

# Kapitel 9

## Objekt-Relationale Datenbanksysteme

Integration von relationalen Konzepten und Objektorientierung:

- Komplexe Datentypen: Erweiterung des Domain-Konzepts von SQL-2 (vgl. DATE, Geo-Koordinaten)
- Abstrakte Datentypen ("Objekttypen"):
  - Unterscheidung zwischen dem *Zustand* und *Verhalten* eines *Objektes* (Kapselung interner Funktionalität).
  - Im Gegensatz zu einem *Tupel* besitzt ein Objekt
    - \* Attribute (beschreiben seinen Zustand),
    - \* Methoden – Abfragen und Ändern des Zustandes: *Prozeduren* und *Funktionen* (Oracle 8: PL/SQL, Oracle 8i: auch Java, siehe Folie 324)
    - \* MAP/ORDER-Funktionen: Ordnung auf Objekttyp
- Spezielle Ausprägungen:
  - Geschachtelte Tabellen als Attributwerte,
  - Erweiternde Datentypen (Spatial etc.),
  - Built-In XMLType zur Verarbeitung von XML-Daten (siehe Folie 384).

## STUFEN DER OBJEKTORIENTIERUNG

"Konservative" objektrelationale Erweiterungen (seit Oracle 8)

(siehe Folie 242)

- Objekte als "Werte" von Attributen:  
Spalten einer Tupeltabelle können *objektwertig* sein.
- Objekte anstelle von Tupeln:  
Tabellen von Tupeln vs. *Object Tables* aus Objekten
- Typ definiert gemeinsame Signatur seiner Instanzen (Objekte)
- bereits behandelt: Komplexe Attributtypen. Besitzen nur *Wertattribute*, keine Methoden.

Objektorientierte Datenbanken

(siehe Folie 268)

- Beziehungen nicht mehr über Schlüssel/Fremdschlüssel sondern über Referenzen  
⇒ Navigation anstatt Joins
- seit ORACLE 9i: Subtypen und Vererbung, Objekttypen aus Java-Klassen.

## 9.1 Objektrelationale Konzepte

Alles funktioniert (fast) genauso wie bisher:

- Spalten einer Tupeltabelle können *objektwertig* sein (vgl. Geo-Koordinaten)

- Tabellen von Tupeln vs. *Object Tables* aus Objekten

```
INSERT INTO <table>
VALUES(<object-constructor>(attr1, ..., attrn))
```

anstatt

```
INSERT INTO <table>
VALUES(attr1, ..., attrn)
```

- Zugriff auf Attribute wie bisher mit `tablename.attr`,
- zusätzlich Aufruf von Methoden mit `tablename.meth(...)`.

### 9.1.1 Definition von Objekttypen

#### Typdeklaration

- Attribute,
- *Signatures* der Methoden,

#### Typ-Implementierung

- **Type Body:** Implementierung der Methoden in PL/SQL
- seit Oracle 8i auch in PL/SQL+Java (siehe Folien 324 und 330)

## OBJEKTYPDEKLARATION

```
CREATE [OR REPLACE] TYPE <type> AS OBJECT
  (<attr> <datatype>,
   <attr> <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>]
);
/ ← dieser Slash ist unbedingt notwendig!
```

- <parameter-list> wie in PL/SQL,
- ähnlich CREATE TABLE, aber *keine* Integritätsbedingungen (erst bei der (Objekt)tabellen-Definition)

## BEISPIEL: GEO-KOORDINATEN

- Methode *Distance*(geo-coord-Wert)
- MAP-Methode: Entfernung von Greenwich.

```
CREATE OR REPLACE TYPE GeoCoord AS OBJECT
  (Latitude NUMBER,
   Longitude NUMBER,
   MEMBER FUNCTION Distance (other IN GeoCoord)
     RETURN NUMBER,
   MAP MEMBER FUNCTION Distance_Greenwich
     RETURN NUMBER
  );
/ ← dieser Slash ist unbedingt notwendig!
```

[Filename: ObjRel/geocoord-type.sql]

- wenn der Objekttyp bereits existiert (wie in diesem Fall):

```
ALTER TYPE GeoCoord
  ADD MEMBER FUNCTION Distance (other IN GeoCoord)
    RETURN NUMBER,
  ADD MAP MEMBER FUNCTION Distance_Greenwich
    RETURN NUMBER
  CASCADE INCLUDING TABLE DATA;
```

[Filename: ObjRel/geocoord-type-ext.sql]

- CASCADE: "forwards" definition to tables and data objects where it is used.

## TYPE BODY

- Implementierung der Objektmethoden,
- muss der der bei CREATE TYPE vorgegeben Signatur entsprechen,
- für *alle* deklarierten Methoden muss Implementierung angegeben werden.
- Variable SELF, um auf die Attribute des Host-Objektes zuzugreifen.

**Funktionen:** dürfen den Datenbankzustand nicht verändern,

**MAP/ORDER-Funktionen:** kein Datenbankzugriff erlaubt

⇒ verwenden nur den Zustand der beteiligten Objekte.

## TYPE BODY

```

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: GEO-KOORDINATEN**

```

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(51.5, 0));
  END;
END;
/

```

[Filename: ObjRel/geocoord-body.sql]

**ERZEUGUNG VON OBJEKTEN**

- Konstruktormethode:

<type>(<arg\_1>, ..., <arg\_n>)

Also kein NEW, sondern nur einfach

GeoCoord(48,8)

CityORType('Berlin', 'Berlin', 'D',

3472009, GeoCoord(52.45, 13.3))

**METHODENAUFBRUF**

- Funktionen: in Anfragen oder in PL/SQL-Programmen
- Prozeduren: in PL/SQL-Programmen
- Syntax:

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

**Beispiel**

Wie gross ist der Abstand zwischen zwei Längengraden auf der Höhe von Berlin, bzw. am Äquator?

```
SELECT geoCoord(52.45,-30).Distance(geoCoord(52.45,-31))
```

```
FROM DUAL;
```

```
SELECT geoCoord(0,-30).Distance(geoCoord(0,-31))
```

```
FROM DUAL;
```

## 9.1.2 Zugriffscharakteristik (Veraltet – bis einschl. Version 8i notwendig)

- Explizite Angabe der Read/Write-Zugriffscharakteristik der Methoden in PRAGMA RESTRICT REFERENCES-Klauseln
- seit 9i: wird zur Compile-Time intern gemacht (überprüft wurde es sowieso schon ...)

```
CREATE [OR REPLACE] TYPE <type> AS OBJECT
(<attr> <datatype>,
 <attr> <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> ]
);
/
```

## PRAGMA-KLAUSELN:

### Read/Write-Zugriffscharakteristik

<pragma-declaration-list>:

für jede Methode eine PRAGMA-Klausel

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

<feature-list>:

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

**Funktionen:** es muss *zugesichert* werden, dass sie den Datenbankzustand nicht verändern:

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

**MAP/ORDER-Funktionen:** kein Datenbankzugriff erlaubt

```
PRAGMA RESTRICT_REFERENCES
 (<function_name>, WNDS, WNPS, RNPS, RNDS)
```

⇒ verwendet nur den Zustand der beteiligten Objekte.

## BEISPIEL: GEO-KOORDINATEN MIT PRAGMA-KLAUSEL

```
CREATE OR REPLACE TYPE GeoCoord AS OBJECT
  (Latitude NUMBER,
   Longitude 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)
  );
/
```

## 9.1.3 Verwendung von Objekttypen

- Als Werte von Attributen: "Spaltenobjekte"  
(vgl. Geo-Koordinaten)
- in *Objekttabellen*: TABLE OF <objecttype>  
"Zeilenobjekte"  
vollwertige Objekte

## SPALTENOBJEKTE

- Attribut eines Tupels oder eines Objekts ist objektwertig:

```
CREATE TABLE Mountain
(Name VARCHAR2(20)
 CONSTRAINT MountainKey PRIMARY KEY,
 Elevation NUMBER,
 Coordinates GeoCoord CONSTRAINT MountainCoord
 CHECK ((Coordinates.Latitude >= -90) AND
 (Coordinates.Latitude <= 90) AND
 (Coordinates.Longitude > -180) AND
 (Coordinates.Longitude <= 180)));
```

[Filename: ObjRel/mountain-table.sql]

- Constraints werden wie immer bei der Tabellendefinition angegeben.

```
INSERT INTO Mountain
VALUES ('Feldberg', 1493, GeoCoord(48.5, 7.5));
SELECT Name, mt.coordinates.distance(geocoord(90, 0))
FROM Mountain mt;
```

- Tupelvariable *mt* um den Zugriffspfad zu *coordinates.distance* eindeutig zu machen.

## ZEILENOBJEKTE

- Elemente von *Objekttabellen*,
- ihre Attribute verhalten sich genauso wie die Attribute von Tupeltabellen,
- zusätzlich kann man Methoden aufrufen,
- referentielle Integritätsbedingungen zwischen bestehenden relationalen Tabellen und solchen Objekttabellen wie üblich formulierbar,
- (erhalten eine eindeutige OID und sind damit referenzierbar)

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

mit <constraint-list> wie bisher:

- attributbezogene Bedingungen entsprechen den Spaltenbedingungen:

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

- Tabellenbedingungen: Syntax wie bei Tupeltabellen.



## ZEILENOBJEKTE

**Beispiel: CityORType**

Objekt-Relationaler City-Typ:

- Spalten des Typs sind literal- oder objektwertig,
- noch keine Objektreferenzen

```
CREATE OR REPLACE TYPE CityORType AS OBJECT
(Name VARCHAR2(50),
 Province VARCHAR2(50),
 Country VARCHAR2(4),
 Population NUMBER,
 Coordinates GeoCoord,
 MEMBER FUNCTION Distance (other IN CityORType)
 RETURN NUMBER,
 MEMBER FUNCTION NoOfOrganizations
 RETURN NUMBER);
/
```

[Filename: ObjRel/cityORtype.sql]

## ZEILENOBJEKTE

```
CREATE OR REPLACE TYPE BODY CityORType
AS
 MEMBER FUNCTION Distance (other IN CityORType)
 RETURN NUMBER
 IS
 BEGIN
 RETURN SELF.coordinates.distance(other.coordinates);
 END;
 MEMBER FUNCTION NoOfOrganizations RETURN NUMBER
 IS
 n NUMBER;
 BEGIN
 SELECT count(*) INTO n
 FROM Organization o
 WHERE o.city = SELF.name
 AND o.province = SELF.province
 AND o.country = SELF.country;
 RETURN n;
 END;
END;
/
```

[Filename: ObjRel/cityORtypebody.sql]

## OBJEKTTABELLEN: ZEILENOBJEKTE

- der (ggf. mehrspaltige) Primärschlüssel wird als Tabellenbedingung angegeben,
- Die Fremdschlüsselbedingung auf die relationale Tabelle *Country* wird ebenfalls als Tabellenbedingung angegeben:

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

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

```
INSERT INTO ORCity
SELECT CityORType
(Name, Province, Country, Population,
GeoCoord(Latitude, Longitude))
FROM City;
```

[Filename (beides zusammen): ObjRel/cityORtable.sql]

## VERWENDUNG VON OBJEKTTABELLEN

### Auslesen und Ändern von Attributwerten wie bekannt

- Auslesen:

```
SELECT Name FROM ORCity;
SELECT * FROM ORCity;
```

- Ändern:

```
UPDATE ORCity cty
SET coordinates = NULL
WHERE cty.coordinates.latitude IS NULL;
```

### Methodenaufrufe wie erwartet

```
SELECT Name, c.NoOfOrganizations() FROM ORCity c
WHERE c.NoOfOrganizations() > 0;
```

## VERWENDUNG VON OBJEKTTABELLEN

... Auslesen von Objekten als Objekte:

Das folgende geht so *nicht*:

```
SELECT cty1.Distance(cty2)
FROM ORCity cty1, ORCity cty2
WHERE cty1.Name='Berlin' AND cty2.Name='Stuttgart';
```

- cty2 ist eine SQL-Variable, die nicht direkt das Objekt enthält.

### Die VALUE()-Funktion

VALUE (<var>)

selektiert ein Objekt als Objekt (ist nur auf Zeilenobjekte anwendbar!):

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

<b>VALUE</b> (Cty)(Name, Province, Country, Population, Coordinates(Latitude, Longitude))
CityORType('Berlin', 'Berlin', 'D', 3472009, GeoCoord(52, 13))
CityORType('Bonn', 'Nordrh.-Westf.', 'D', 293072, GeoCoord(50, 8))
CityORType('Stuttgart', 'Baden-Württt.', 'D', 588482, GeoCoord(49, 9))
⋮

## VERWENDUNG VON OBJEKTEN: VALUE

- (Zeilen)Objekte auf Gleichheit testen: mit value(...)
- Objekt als Argument einer Methode

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

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

- Zuweisung eines Objektes mit einem SELECT INTO-Statement an eine PL/SQL-Variable

```
set serveroutput on;
DECLARE c cityORtype;
BEGIN
    SELECT VALUE(cty) INTO c
    FROM ORcity cty
    WHERE name='Berlin';
dbms_output.put_line(c.name);
END;
/
```

### 9.1.4 ORDER- und MAP-Methoden

- Objekttypen besitzen im Gegensatz zu den Datentypen NUMBER und VARCHAR keine inhärente Ordnung.
- Ordnung auf Objekten eines Typs kann über dessen funktionale Methoden definiert werden.
- Für jeden Objekttyp eine MAP FUNCTION oder eine ORDER FUNCTION.

**MAP-Funktion:** (Betragsfunktion)

- keine Parameter,
- bildet jedes Objekt auf eine Zahl ab.
- Lineare Ordnung auf dem Objekttyp, "Betragsfunktion"
- sowohl für Vergleiche <, > und BETWEEN, als auch für ORDER BY verwendbar.

**ORDER-Funktion:** (vgl. Methode compareTo(*other*) des "Comparable" Interfaces in Java)

- besitzt *ein* Argument desselben Objekttyps das mit dem Hostobjekt verglichen wird.
- Damit sind ORDER-Funktionen für Vergleiche <, > geeignet, im allgemeinen aber nicht unbedingt für Sortierung.
- MAP- und ORDER-Funktionen dürfen *keinen* Datenbankzugriff enthalten.

### MAP-METHODEN: BEISPIEL

MAP-Methode auf *GeoCoord*:

```
CREATE OR REPLACE TYPE BODY GeoCoord
AS
:
MAP MEMBER FUNCTION Distance_Greenwich

RETURN NUMBER
IS
BEGIN
RETURN SELF.Distance(GeoCoord(51.5, 0));
END;
END;
/
```

```
SELECT Name, cty.coordinates.latitude,
        cty.coordinates.longitude,
        cty.coordinates.Distance_Greenwich()
FROM ORCity cty
WHERE NOT coordinates IS NULL;
ORDER BY coordinates;
```

[Filename: ObjRel/orderby.sql]

## ORDER-METHODEN

- Vergleich von SELF mit einem anderen Objekt desselben Typs, das formal als Parameter angegeben wird.
- Ergebnis: NUMBER
  - $x < 0$  falls  $SELF < \text{Parameter}$ ,
  - 0 (Gleichheit), oder
  - $x > 0$  falls  $SELF > \text{Parameter}$ .

## • Mit

```
SELECT ...
FROM <tablename> x
ORDER BY VALUE(x)
```

werden die Ausgabeobjekte paarweise verglichen und entsprechend der ORDER-Methode geordnet.

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

## VERGLEICHE ZWISCHEN OBJEKTEN

- $<$ ,  $>$ , “between” und Ordnung basieren auf der MAP bzw. ORDER-Methode.
- Gleichheit/Ungleichheit:
  - bei Zeilenobjekten: diese haben eine Objektidentität; Vergleich mit  $VALUE(obj_1) = VALUE(obj_2)$ .
  - Anmerkung: bei Spaltenobjekten, wie z.B. Instanzen von GeoCoord, wäre ein Vergleich auf Objektidentität nicht sinnvoll.
  - der Vergleich  $obj_1 = obj_2$  basiert auf der MAP bzw. ORDER-Methode.
- ⇒ Fehler beim Compilieren, falls keine solche definiert ist!
- ⇒ falls eine solche definiert ist, wird sie verwendet, und damit “betragsgleiche” Objekte als gleich behandelt!
- ⇒ “Echten” Vergleich ggf. ausprogrammieren auf Basis der “Schlüsselattribute” der Objekte (ggf. in equals(...)-Methode).

### 9.1.5 Objektrelationale Modellierung: Zusammenfassung

- Objekte anstatt Tupel oder Attributwerte
- Anfragen praktisch unverändert gegenüber rein relationaler DB (insb. Beziehungen weiterhin über Schlüssel/Fremdschlüssel und Join-basierte Anfragen)
- zusätzlich Methoden, Ordnungsmethoden.

### BEISPIEL/AUFGABE

(wird auf Folie 279 analog ausprogrammiert)

- City, Country, Organization als Objekttypen und -tabellen
- Komfortablere Methoden: Mitgliedschaften werden über Methoden eingetragen und abgefragt (ohne Berücksichtigung der Arten der Mitgliedschaft):

`organization.isMember(carcode)`

`country.isMemberIn(org-abbrev)`

`organization.makeMember(carcode)`

`country.makeMemberIn(org-abbrev)`

Interne Implementierung z.B. über die bekannte Tabelle `isMember`.

Hinweis: Boolesche Anfragen der Art "Ist  $x$  Mitglied in  $y$ " sind damit möglich. Es ist jedoch keine Methode "alle Mitglieder von  $y$  möglich – diese müsste eine Relation bzw. Menge zurückgeben.

- ... man kann aber diese Implementierung dann auch beliebig ändern.

## 9.2 Objektorientierte Modellierung

... soweit dienen die Datentypen im wesentlichen zur Bereitstellung von spezialisiertem Verhalten:

- Built-in: DATE
- zusammengesetzt: Geo-Koordinaten
- Geschachtelte Tabellen (parametrisierter Datentyp)
- benutzerdefinierte Objekttypen
- Grundlage für Datentypen wie XMLType etc.

### Objektorientierte Modellierung

Geht über die Nutzung als "Datentypen" hinaus ...

- ... zu Modellierungsaspekten:
- Spezialisierung: Klassenhierarchie; Subtypen als Spezialisierung allgemeiner Typen.
- Objekt-Identität und Referenzen auf Objekte als Werte von Attributen zum Ausdrücken von Beziehungen,
- Objekte: *Wertattribute* und *Referenzattribute*.
- Anfragen durch Navigation etc. (⇒ unsymmetrisch)

## OBJEKTFERENZEN

- Weiterer Datentyp für Attribute: Referenzen auf Objekte  
`<ref-attr> REF <object-datatype>`
- *Objekttyp* als Ziel der Referenz.
- nur Objekte, die eine OID besitzen – also Zeilenobjekte einer Objekttable – können referenziert werden.
- Erzeugen einer Referenz (Selektieren einer OID):

```
SELECT ..., REF(<var>), ...
FROM <object-table> <var>
WHERE ... ;
```

## OBJEKTFERENZEN: CONSTRAINTS

- Objekttyp kann in verschiedenen Tabellen vorkommen: Einschränkung auf eine bestimmte Tabelle bei der Deklaration der entsprechenden Tabelle als Spalten- oder Tabellenconstraints mit **SCOPE**:
  - als Spaltenconstraint (nur bei Tupeltabellen):
 

```
<ref-attr> REF <object-datatype>
SCOPE IS <object-table>
```
  - als Tabellenconstraint:
 

```
SCOPE FOR (<ref-attr>) IS <object-table>
```
- problemlose Integration referentieller Integritätsbedingungen von Objekttabellen zu bestehenden relationalen Tabellen.
- PRIMARY KEYS dürfen keine REF-Attribute umfassen.
- Referentielle Integritätsbedingung
 

```
FOREIGN KEY (<ref-attr>) REFERENCES <object-table>
```

 geht auch, wenn auf <object-table> kein Primary Key definiert ist (verwendet Object-ID).

## Beispiel: Objekttyp *Organization*

```
CREATE TYPE Member_Type AS OBJECT
  (Country VARCHAR2(4),
   Type VARCHAR2(60));
/
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,
   hasHqIn REF CityORType,
   MEMBER FUNCTION isMember (the_country IN VARCHAR2)
   -- EU.isMember('SLO') = 'membership applicant'
   RETURN VARCHAR2,
   MEMBER FUNCTION people RETURN NUMBER,
   MEMBER FUNCTION numberOfMembers RETURN NUMBER,
   MEMBER PROCEDURE addMember
     (the_country IN VARCHAR2, the_type IN VARCHAR2));
/
```

[Filename: Obj/org-type.sql]



Beispiel: Objekttyp *Organization*

Tabellendefinition:

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

- Type Body noch nicht definiert.  
Weiter erstmal nur mit einem Objekt als Beispiel:

Einfügen unter Verwendung des Objektconstructors:

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

Setzen des Referenzattributes *hasHqIn*:

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

[Filename (alles zusammen): Obj/org-table.sql]

SELEKTION VON OBJEKTATTRIBUTEN

- Wertattribute

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

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

- Referenzattribute:

```
SELECT <ref-attr-name>
```

liefert OID:

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

Name	Abbrev	hasHqIn
European Community	EU	<oid>

- Deref(<oid>) liefert das zugehörige Objekt:

```
SELECT Abbrev, Deref(hasHqIn)
FROM Organization_ObjTab;
```

Abbrev	hasHqIn
EU	CityORType('Bruxelles', 'Bruxelles', 'B', 168576, GeoCoord(4.35, 50.85))

## VERWENDUNG VON REFERENZATTRIBUTEN

- Attribute und Methoden eines referenzierten Objekts werden durch *Pfadausdrücke* der Form

```
SELECT <ref-attr-name>.<attr-name>
```

adressiert (“*navigierender Zugriff*”).

- Aliasing mit einer Variablen um den Pfadausdruck eindeutig zu machen:

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

Abbrev	hasHqIn.Name
EU	Bruxelles

- Auch erlaubt:

```
SELECT name, Deref(hasHqIn).name
FROM Organization_ObjTab;
```

### Die Funktionen VALUE, REF, Deref

Mit REF und Deref lässt sich VALUE ersetzen:

```
SELECT VALUE(org) FROM Organization_ObjTab org;
```

und

```
SELECT Deref(REF(org)) FROM Organization_ObjTab org;
```

## ZYKLISCHE REFERENZEN

Die Modellierung als Objektgraph (d.h., Beziehungen nicht durch Tabellen, sondern als Objektreferenzen) führt oft zu Zyklen:

- City\_Type: country REF Country\_Type
- Country\_Type: capital REF City\_Type
- Deklaration jedes Datentypen benötigt bereits die Definition des anderen.
- Definition von *unvollständigen* Typen  
“Forward-Deklaration”:  

```
CREATE TYPE <name>;
/
```
- wird später durch eine komplette Typdeklaration ergänzt.

## UNVOLLSTÄNDIGE DATENTYPEN

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

```
CREATE OR REPLACE TYPE City_type;
/
```

- Die Nutzung in Referenzen ist damit erlaubt:

```
CREATE TYPE city_list AS TABLE OF REF City_type;
/
```

```
CREATE OR REPLACE TYPE Country_Type AS OBJECT
(Name VARCHAR2(50),
 Code VARCHAR2(4),
 Capital REF City_Type);
/
```

- Die *direkte Nutzung* wäre erst erlaubt, wenn City\_type komplett ist:

```
CREATE TYPE city_list_2 AS TABLE OF City_type;
/ -- waere eine Tabelle von City-Objekten
CREATE OR REPLACE TYPE Country_Type_2 AS OBJECT
(Name VARCHAR2(50),
 Code VARCHAR2(4),
 Capital City_Type);
/ -- Capital waere ein Spaltenobjekt
```

## ZYKLISCHE REFERENZEN: BEISPIEL

```
CREATE OR REPLACE TYPE City_Type
/
```

```
CREATE OR REPLACE TYPE Country_Type AS OBJECT
(Name VARCHAR2(50),
 Code VARCHAR2(4),
 Capital REF City_Type,
 Area NUMBER,
 Population NUMBER);
/
```

```
CREATE OR REPLACE TYPE Province_Type AS OBJECT
(Name VARCHAR2(50),
 Country REF Country_Type,
 Capital REF City_Type,
 Area NUMBER,
 Population NUMBER);
/
```

```
CREATE OR REPLACE TYPE City_Type AS OBJECT
(Name VARCHAR2(50),
 Province REF Province_Type,
 Country REF Country_Type,
 Population NUMBER,
 Coordinates GeoCoord);
/
```

## OBJEKTORIENTIERUNG: MODELLIERUNGSASPEKTE

- Beziehungen durch Referenzattribute,
- Anfragen per Navigation (anstatt Join),
- können nur in einer Richtung verfolgt werden,
- erfordert also doppelte Speicherung,
- müssen auf beiden Seiten separat konsistent gehalten werden.

### Beispiel/Aufgabe

- City, Country, Organization als Objektgraph
- Beziehungen immer über Methoden behandeln:
  - organization.isMember(*carcode*)
  - country.isMemberIn(*org-abbrev*)
  - organization.makeMember(*carcode*)
  - country.makeMemberIn(*org-abbrev*)
- Interne Implementierung von z.B. Mitgliedschaften wie oben als Collection von Referenzen, oder über die bekannte Tabelle `isMember`.

## 9.3 Methoden: Funktionen und Prozeduren

TYPE BODY enthält die Implementierungen der Methoden in PL/SQL

### Anpassung von PL/SQL an Objektrelationale Features

- **PL/SQL unterstützt keine Navigation entlang Pfadausdrücken** (in SQL ist es erlaubt).
- Jede MEMBER METHOD besitzt einen *impliziten* Parameter SELF, der das jeweilige Host-Objekt referenziert.
- Tabellenwertige Attribute können innerhalb PL/SQL wie PL/SQL-Tabellen behandelt werden:
  - Built-in Methoden für Collections (PL/SQL-Tabellen) können auch auf tabellenwertige Attribute angewendet werden:
  - `<attr-name>.COUNT`: Anzahl der in der geschachtelten Tabelle enthaltenen Tupel
  - Verwendung in in PL/SQL eingebetteten SQL-Statements – z.B. `SELECT <attr>.COUNT` – nicht erlaubt.
- Weitere Erweiterung: Java (siehe Folie 324).

## Member-Methods: Beispiel

```

CREATE OR REPLACE TYPE BODY Organization.Type IS
MEMBER FUNCTION isMember (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;

```

(bitte umblättern)

## Member-Methods: Beispiel (Forts.)

```

MEMBER FUNCTION numberOfMembers RETURN NUMBER
IS
BEGIN
    IF SELF.Members IS NULL THEN RETURN 0; END IF;
    RETURN Members.COUNT;
END;

MEMBER PROCEDURE addMember
    (the_country IN VARCHAR2, the_type IN VARCHAR2) IS
BEGIN
    IF NOT SELF.isMember(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;
/

```

[Filename: Obj/orgs-type-body.sql]

- FROM THE(SELECT ...) kann nicht durch FROM SELF.Members ersetzt werden (PL/SQL vs. SQL).

## METHODENAUFGRUFE

### Funktionen

- MEMBER FUNCTIONS können in SQL und PL/SQL durch `<object>.<function>(<argument-list>)` selektiert werden.
- parameterlose Funktionen: `<object>.<function>()`
- aus SQL: `<object>` ist durch einen Pfadausdruck mit Alias gegeben.

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

(noch ist die Tabelle aber nicht sinnvoll gefüllt ...)

### Prozeduren

- MEMBER PROCEDURES können nur aus PL/SQL mit `<objekt>.<procedure>(<argument-list>)` aufgerufen werden.

⇒ freie Prozeduren in PL/SQL, um MEMBER PROCEDURES aufzurufen

### Beispiel: Freie Prozedur

```
CREATE OR REPLACE PROCEDURE makeMember
(the_org IN VARCHAR2, the_country IN VARCHAR2,
the_type IN VARCHAR2) IS
n NUMBER;
x 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 x
FROM Organization_ObjTab org
WHERE Abbrev = the_org;
IF x.isMember(the_country)='no' THEN
x.addMember(the_country, the_type);
END IF;
END;
/
```

[Filename: Obj/makemember.sql]

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

**Beispiel: Füllen der Objekttablelle**

Übertragung des Datenbestandes aus den relationalen Tabellen *Organization* und *isMember* in die Objekttablelle *Organization\_ObjTab*:

```

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 isMember)
  LOOP makeMember(the_membership.organization,
                  the_membership.country,
                  the_membership.type);
  END LOOP;
END;
/
EXECUTE Insert_All_Members;
UPDATE Organization_ObjTab org
SET hasHqIn =
  (SELECT REF(cty)
   FROM ORCity cty, Organization old
   WHERE org.Abbrev = old.Abbreviation
   AND cty.Name = old.City
   AND cty.Province = old.Province
   AND cty.Country = old.Country);

```

[Filename: Obj/fill-organizations.sql]

**Beispiel: Nutzung freier Methoden**

```

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

```

[Filename: Obj/is-member.sql]

```

SELECT isMemberIn('EU', 'TR')
FROM DUAL;

```

<b>isMemberIn('EU', 'TR')</b>
candidate

Es ist (zumindest bis ORACLE 12c) nicht möglich, durch Navigation mit Pfadausdrücken Tabelleninhalte zu verändern:

```

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

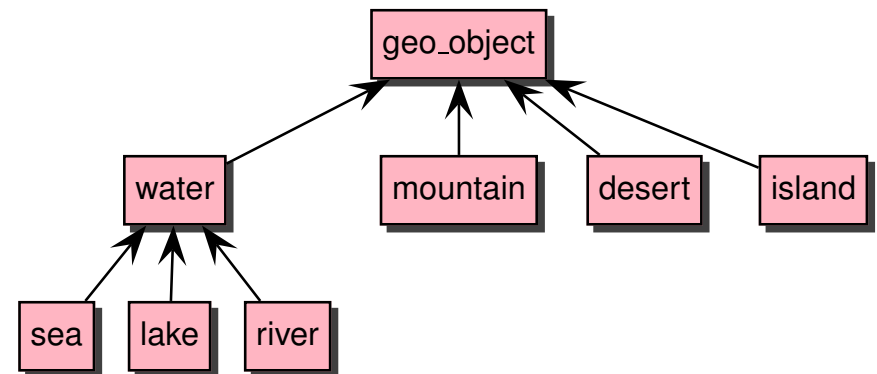
```

### MODELLIERUNG VS. IMPLEMENTIERUNG

- Das Beispiel illustriert Objektmethoden und ihre Anbindung durch freie Methoden am *objektorientierten* Szenario:
    - Headquarter als Referenz,
    - Mitglieder als geschachtelte Tabelle,
  - man kann dieselben Methoden auch mit einer *objektrelationalen* Tabelle OROrganization, und Ablegen der Mitgliedschaften in der Relation isMember implementieren (Aufgabe).
- ⇒ für den Benutzer bleiben die freien Methoden dieselben.

### 9.4 Klassenhierarchie und Vererbung

(Abschnitt ist optional)





## KLASSENHIERARCHIE UND VERERBUNG

- seit ORACLE 9i
- **Typhierarchie:**
- **Subtyp:** abgeleitet von einem Elterntyp
- **Vererbung:** Verbindung zwischen Subtypen und ihren Obertypen in einer Typhierarchie
- Subtypen: "Spezialisierung"
  - hinzufügen neuer Attribute und Methoden
  - **overriding** (Überschreiben) von geerbten Methoden
- Spezielle Eigenschaften von Klassen:
  - abstrakte** (NOT INSTANTIABLE) und **finale** (FINAL) Klassen
- abstrakte Klassen können **abstrakte Methoden** (NOT INSTANTIABLE) haben
- Klassen können **finale Methoden** haben: diese können in Subklassen nicht überschrieben werden

## ERZEUGEN VON SUBTYPEN

Abstrakte Klasse für geographische Objekte  
(die alle einen Namen besitzen):

```
CREATE OR REPLACE TYPE geo_object_type AS OBJECT (
    name VARCHAR2(50),
    MEMBER FUNCTION get_name RETURN VARCHAR2,
    NOT INSTANTIABLE
    MEMBER FUNCTION set_name RETURN VARCHAR2
)
NOT INSTANTIABLE -- DEFAULT: INSTANTIABLE
NOT FINAL;      -- DEFAULT: FINAL
/
```

```
CREATE OR REPLACE TYPE BODY geo_object_type IS
    MEMBER FUNCTION get_name RETURN VARCHAR2
    IS BEGIN RETURN name; END;
    -- no implementation for set_name
    -- (to be class-specific)
    END;
/
```

## ERZEUGEN VON SUBTYPEN

```
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;
/
```

- + Angabe eines TYPE BODY, der bla implementiert.

## ERZEUGEN VON SUBTYPEN

- finale Klassen für Meere, Seen und Flüsse etc.
- müssen alle bis jetzt nicht implementierten Methoden anbieten
- erfordert **OVERRIDING**

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

- + Angabe eines TYPE BODY, der set\_name implementiert.
- optional kann man auch bla überschreiben.
- analog für Meere, Flüsse, Berge, Inseln und Wüsten.

## TABELLEN ÜBER ALLGEMEINEN KLASSEN

- eine Tabelle für alle geographischen Objekte

```
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, elevation, 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;
```

## ANFRAGEN AN TABELLEN ÜBER ALLGEMEINEN KLASSEN

- die Tabelle `geo_obj` ist eine Kollektion von Objekten der Klasse `geo_obj_type` (abstrakt)
- enthält Instanzen der finalen Subklassen, z.B. Flüsse und Berge.
- **Substituierbarkeit:**  
 “Ein Objekt eines Typs  $t$  kann überall auftreten, wo ein Objekt eines Obertyps von  $t$  erwartet wird”
  - Zeilenobjekte in Objekttabellen
  - Spaltenobjekte (objektwertige Attribute)
  - Referenzattribute
  - Argumente und Rückgabewerten von Methoden
- `select name from geo_obj;`  
 da alle `geo_objects` einen Namen haben.

## ANFRAGEN AN KLASSENSPEZIFISCHE EIGENSCHAFTEN

- Kollektion von Instanzen einer abstrakten Klasse
- Auswahl der Objekte einer speziellen Subklasse
- Verwendung von klassenspezifischen Eigenschaften
- ähnlich wie in C++/Java: Typumwandlungen

## SPEZIELLSTE KLASSENZUGEHÖRIGKEIT

- `SYS_TYPEID(<object>)`  
ergibt die **ID** der speziellsten Klasse, zu der ein Objekt gehört
- herausfinden des Klassennamens in `all_types`

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

type_name	typeid	supertype_name
mountain	08	geo_object

## TYPTESTS

- `<object> IS OF(<type>)`  
testet ob `<object>` vom Typ `<type>` ist.
- normalerweise testet man Zugehörigkeit zu einem Subtyp des für die Tabelle bekannten Typs.
- Ausgeben aller Namen von Bergen:  

```
SELECT x.name
FROM geo_obj x
WHERE value(x) IS OF (mountain_type);
```
- wie bekommt man die Namen und die Höhe?  

```
SELECT x.name, x.elevation
```

  
ist nicht erlaubt  
(`geo_objects` haben keine Höhe!)

## TYPUMWANDLUNGEN

- **TREAT** (<object> AS <type>)  
behandelt <object> als eine Instanz des Typs <type>
- falls möglich
- sonst: NULL

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

## 9.5 Diverses zu Objekttypen

### ÄNDERUNGEN AN OBJEKTYPEN

Benutzerdefinierte Typen können mit ALTER TYPE verändert werden:

- Hinzunehmen und Löschen von Attributen
- Hinzunehmen und Löschen von Methoden
- Modifikation eines numerischen Attributs (Länge, Präzision)
- VARCHAR kann verlängert werden
- Ändern der FINAL- und INSTANTIABLE-Eigenschaften

**ALTER TYPE** <type>

```
ADD ATTRIBUTE <name> <datatype>,  
DROP ATTRIBUTE <name>,  
MODIFY ATTRIBUTE <name> <datatype>,  
ADD MEMBER FUNCTION/PROCEDURE <method-spec>  
    -- dann CREATE TYPE BODY neu machen!  
DROP MEMBER FUNCTION/PROCEDURE <method-spec>  
    <options>
```

## ÄNDERUNG VON TYPDEFINITIONEN: ABHÄNGIGKEITEN

Objekttypen-Definitionen und Referenzattribute erzeugen einen Graphen, der dem von Fremdschlüsseldefinitionen erzeugten ähnlich ist.

- Abhängige Schemaobjekte, die einen Typ referenzieren sind z.B.:
  - Tabellen
  - Typen, insb. Subtypen
  - PL/SQL: Prozeduren, Funktionen, Trigger
  - Views, Objekt-Views
- Veränderungen: `ALTER TYPE`
- Propagieren von Änderungen: `CASCADE`
- Compilierbare abhängige Datenbankobjekte (PL/SQL, Sichten, ...): `INVALIDATE` werden als *invalid* markiert und bei der nächsten Benutzung neu compiliert.
- Tabellen: neue Attribute werden mit NULLwerten initialisiert.

Die Datenbank muss nach Typveränderungen revalidiert werden (siehe Handbücher).

## INDEXE AUF OBJEKTATTRIBUTEN

Indexe können auch auf Objektattributen erstellt werden:

```
CREATE INDEX <name>
```

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

- Indexe können *nicht* über komplexen Attributen erstellt werden:

```
-- nicht erlaubt:
```

```
CREATE INDEX city_index
```

```
ON City_ObjTab(coordinates);
```

- Indexe können über elementare Teilattribute eines komplexen Attributes erstellt werden:

```
CREATE INDEX city_index
```

```
ON City_ObjTab(coordinates.Latitude,  
coordinates.Longitude);
```

- Funktions-basierte Indexe:

```
CREATE INDEX name ON
```

```
Organization_Obj_Tab (numberOfMembers);
```

arbeiten mit vorberechneten Werten.

## ZUGRIFFSRECHTE AUF OBJEKTE

Recht an Objekttypen:

```
GRANT EXECUTE ON <Object-datatype> TO ...
```

- bei der Benutzung eines Datentyps stehen vor allem die Methoden (u.a. die entsprechende Konstruktormethode) im Vordergrund.

## REFERENTIELLE INTEGRITÄT

- Vgl. FOREIGN KEY ... REFERENCES ... ON DELETE/UPDATE CASCADE
- Veränderungen an Objekten:  
OID bleibt unverändert  
→ referentielle Integrität bleibt gewahrt.
- Löschen von Objekten:  
*dangling references* möglich.

Überprüfung durch

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

Verwendung z.B. in einem AFTER-Trigger:

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

## 9.6 Object-Views

- maßgeschneiderte Object-Views mit sehr weitgehender Funktionalität

**Legacy-Datenbanken:** Integration bestehender Datenbanken in ein "modernes" objektorientiertes Modell:

*Objekt-Views* über relationale Ebene legen:  
"Objekt-Abstraktionen"

**Effizienz + Benutzerfreundlichkeit:**

Die relationale Repräsentation ist oft effizienter:

- Geschachtelte Tabellen intern als separate Tabellen gespeichert.
- $n : m$ -Beziehungen: gegenseitige geschachtelte Tabellen notwendig.

⇒ Definition eines relationalen Basisschemas mit Object-Views.

**Einfache Modifizierbarkeit:** CREATE OR REPLACE TYPE und ALTER TYPE nur sehr eingeschränkt

⇒ Veränderungen durch Neudefinition geeigneter Object-Views abfangen.

Häufige Empfehlung: Object Views mit geschachtelten Tabellen, Referenzen etc. auf Basis eines relationalen Grundschemas verwenden.

## OBJECT-VIEWS

Benutzer führt seine Änderungen auf dem durch die Objektviews gegebenen externen Schema durch.

- enthalten Zeilenobjekte, d. h. hier werden neue Objekte *definiert*.
- Abbildung direkter Änderungen (INSERT, UPDATE und DELETE) durch INSTEAD OF-Trigger auf das darunterliegende Schema.
- Benutzer darf erst gar keine solchen Statements an das View stellen. Entsprechende Funktionalität durch Methoden der Objekttypen, die die Änderungen direkt auf den zugrundeliegenden Basistabellen ausführen.

### Syntax

- durch WITH OBJECT OID <attr-list> wird angegeben, wie die Objekt-ID berechnet werden soll.
- Verwendung von CAST und MULTISSET.

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

- in <select-statement> wird *kein* Objektkonstruktor verwendet!



**OBJECT VIEWS: *Country***

```
CREATE OR REPLACE TYPE Country_Type AS OBJECT
(Name          VARCHAR2(50),
 Code          VARCHAR2(4),
 Capital       REF City_Type,
 Area          NUMBER,
 Population    NUMBER);
/
```

Sinnvollerweise würde man hier gleich auch noch Methoden definieren.

```
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;
```

**OBJECT VIEWS: WAS NICHT GEHT**

- Object View darf keine geschachtelte Tabelle und
- kein Ergebnis einer funktionalen Methode einer zugrundeliegenden Tabelle enthalten.

Object View auf Basis von *Organization\_ObjTab*:

```
CREATE OR REPLACE TYPE Organization_Ext_Type AS OBJECT
(Name VARCHAR2(80),
 Abbrev VARCHAR2(12),
 Members Member_List_Type,
 established DATE,
 hasHqIn REF City_Type,
 numberOfPeople NUMBER);
/
```

```
CREATE OR REPLACE VIEW Organization_ObjV
OF Organization_Ext_Type
AS
SELECT Name, Abbrev, Members, established,
       hasHqIn, org.people()
FROM Organization_ObjTab org;
```

FEHLER in Zeile 3:

ORA-00932: nicht übereinstimmende Datentypen

*Beide* angegebenen Attribute sind auch einzeln nicht erlaubt.

## 9.7 Objektorientierung und Data Dictionary

- TABS = USER\_TABLES(table\_name, ...): alle *relationalen* Tabellen die dem Benutzer gehören
- USER\_OBJECT\_TABLES(table\_name, table\_type, nested...): alle Objekttabellen die dem Benutzer gehören. table\_type gibt den Objekttyp an, aus dessen Instanzen die Tabelle besteht. Hierzu gehören auch Nested Tables!
- USER\_ALL\_TABLES(table\_name, ...): alle Tabellen die dem Benutzer gehören.
- USER\_OBJECTS(object\_name, object\_type, status): alle Datenbankobjekte, die dem Benutzer gehören.
- Objekttabellen, deren Type mit DROP TYPE ... FORCE gelöscht wurde, stehen nicht mehr in USER\_OBJECT\_TABLES. Sie existieren aber noch (neues CREATE TABLE ergibt eine Fehlermeldung). (Bsp. nächste Folie)
- USER\_TYPES(type\_name): alle Objekttypen, die dem Benutzer gehören.

### Löschen von Datenbankobjekten

```
CREATE TYPE bla AS OBJECT (x NUMBER);
/
CREATE TABLE blatab OF bla;
SELECT table_name, table_type FROM user_object_tables;
-- name: blatab, type: bla
DROP TYPE bla FORCE;
SELECT table_name, table_type FROM user_object_tables;
-- nichts
CREATE TYPE bla AS OBJECT (x NUMBER);
/
CREATE TABLE blatab OF bla;
-- ORA-00955: name is already used by an existing object
SELECT * FROM user_objects WHERE object_name='BLATAB';
-- blatab - invalid
SELECT object_name, object_type, status from user_objects;
SELECT object_name, object_type, status from user_objects
WHERE object_type = 'TABLE';
```

Problem: alle Tabellen löschen (Skript drop-all-tables)

- Verwendung von user\_tables ignoriert invalide Tabellen.
- Nested Tables können nicht mit DROP TABLE gelöscht werden, sondern nur mit der Tabelle zu der sie gehören.
- Verwendung von user\_objects where object\_type = 'TABLE' bricht ab, wenn es eine (möglicherweise invalide, also in user\_object\_tables auch nicht mehr als nested gelistete) Nested Table löschen soll.

## 9.8 Fazit

- *Objektrelationale Tabellen* (Folie 242):  
Kompatibilität mit den grundlegenden Konzepten von SQL.  
U.a. Fremdschlüsselbedingungen von objektrelationalen Tabellen zu relationalen Tabellen.
- *Objektorientiertes Modell* (Folie 268):  
... etwas kompliziert zu handhaben.
- *Object/Objekt-Relationale Views* (Folie 302):  
erlauben ein objektorientiertes externes Schema.  
Benutzer-Interaktionen werden durch Methoden und  
INSTEAD OF-Trigger auf das interne Schema umgesetzt.  
Implementierung auf relationaler Basis.
- *Objekttypen-Konzept* als Basis für (vordefinierte, in Java implementierte Klassen als) Datentypen zur Behandlung von nicht-atomaren Werten (XML (siehe Folie 384), Multimedia etc.).