellen.

Referentielle Integrität. Die Verwendung von OID's garantiert, dass auch bei Änderung der Schlüsselattribute eines Objekts die referentielle Integrität gewahrt bleibt – im Gegensatz zu dem relationalen Konzept mit Fremdschlüsseln, bei dem explizit integritätserhaltende Maßnahmen durch `ON UPDATE CASCADE/RESTRICT` ergriffen werden müssen.

Durch Löschen von Objekten können dennoch *dangling references* entstehen; der Fall `ON DELETE CASCADE` wird also nicht direkt abgedeckt. Dafür lassen sich *entstandene* dangling references durch

```
WHERE <ref-attribute> IS DANGLING
```

überprüfen (vgl. `WHERE ... IS NULL`). Dies kann z. B. in einem `AFTER`-Trigger der Form

```
UPDATE <table>
  SET <attr> = NULL
  WHERE <attr> IS DANGLING;
```

genutzt werden.

15.5 Methoden: Funktionen und Prozeduren

Der *Type Body* enthält die Implementierungen der Methoden in PL/SQL (als zukünftige Erweiterung ist die direkte Einbindung von Java vorgesehen). Dazu ist PL/SQL ebenfalls an einigen Stellen an geschachtelte Tabellen und objektorientierte Features angepasst worden. Allerdings wird Navigation mit Pfadausdrücken in PL/SQL *nicht* unterstützt.

Beispiel 29 (Pfadausdrücke) Für jede Organisation soll die Bevölkerungszahl des Landes, in der ihr Sitz liegt, ausgegeben werden.

In SQL ist das einfach:

```
SELECT org.seated_in.country.population
FROM Organization_ObjTab org;
```

Ein entsprechende Zuweisung in PL/SQL

```
-- the_org enth"alt ein Objekt vom Typ Organization_Type
pop := the_org.seated_in.country.population;
```

ist nicht erlaubt: Beim Navigieren mit Pfadausdrücken werden effektiv Datenbankzugriffe ausgeführt. Diese können in PL/SQL nur in eingebetteten SQL-Ausdrücken vorgenommen werden. Solche Operationen müssen durch entsprechende Datenbankzugriffe ersetzt werden:

```
-- the_org enth"alt ein Objekt vom Typ Organization_Type
SELECT org.seated_in.country.population
INTO pop
FROM Organization_ObjTab org
WHERE org.abbrev = the_org.abbrev;
```

Jede `MEMBER METHOD` besitzt einen *impliziten* Parameter `SELF` (wie in objektorientierten Programmiersprachen üblich), der bei einem Methodenaufruf das jeweilige Host-Objekt referenziert: ruft man z. B. die Methode `is_member` des Objektes "NATO" auf, so ergibt `SELF` die das Objekt "NATO".

Weiterhin können die bereits in Abschnitt 14.6 beschriebenen Methoden, die ursprünglich für *Collections* (d.h. damals nur PL/SQL-Tabellen) definiert sind, aus PL/SQL auch auf geschachtelte Tabellen angewendet werden. Für ein tabellenwertiges Attribut `<attr-name>` gibt z. B. die Methode `<attr-name>.COUNT` die Anzahl der in der geschachtelten Tabelle enthaltenen Tupel an. Damit wird die geschachtelte Tabelle wie ein Array behandelt[2]. Diese Methoden sind allerdings nicht innerhalb in PL/SQL eingebetteter SQL-Statements – z. B. `SELECT <attr>.COUNT` – erlaubt.

Beispiel 30 (Objekttyp *Organization*: Type Body) Der `TYPE BODY` von *Organization_ Type* sieht damit wie folgt aus:

```
CREATE OR REPLACE TYPE BODY Organization_Type IS
MEMBER FUNCTION is_member (the_country IN VARCHAR2)
RETURN VARCHAR2
IS
BEGIN
  IF SELF.Members IS NULL OR SELF.Members.COUNT = 0 THEN RETURN 'no'; END IF;
  FOR i in  1 .. Members.COUNT
  LOOP
    IF the_country = Members(i).country
    THEN RETURN Members(i).type;
    END IF;
  END LOOP;
  RETURN 'no';
END;
MEMBER FUNCTION people RETURN NUMBER IS
p NUMBER;
BEGIN
  SELECT SUM(population) INTO p
  FROM Country ctry
  WHERE ctry.Code IN
    (SELECT Country
     FROM THE (SELECT Members
               FROM Organization_ObjTab org
               WHERE org.Abbrev = SELF.Abbrev));
  RETURN p;
END;
MEMBER FUNCTION number_of_members RETURN NUMBER
IS
BEGIN
  IF SELF.Members IS NULL THEN RETURN NULL; END IF;
  RETURN Members.COUNT;
END;
MEMBER PROCEDURE add_member
       (the_country IN VARCHAR2, the_type IN VARCHAR2) IS
BEGIN
```

[2]nicht elegant, aber praktisch.

```
    IF NOT SELF.is\_member(the\_country) = 'no' THEN RETURN;
       END IF; \\
    IF SELF.Members IS NULL THEN
       UPDATE Organization_ObjTab
       SET Members = Member_List_Type()
       WHERE Abbrev = SELF.Abbrev;
    END IF;
    INSERT INTO
    THE (SELECT Members
                FROM Organization_ObjTab org
                WHERE org.Abbrev = SELF.Abbrev)
    VALUES (the_country,the_type);
  END;
  END;
  /
```

In *is_member* wäre die naheliegendste Idee gewesen, einen Cursor über `(THE) SELF.Members` zu definieren. Dies ist aber zumindest in ORACLE 8.0 nicht möglich (wo würde syntaktisch das `THE` hingehören?). Stattdessen wird eine `FOR`-Schleife mit `Members.COUNT` verwendet. Zu beachten ist weiterhin, dass der Fall "*Members* ist `NULL`" bei *is_member* und *add_member* separat betrachtet werden muss.

In *people* und *add_member* würde man gerne das innere umständliche `FROM THE(SELECT ...)` durch `FROM SELF.Members` oder etwas ähnliches ersetzen. Dies ist (zumindest in ORACLE bis jetzt) nicht möglich: Mit `SELF.Members` befindet man sich in PL/SQL, d.h. man müsste PL/SQL-Konstrukte verwenden um diese Tabelle zu bearbeiten – `SELECT` ist jedoch ein SQL-Konstrukt, das sich an Datenbank-Tabellen richtet. Hier fällt der immer noch bestehende Unterschied zwischen Datenbank und PL/SQL-Umgebung störend auf[3]. □

Die Objekt-`MEMBER FUNCTIONS` können dann in SQL und PL/SQL wie Attribute durch `<object>.<function>(<ar` selektiert werden (im Falle einer parameterlosen Funktion muss `<object>.<function>()` angegeben werden). Bei einem Aufruf aus SQL ist `<object>` durch einen Pfadausdruck gegeben, wobei Aliasing verwendet werden muss um diesen eindeutig zu machen.

Beispiel 31 (Objekttyp *Organization*: Nutzung von `MEMBER FUNCTIONs`) Die folgende Anfrage gibt alle politischen Organisationen sowie die Art der Mitgliedschaft aus, in denen Deutschland Mitglied ist:

```
SELECT Name, org.is_member('D')
  FROM Organization_ObjTab org
  WHERE NOT org.is_member('D') = 'no';
```

Objekt-`MEMBER PROCEDURES` können nur aus PL/SQL mit `<objekt>.<procedure>(<argument-list>)` aufgerufen werden. Dazu werden freie Prozeduren (d.h., nicht als Objektmethoden an einen Typ/Objekt gebunden) in PL/SQL geschrieben, die es ermöglichen, auf die entsprechenden Objekttabellen zuzugreifen.

[3] dieser Unterschied ist aber bei embedded SQL in C noch viel störender.

Beispiel 32 (Objekttyp *Organization*: Verwendung) Die folgende freie Prozedur erlaubt das Einfügen eines Mitglieds in ein politische Organisation. Dabei wird der Aufruf auf die Objektmethode *add_ member* der ausgesuchten Organisation abgebildet. Ist die angegebene Organisation noch nicht vorhanden, so wird ein entsprechendes Objekt erzeugt:

```
CREATE OR REPLACE PROCEDURE make_member
  (the_org IN VARCHAR2, the_country IN VARCHAR2, the_type IN VARCHAR2) IS
  n NUMBER;
  c Organization_Type;
BEGIN
  SELECT COUNT(*) INTO n
    FROM Organization_ObjTab
    WHERE Abbrev = the_org;
  IF n = 0
  THEN
    INSERT INTO Organization_ObjTab
      VALUES(Organization_Type(NULL,the_org,Member_List_Type(),NULL,NULL));
  END IF;
  SELECT VALUE(org) INTO c
    FROM Organization_ObjTab org
    WHERE Abbrev = the_org;
  IF c.is_member(the_country)='no' THEN
     c.add_member(the_country,the_type);
  END IF;
END;
/
```

Mit den bisher in diesem Beispiel angegebenen `INSERT`-Statements wurde die EU in die neue Tabelle eingefügt. Mit den folgenden Befehlen werden die USA als *special member* in die EU eingetragen, außerdem wird eine noch nicht existierende Organisation erzeugt, deren einziges Mitglied die USA sind:

```
EXECUTE make_member('EU','USA','special member');
EXECUTE make_member('XX','USA','member');
```

Damit kann der gesamte Datenbestand aus den relationalen Tabellen *Organization* und *is_ member* in die Objekttabelle *Organization_ ObjTab* übertragen werden:

```
INSERT INTO Organization_ObjTab
 (SELECT Organization_Type(Name,Abbreviation,NULL,Established,NULL)
  FROM Organization);

CREATE OR REPLACE PROCEDURE Insert_All_Members IS
BEGIN
 FOR the_membership IN
     (SELECT * FROM is_member)
 LOOP
     make_member(the_membership.organization,
                 the_membership.country,
```

```
                    the_membership.type);
 END LOOP;
END;
/

EXECUTE Insert_All_Members;

UPDATE Organization_ObjTab org
SET seated_in =
  (SELECT REF(cty)
   FROM City_ObjTab cty, Organization old
   WHERE org.Abbrev = old.Abbreviation
     AND cty.Name = old.City
     AND cty.Province = old.Province
     AND cty.Country = old.Country);
```

Als weiteres kann man eine freie Funktion *is_member_in* definieren, die testet, ob ein Land Mitglied in einer Organisation ist. Diese Prozedur verwendet natürlich die Objektmethode *is_member* der entsprechenden Organisation:

```
CREATE OR REPLACE FUNCTION is_member_in
     (the_org IN VARCHAR2, the_country IN VARCHAR2)
 RETURN is_member.Type%TYPE IS
 c is_member.Type%TYPE;
BEGIN
  SELECT org.is_member(the_country) INTO c
  FROM Organization_ObjTab org
  WHERE Abbrev=the_org;
  RETURN c;
END;
/
```

Beispiel: Die system-eigene Tabelle `DUAL` wird verwendet um das Ergebnis freier Funktionen ausgeben zu lassen.

```
SELECT is_member_in('EU','SL')
FROM DUAL;
```

is_member_in('EU','SL')
applicant

Bemerkung 3 Es ist (zumindest in ORACLE 8.0.x) nicht möglich, durch Navigation mit Pfadausdrücken Tabelleninhalte zu verändern:

```
UPDATE Organization_ObjTab org
SET org.seated_in.Name = 'UNO City'   -- NICHT ERLAUBT
WHERE org.Abbrev = 'UN';
```

15.6 ORDER- und MAP-Methoden

Objekttypen besitzen im Gegensatz zu den Datentypen `NUMBER` und `VARCHAR` keine inhärente Ordnung. Um eine Ordnung auf Objekten eines Typs zu definieren, können dessen funktionale Methoden verwendet werden. In ORACLE kann dazu für jeden Objekttyp eine Funktion als `MAP FUNCTION` oder `ORDER FUNCTION` ausgezeichnet werden.

- Eine `MAP`-Funktion besitzt keine Parameter und bildet jedes Objekt auf eine Zahl ab. Damit wird eine lineare Ordnung auf dem Objekttyp definiert, die sowohl für Vergleiche `<`, `>` und `BETWEEN`, als auch für `ORDER BY` verwendet werden kann.
  Einer `MAP`-Funktion entspricht dem mathematischen Konzept einer *Betragsfunktion*.
- Eine `ORDER`-Funktion besitzt *ein* Argument desselben Datentyps das ein reiner `IN`-Parameter ist (deshalb die Deklaration `<func-name>(<var> <type>)`) und mit dem Hostobjekt verglichen wird.
  Eine `ORDER`-Funktionen entspricht einem *direkten Vergleich* zweier Werte/Objekte. Damit sind `ORDER`-Funktionen für Vergleiche `<`, `>` geeignet, im allgemeinen aber nicht unbedingt für Sortierung.
- `MAP`- und `ORDER`-Funktionen erfordern `PRAGMA RESTRICT_REFERENCES (<name>,WNDS,WNPS,RNPS,RNDS)`, d.h. sie dürfen *keinen Datenbankzugriff* enthalten.

`MAP`-Methoden.

Beispiel 33 (Objekttyp *GeoCoord*: `MAP`-Methode) Der Objekttyp *City_ Type* verwendet den Objekttyp *GeoCoord* zur Speicherung geographischer Koordinaten. Auf *GeoCoord* ist eine `MAP`-Methode definiert, die nun verwendet werden kann, um Städte nach ihrer Entfernung von Greenwich zu ordnen:

```
SELECT Name, cty.coordinates.longitude, cty.coordinates.latitude
FROM City_ObjTab cty
WHERE NOT coordinates IS NULL
ORDER BY coordinates;
```

Andererseits sind viele Operationen in `MAP`-Methoden nicht erlaubt: Wie bereits beschrieben, dürfen diese keine Anfragen an die Datenbank enthalten. Weiterhin scheinen – zumindest in ORACLE 8 – keine built-in Methoden von geschachtelten Tabellen verwendbar zu sein (siehe folgendes Beispiel):

Beispiel 34 (Objekttyp *Organization*: `MAP`-Methode)

- In *Organization_ Type* is es nicht möglich, die Methode *People* als `MAP`-Methode zu verwenden, da diese eine Datenbankanfrage enthält.
- Die Methode *number_ of_ members* kann – zumindest in ORACLE 8.0.x – ebenfalls nicht als `MAP`-Methode verwendet werden: Wird die Deklaration von *Organization_ Type* in `MAP MEMBER FUNCTION number_of_members RETURN NUMBER` geändert, so ergibt

  ```
  SELECT * FROM Organization_ObjTab org ORDER BY VALUE(org);
  ```

  eine Fehlermeldung, dass der Zugriff auf geschachtelte Tabellen hier nicht erlaubt sei. □

ORDER-Methoden. ORDER-Methoden *vergleichen* jeweils SELF mit einem anderen Objekt desselben Typs, das formal als Parameter angegeben wird. Als Ergebnis muss -1 (SELF < Parameter), 0 (Gleichheit) oder 1 (SELF > Parameter) zurückgegeben werden. Damit lassen sich kompliziertere (Halb-)Ordnungen definieren.

Wird bei einer Anfrage ein ORDER BY angegeben, werden die Ausgabeobjekte paarweise verglichen und entsprechend der ORDER-Methode geordnet.[4].

Ein Beispiel hierfür ist etwa die Erstellung der Fußball-Bundesligatabelle: Ein Verein wird vor einem anderen platziert, wenn er mehr Punkte hat. Bei Punktgleichheit entscheidet die Tordifferenz. Ist auch diese dieselbe, so entscheidet die Anzahl der geschossenen Tore (vgl. Aufgabe).

Verwendung. ORDER/MAP-Methoden definieren eine Ordnung auf Objekten eines Typs, und können damit auf zwei Arten verwendet werden:

- Ordnen nach dem Attributwert einer objektwertigen Spalte:

```
CREATE TYPE <type> AS OBJECT
  (  ...
     MAP/ORDER MEMBER METHOD <method> ...);

CREATE TABLE <table>
  (  ...,  <attr> <type>,
     ...);

SELECT * FROM <table>
ORDER BY <attr>;
```

- Die Elemente einer Objekttabelle sollen nach ihrem *eigenen* Objektwert geordnet werden:

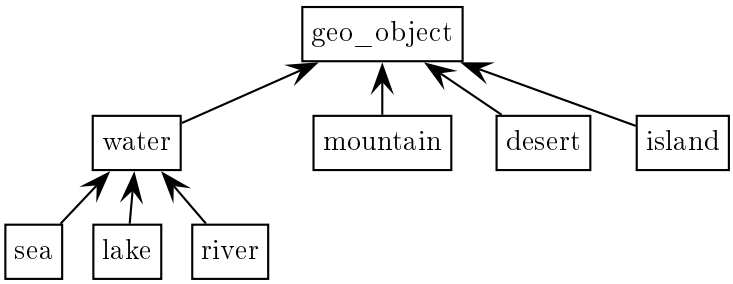
```
CREATE TYPE <type> AS OBJECT
  (  ...
     MAP/ORDER MEMBER METHOD <method> ...);

CREATE TABLE <table> OF <type>;

SELECT * FROM <table> <alias>
ORDER BY value(<alias>);
```

15.7 Klassenhierarchie und Vererbung

Seit ORACLE 9 kann auch eine Typhierarchie (in diesem Fall auch die Klassenhierarchie definierend) erzeugt werden indem *Subtypen* von existierenden Typen abgeleitet werden. Dabei werden als Default die Signatur (Attribute und Methoden) sowie die Implementierung der Methoden von den Obertypen zu den Subtypen vererbt. Weiterhin können Subtypen neue Attribute und Methoden hinzudefinieren, und geerbte Methoden durch eigenen Implementierungen überschreiben (*overriding*).

Beispiel 35 (Klassenhierarchie geographischer Objekte)

[4]Da muss man mal eine nicht-zyklenfreie Relation ausprobieren!



Als weiteres ist zu beachten, dass – wie z. B. auch in Java – weitere Informationen bei der Klassendefinition angegeben werden können:

- Wenn eine Methode in einem Subtyp überschrieben wird, muss dies in der Deklaration des Subtyps explizit durch das Schlüsselwort `OVERRIDING` angegeben werden.
- *Abstrakte* Klassen dienen im wesentlichen der Hierarchiebildung. Von ihnen werden keine Instanzen gebildet. In ORACLE wird dies durch `NOT INSTANTIABLE` angegeben. Defaultwert ist `INSTANTIABLE`.

  Werden einzelne Methoden bei einer abstrakten Klasse nicht implementiert, muss dies bei der Methodendeklaration angegeben werden (sonst bekommt man eine Fehlermeldung bei `CREATE TYPE BODY`, wenn für diese Methode die Implementierung fehlt).
- Wenn eine solche abstrakte Methode dann doch irgendwo implementiert wird, muss sie (i) in der Typdefinition angegeben werden, und (ii) als `OVERRIDING` spezifiziert werden.
- Von *finalen* Klassen können keine weiteren Subklassen abgeleitet werden. Weiterhin können auch einzelne Methoden als *final* deklariert werden, was bedeutet, dass zwar Subklassen gebildet werden können, aber diese Methode nicht überschrieben werden darf.

  Hinweis: Defaultwert für Klassen ist `FINAL`, für Methoden `NOT FINAL`. Daher muss man, wenn man von einem Typ weitere Subtypen ableiten will, `NOT FINAL` explizit angeben.

Beispiel 36 (Typhierarchie geographischer Objekte)

Die im vorigen Beispiel angegebene Klassenhierarchie wird in SQL wie folgt erzeugt:

```
CREATE OR REPLACE TYPE geo_object_type AS OBJECT (
     name VARCHAR2(25),
     MEMBER FUNCTION get_name RETURN VARCHAR2,
     NOT INSTANTIABLE MEMBER FUNCTION set_name RETURN VARCHAR2
   )
   NOT INSTANTIABLE
   NOT FINAL;
/

CREATE OR REPLACE TYPE BODY geo_object_type IS
  MEMBER FUNCTION get_name RETURN VARCHAR2
  IS BEGIN RETURN name; END;
  -- keine Implementierung fuer set_name angegeben.
END;
/
```

Der Objekttyp `geo_object` implementiert nur `get_name`, während `set_name` erst in den einzelnen Klassen implementiert wird. Als erstes wird eine ebenfalls abstrakte *Gewässer*-Klasse eingeführt. In der aktuellen Version können Subklassen nur definiert werden, wenn sie irgendetwas verfeinern, deswegen wird hier eine `bla`-Methode angegeben.

```
CREATE OR REPLACE TYPE water_type UNDER geo_object_type (
   MEMBER FUNCTION bla RETURN NUMBER
   -- empty derivation not allowed in current version
   )
   NOT FINAL
   NOT INSTANTIABLE;
/
```

Hierzu müsste jetzt ein Type Body für `water_type` angegeben werden, der `bla` implementiert.

Die finalen Objekttypen für Meere, Seen und Flüsse müssen nun als `INSTANTIABLE` deklariert werden. Dabei müssen auch noch Methoden, die bisher als `NOT INSTANTIABLE` deklariert sind, angegeben werden. Dies erfordert die Deklaration als `OVERRIDING`:

```
CREATE OR REPLACE TYPE sea_type UNDER water_type (
   depth NUMBER,
   OVERRIDING MEMBER FUNCTION set_name RETURN VARCHAR2
   )
   INSTANTIABLE;
/
```

Hierzu müsste jetzt ein Type Body für `sea_type` angegeben werden, der `set_name` implementiert. Mit der Deklaration

```
   OVERRIDING MEMBER FUNCTION bla RETURNS NUMBER;
```

könnte man auch die für Gewässer bereits implementierte Methode `bla` überschreiben und neu implementieren.

Die weiteren finalen Klassen werden wie folgt erzeugt:

```
CREATE OR REPLACE TYPE lake_type UNDER water_type (
   area NUMBER,
   OVERRIDING MEMBER FUNCTION set_name RETURN VARCHAR2)
   INSTANTIABLE;
/
CREATE OR REPLACE TYPE river_type;
/
CREATE OR REPLACE TYPE river_type UNDER water_type (
   river REF river_type,
   lake REF lake_type,
   sea REF sea_type,
   length NUMBER,
   OVERRIDING MEMBER FUNCTION set_name RETURN VARCHAR2)
```

```
   INSTANTIABLE;
/
CREATE OR REPLACE TYPE mountain_type UNDER geo_object_type (
   height NUMBER,
   coordinates GeoCoord,
   OVERRIDING MEMBER FUNCTION set_name RETURN VARCHAR2)
   INSTANTIABLE;
/
CREATE OR REPLACE TYPE desert_type UNDER geo_object_type (
   area NUMBER,
   OVERRIDING MEMBER FUNCTION set_name RETURN VARCHAR2)
   INSTANTIABLE;
/
CREATE OR REPLACE TYPE island_type UNDER geo_object_type (
   islands VARCHAR2(25),
   area NUMBER,
   coordinates GeoCoord,
   OVERRIDING MEMBER FUNCTION set_name RETURN VARCHAR2)
   INSTANTIABLE;
/
```

Damit können jetzt alle geographischen Objekte (Berge, Wüsten, Gewässer) in einer einzigen Tabelle gespeichert werden:

```
CREATE TABLE geo_obj OF geo_object_type;
INSERT INTO geo_obj
  SELECT sea_type(name, depth) FROM sea;
INSERT INTO geo_obj
  SELECT lake_type(name, area) FROM lake;
INSERT INTO geo_obj
  SELECT river_type(name, NULL, NULL, NULL, length) FROM river;
INSERT INTO geo_obj
  SELECT mountain_type(name, height, coordinates) FROM mountain;
INSERT INTO geo_obj
  SELECT desert_type(name, area) FROM desert;
INSERT INTO geo_obj
  SELECT island_type(name, islands, area, coordinates) FROM island;
```

Die Tabelle `geo_obj` ist nun eine Kollektion von Objekten der Klasse `geo_obj_type`. Nach dem *Liskov'schen Substituierbarkeitsprinzip* kann ein Element einer Subklasse überall auftreten, wo ein Elemente einer seiner Superklassen erwartet wird (also z. B. eine Instanz von *River* überall dort, wo ein *Gewässer* erwartet wird). In objektorientierten Datenbanken betrifft dies die folgenden Situationen:

- als Zeilenobjekte in Objekttabellen,
- als Spaltenobjekte (objektwertige Attribute)
- als Referenzattribute,

- Als Argumente und Rückgabewerte bei Funktionen und Prozeduren.

In allen diesen Situationen kann man die betreffenden Objekte problemlos nach Eigenschaften anfragen, die alle Objekte der dem Compiler/Interpreter an dieser Stelle bekannten Objekte haben. In dem obigen Beispiel kann man z. B. die Namen aller Objekte der Objekttabelle `geo_obj` ausgeben lassen:

```
SELECT name FROM geo_obj;
```

Will man eine Eigenschaft abfragen, die nicht für alle in der Kollektion enthaltenen Instanzen deklariert sind, muss man Typtests und *Typecasts* (wie in C++/Java) verwenden:

Typanfrage: Die Funktion `SYS_TYPEID(<object>` ergibt die ID der speziellsten Klasse, der ein Objekt angehört. Für die dabei enthaltene Zahl kann dann in `ALL_TYPES` nachgesehen werden, welcher Typ es ist.
Die folgende Anfrage ergibt, dass "Llullaillaco" ein Berg ist:

```
SELECT type_name, typeid, supertype_name
FROM all_types
WHERE typeid = (SELECT SYS_TYPEID(value(x))
                FROM geo_obj x
                WHERE name='Llullaillaco');
```

Typtest: Mit der Funktion `IS OF <type>` kann angefragt werden, ob ein Objekt von einem bestimmten Typ (meistens ein Subtyp des Typs über dem die Objekttabelle/Kollektion definiert ist) ist.
Die Anfrage

```
SELECT x.name
FROM geo_obj x
WHERE value(x) IS OF (mountain_type);
```

gibt alle Namen von Bergen aus. Will man nun deren Höhen wissen, kann man nicht einfach `SELECT x.name, x.height` schreiben, da für den Interpreter nur bekannt ist, dass die Objekte in `geo_obj` vom Typ `geo_obj_type` sind – dieser besitzt aber kein Attribut `height`. Dazu muss man erst diese Objekte explizit als Berge bekanntmachen.

Type-Cast: Mit der Funktion `TREAT (<object> AS <type>)` kann man ein Objekt explizit als ein Objekt eines bestimmten Subtyps auffassen – was natürlich nur möglich ist, wenn es eine Instanz dieses Subtyps ist. Ansonsten liefert die Funktion `NULL` zurück. Im obigen Beispiel bekommt man alle Namen von Bergen mit

```
SELECT x.name, (TREAT (value(x) AS mountain_type)).height
FROM geo_obj x
WHERE value(x) IS OF (mountain_type);
```

15.8 Änderungen an Objekttypen

Existierende und bereits verwendete Objekttypen können seit Version ORACLE 9 auch durch `ALTER TYPE` verändert werden:

- Setzen und Löschen der `FINAL` und `INSTANTIABLE`-Schalter:
  `ALTER TYPE <name> [NOT] INSTANTIABLE`, und
  `ALTER TYPE <name> [NOT] FINAL`.
- Hinzunehmen, Löschen, und Verändern von Attributdeklarationen:

  ```
  ALTER TYPE <name>
    ADD    ATTRIBUTE (<name> <datatype>),
    DROP   ATTRIBUTE <name>;
  <options>
  ```

  Hinzugefügte Attribute werden mit Nullwerten aufgefüllt.
- Hinzunehmen, Löschen, und Verändern von Methodendeklarationen:

  ```
  ALTER TYPE <name>
    ADD  MEMBER {PROCEDURE|FUNCTION} (<method-spec>),
    DROP MEMBER {PROCEDURE|FUNCTION} <method-spec>;
         <options>
  ```

  Nach dem Hinzufügen oder Modifizieren einer Methodendeklaration muss der *gesamte* Type Body neu mit `CREATE TYPE BODY` erzeugt werden. Ein `ALTER TYPE BODY` wird derzeit nicht angeboten.
- bei `VARCHAR`-Attributen kann die Länge erhöht werden (über `MODIFY ATTRIBUTE`).
- Veränderungen müssen im Benutzungsgraphen zwischen Objekttypen und anderen Datenbankobjekten propagiert werden. Innerhalb der Typhierarchie, Tabellen etc. geschieht dies durch die Angabe von Optionen:
  - `CASCADE` kaskadiert die Aktualisierung zu abhängigen Definitionen (siehe auch Manuals für Suboptionen),
  - `INVALIDATE` kennzeichnet abhängige Datenbankobjekte, die bei der nächsten Benutzung neu compiliert werden.

Beispiel:

```
ALTER TYPE mountain_type FINAL CASCADE;
```

Weitere Beispiele ließen sich mit der vorliegenden Version nicht ausprobieren (System hängt sich zu oft auf).

Eine benutzte Definition kann nicht mit `CREATE OR REPLACE` verändert oder einfach mit `DROP` gelöscht werden. Daher empfiehlt sich, mit dem Zusatzargument `FORCE` auch Typen, für die Abhängigkeiten existieren, zu löschen, z. B.

```
DROP TYPE geo_object_type FORCE;
DROP TYPE water_type FORCE;
DROP TYPE sea_type FORCE;
DROP TYPE lake_type FORCE;
```

```
DROP TYPE river_type FORCE;
DROP TYPE mountain_type FORCE;
DROP TYPE desert_type FORCE;
DROP TABLE geo_obj;
```

15.9 Objekttypen: Indexe und Zugriffsrechte

Indexe auf Objektattributen. Indexe (vgl. Abschnitt 13) können auch auf Objektattributen erstellt werden:

```
CREATE INDEX <name>
  ON <object-table-name>.<attr>[.<attr>]*;
```

Hierbei trägt die eckige Klammer der Tatsache Rechnung, dass evtl. ein Index über ein Teilattribut eines komplexen Attributes erstellt werden soll. Indexe können *nicht* über komplexen Attributen (also Attributen eines selbstdefinierten Typs) erstellt werden: Es ist z. B. nicht möglich, in Beispiel 6 einen Index über das Attribut *coordinates*, bestehend aus *longitude* und *latitude* zu erstellen:[5]

```
CREATE INDEX city_index ON City_ObjTab(coordinates);           -- nicht erlaubt
CREATE INDEX city_index ON City_ObjTab(coordinates.Longitude,
                                       coordinates.Latitude);  -- erlaubt
```

Indexe über Referenzattributen sind (zumindest in ORACLE 8.0.3) nur erlaubt, wenn für das Referenzattribut ein `SCOPE IS`-Constraint angegeben ist.

Außerdem können auch Indexe auf Funktionswerte erzeugt werden, die dann auf vorberechneten Werten basieren:

```
CREATE INDEX name ON
    Organization_Obj_Tab (number_of_members);
```

Zugriffsrechte auf Objekte. Rechte an Objekttypen werden durch `GRANT EXECUTE ON <Object-datatype> TO ...` vergeben – die zugrundeliegende Idee ist hier, dass bei der Benutzung eines Datentyps vor allem die Methoden (u. a. die entsprechende Konstruktormethode) im Vordergrund stehen.

[5] Fehlermeldung in solchen Fällen: *Index für Spalte mit Datentyp ADT kann nicht erstellt werden*; ADT bedeutet *Abstrakter Datentyp*.

16 OBJECT-VIEWS IN ORACLE

Object Views dienen genauso wie relationale Views dazu, eine andere Darstellung der in der Datenbank enthaltenen Daten zu ermöglichen. Durch die Definition von Objektdatentypen, die evtl. nur in den Views verwendet werden, ist es möglich, den Benutzern maßgeschneiderte Object-Views mit sehr weitgehender Funktionalität anzubieten.

Object Views haben verschiedene Anwendungen:

Legacy-Datenbanken Häufig müssen bestehende Datenbanken in ein "modernes" objektorientiertes Modell eingebunden werden. In diesem Fall kann man über die relationale Ebene eine Ebene von *Objekt-Views* legen, die nach außen hin ein objektorientiertes Datenbanksystem erscheinen lassen. In diesem Fall wird auch von *Objekt-Abstraktionen* gesprochen.

Effizienz + Benutzerfreundlichkeit: Selbst bei einem Neuentwurf zeigt sich an vielen Stellen, dass eine relationale Repräsentation am effizientesten ist:

- Geschachtelte Tabellen werden intern als separate Tabellen gespeichert.
- $n : m$-Beziehungen könnten "objektorientiert" durch gegenseitige geschachtelte Tabellen modelliert werden, was aber sehr ineffizient ist.

Auch hier ist es sinnvoll, ein – zumindest teilweise – relationales Basisschema zu definieren, und darauf Object-Views zu erstellen.

In der ORACLE-Community wird daher häufig empfohlen, Object Views mit geschachtelten Tabellen, Referenzen etc. *auf Basis eines relationalen Grundschemas* zu verwenden. In diesem Fall können das Grundschema und das externe Schema unabhängig voneinander (durch jeweilige Anpassung der Methodenimplementierung) geändert werden.

In jedem Fall führt der Benutzer seine Änderungen auf dem durch die Objektviews gegebenen externen Schema durch. Direkte Änderungen durch `INSERT`, `UPDATE` und `DELETE`, werden durch `INSTEAD OF`-Trigger (vgl. Abschnitt 14.7.2) auf das darunterliegende Schema umgesetzt. Viel besser ist es jedoch, den Benutzer erst gar keine solchen Statements an das View stellen zu lassen, sondern die entsprechende Funktionalität durch Methoden der Objekttypen zur Verfügung zu stellen, die die Änderungen direkt auf den zugrundeliegenden Basistabellen ausführen.

16.1 Anlegen von Objektviews

Objektrelationale Views. Relativ einfache objektrelationale Views auf Basis von relationalen oder objektorientierten Basistabellen lassen sich auch ohne Objekttypen definieren (wobei man natürlich dann keine Methoden hat). Die Syntax ist prinzipiell dieselbe wie für relationale Views:

```
CREATE [OR REPLACE] VIEW <name> (<column-list>) AS
  <select-clause>;
```

Dabei sind in der SELECT-clause jetzt zusätzlich Konstruktormethoden für Objekte und geschachtelte Tabellen erlaubt. Für die Erzeugung geschachtelter Tabellen für Object Views werden die Befehle CAST und MULTISET (vgl. Abschnitt 5.2/Beispiel 7) verwendet.

Beispiel 37 (Objektrelationales View über Relationalem Schema) Um für jedem Fluss auch direkt auf seinen Nebenflüsse zugreifen zu können, wird ein entsprechendes View definiert:

```
CREATE TYPE River_List_Entry AS OBJECT
 (name VARCHAR2(20),
  length NUMBER);
/
CREATE TYPE River_List AS
 TABLE OF River_List_Entry;
/

CREATE OR REPLACE VIEW River_V
  (Name, Length, Tributary_Rivers)
AS SELECT
   Name, Length,
   CAST(MULTISET(SELECT Name,Length FROM River
                 WHERE River = A.Name) AS River_List)
   FROM River A;

SELECT * FROM River_V;
```

Objektviews. Objektviews enthalten wie Objekttabellen Zeilenobjekte, d.h. hier werden neue Objekte *definiert*. Bei der Definition von Objektviews wird durch WITH OBJECT OID <attr-list> angegeben, aus welchen Attributen die Objekt-ID berechnet wird. Auch hier werden CAST und MULTISET verwendet um Views mit geschachtelten Tabellen zu definieren.

```
CREATE [OR REPLACE] VIEW <name> OF <type>
  WITH OBJECT OID (<attr-list>)
  AS <select-statement>;
```

Hierbei ist zu beachten, dass in <select-statement> *kein* Objektkonstruktor verwendet wird:

Beispiel 38 (Object Views: *Country*)

```
CREATE OR REPLACE TYPE Country_Type AS OBJECT
 (Name       VARCHAR2(32),
  Code       VARCHAR2(4),
  Capital    REF City_Type,
  Area       NUMBER,
  Population NUMBER);
/

CREATE OR REPLACE VIEW Country_ObjV OF Country_Type
WITH OBJECT OID (Code)
```

```
AS
SELECT Country.Name, Country.Code, REF(cty), Area, Country.Population
FROM Country, City_ObjTab cty
WHERE cty.Name = Country.Capital
  AND cty.Province = Country.Province
  AND cty.Country = Country.Code;

SELECT Name, Code, c.capital.name, Area, Population
FROM Country_ObjV c;
```

Beispiel 39 (Object Views: Was nicht geht) Leider scheint das Konzept nicht absolut ausgereift zu sein: Will man ein Object View auf Basis von *Organization_ ObjTab* definieren, stellt man fest, dass dieses offensichtlich weder die geschachtelte Tabelle noch das Ergebnis einer funktionalen Methode der zugrundeliegenden Tabellen enthalten darf:

```
CREATE OR REPLACE TYPE Organization_Ext_Type AS OBJECT
 (Name VARCHAR2(80),
  Abbrev VARCHAR2(12),
  Members Member_List_Type,  -- nicht erlaubt
  established DATE,
  seated_in REF City_Type,
  number_of_people NUMBER);  -- nicht erlaubt
/
CREATE OR REPLACE VIEW Organization_ObjV OF Organization_Ext_Type
AS
SELECT Name, Abbrev, Members, established, seated_in, org.people()
 FROM Organization_ObjTab org;

FEHLER in Zeile 3:
ORA-00932: nicht uebereinstimmende Datentypen
```

Beide angegebenen Attribute sind auch einzeln nicht erlaubt. □

16.2 Fazit

- Kompatibilität mit den grundlegenden Konzepten von ORACLE 7. U.a. können Fremdschlüsselbedingungen von Objekttabellen zu relationalen Tabellen definiert werden.
- Objekttypen bieten seit der in Version ORACLE 9 hinzugekommenen Möglichkeit, eine Typhierarchie zu definieren sowie Typen zu verändern, die Möglichkeit objektorientiert zu modellieren, ohne auf die Effizienz eines hochoptimierten Datenbanksystems verzichten zu müssen.
- Durch *Object Views* kann basierend auf einem existierenden relationalen Grundschema ein Zweitschema definiert werden, mit dem sich eine Datenbank den Benutzern präsentiert. Durch entsprechende Methoden und `INSTEAD OF`-Trigger können Benutzer-Interaktionen auf das Grundschema umgesetzt werden.