Objekt-Relationale Datenbanksysteme

Integration von relationalen Konzepten und Objektorientierung:

- Komplexe Datentypen: Erweiterung des Domain-Konzepts von SQL-2
- Abstrakte Datentypen ("Objekttypen"): Objekt-Identität und Kapselung interner Funktionalität.
- Spezialisierung: Klassenhierarchie; Subtypen als Spezialisierung allgemeiner Typen.
- Tabellen als Subtabellen von Tabellen.
- Funktionen als Bestandteile eines ADT's oder von Tabellen, oder freie Funktionen.
- Methoden- und Funktionsaufrufe im Rahmen von `SELECT`-Ausdrücken.

Objektorientierung

- Unterscheidung zwischen dem *Zustand* und *Verhalten* eines *Objektes*.
- in ORACLE 8: Tabellen von Tupeln vs. *Object Tables* aus Objekten
- Im Gegensatz zu einem *Tupel* besitzt ein Objekt *Attribute* (beschreiben seinen Zustand) und *Methoden* (Abfragen und Ändern des Zustandes).
- Typ definiert gemeinsame Signatur seiner Instanzen (Objekte)
- bereits behandelt: Komplexe Attributtypen. Besitzen nur *Wertattribute*, keine Methoden.
- Methoden: *Prozeduren* und *Funktionen*
- `MAP`/`ORDER`-Funktion: Ordnung auf Objekttyp
- Spalten einer Tupeltabelle können *objektwertig* oder *referenzwertig* sein.
- Objekte: *Wertattribute* und *Referenzattribute*.
- ORACLE8: keine Subtypen und Vererbung

Typdeklaration: Attribute, *Signaturen* der Methoden, `READ`/`WRITE` Zugriffscharakteristik.

Type Body: Implementierung der Methoden in PL/SQL.

Objekttypdeklaration

```
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>]
   );
```

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

`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: werden nur ausgeführt, wenn *zugesichert* ist, 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

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

```
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)
   );
/
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

Type Body

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

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;
/
```


Erzeugung von Objekten

- Konstruktormethode:

  `<type>(<arg_1>, ..., <arg_n>)`

Methodenaufruf

(aus einem PL-SQL Programm)

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

mit `SELF` für <object> kann ein Objekt seine eigenen Methoden aufrufen.

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(0,51));
  END;

END;
/
```


Spaltenobjekte

- Attribut eines Tupels (oder eines Objekts) ist objektwertig,
- *keine* OID, also *nicht* referenzierbar.

Beispiel: Geo-Koordinaten

```
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)));
```

- Constraints werden wie immer bei der Tabellendefinition angegeben.

```
INSERT INTO Mountain
   VALUES ('Feldberg', 1493, GeoCoord(8,48));
SELECT Name, mt.coordinates.distance(geocoord(0,90))
FROM Mountain mt;
```

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

Zeilenobjekte

- Elemente von *Objekttabellen*,
- erhalten eine eindeutige OID und sind damit referenzierbar.
- OID entspricht dem *Primärschlüssel* und wird mit den (weiteren) Integritätsbedingungen bei der Tabellendefinition angegeben.
- problemlose Integration referentieller Integritätsbedingungen von Objekttabellen zu bestehenden relationalen Tabellen.

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

<constraint-list>:

- attributbezogene Bedingungen entsprechen den Spaltenbedingungen:

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

- Tabellenbedingungen: Syntax wie bei Tupeltabellen.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Zeilenobjekte

Beispiel: *City_Type*

```
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));
/
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;
/
```

Objekttabellen: Zeilenobjekte

- der (mehrspaltige) Primärschlüssel wird als Tabellenbedingung angegeben,
- Primärschlüssel darf keine Referenzattribute umfassen,
- Die Fremdschlüsselbedingung auf die relationale Tabelle *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.

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


Praktikum: Datenbankprogrammierung in SQL/ORACLE

Verwendung von Objekten

- Zeilenobjekt *als Ganzes* selektieren,

  ```
  VALUE (<var>)
  ```

  in Kombination mit Aliasing

  ```
  FROM <tabelle> <var>
  ```

- z.B. Vergleich oder in einer ORDER BY-Klausel,

Beispiel

```
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', 'Nordrh.-Westf., 'D', 293072, GeoCoord(8, 50))
City_Type('Stuttgart', 'Baden-Wuertt., 'D', 588482, GeoCoord(9, 49))
⋮

Verwendung von Objekten: VALUE

- Objekte auf Gleichheit testen
- Objekt als Argument einer Methode

```
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);
```

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

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


Objektreferenzen

- Weiterer Datentyp für Attribute: Referenzen auf Objekte

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

- PRIMARY KEYs dürfen keine REF-Attribute umfassen.
- *Objekttyp* als Ziel der Referenz
- nur Objekte, die eine OID besitzen – also Zeilenobjekte einer Objekttabelle – können referenziert werden.
- 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>
    ```

- Erzeugen einer Referenz (Selektieren einer OID):

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

Beispiel: Objekttyp *Organization*

```
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,
   has_hq_in REF City_Type,
   MEMBER FUNCTION is_member (the_country IN VARCHAR2)
  -- EU.is_member('SL') = 'membership applicant'
      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))
/
```


Beispiel: Objekttyp *Organization*

Tabellendefinition:

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

Einfügen unter Verwendung des Objektkonstruktors:

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

Referenzattribut *has_hq_in*:

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

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, has_hq_in
FROM Organization_ObjTab;
```

Name	Abbrev	has_hq_in
European Community	EU	<oid>

- `DEREF(<oid>)` liefert das zugehörige Objekt:

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

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

Praktikum: Datenbankprogrammierung in SQL/ORACLE

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.has_hq_in.name
FROM Organization_ObjTab org;
```

Abbrev	has_hq_in.Name
EU	Brussels

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

- `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

Zyklische Referenzen: Beispiel

```
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);
/
```

Unvollständige Datentypen

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

```
CREATE TYPE City_type;
/
```

erlaubt:

```
CREATE TYPE city_list AS TABLE OF REF City_type;
/
CREATE OR REPLACE TYPE Country_Type AS OBJECT
  (Name VARCHAR2(32),
   Code VARCHAR2(4),
   Capital REF City_Type);
/
```

erst erlaubt, wenn city_type komplett ist:

```
CREATE TYPE city_list AS TABLE OF City_type;
/
CREATE OR REPLACE TYPE Country_Type AS OBJECT
  (Name VARCHAR2(32),
   Code VARCHAR2(4),
   Capital City_Type);
/
```


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;
  ```

Methoden: Funktionen und Prozeduren

- `TYPE BODY` enthält die Implementierungen der Methoden in PL/SQL
- PL/SQL an geschachtelte Tabellen und objektorientierte Features angepaßt.
- 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.
- zukünftige Erweiterung: Java


```
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 0; END IF;
  RETURN Members.COUNT;
END;

MEMBER PROCEDURE add_member
   (the_country IN VARCHAR2, the_type IN VARCHAR2) IS
BEGIN
  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;
/
```

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

Methodenaufrufe: 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.is_member('D')
  FROM Organization_ObjTab org
  WHERE NOT org.is_member('D') = 'no';
```

- `MEMBER PROCEDURES` können nur aus PL/SQL mit `<objekt>.<procedure>(<argument-list>)` aufgerufen werden.
- freie Prozeduren in PL/SQL, um `MEMBER PROCEDURES` aufzurufen

Methodenaufrufe: Prozeduren

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


Praktikum: Datenbankprogrammierung in SQL/ORACLE

Übertragung des Datenbestandes aus den relationalen Tabellen *Organization* und *is_member* in die Objekttabelle *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 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 has_hq_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);
```

Verwendung von Objekten

```
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;
/
```

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

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

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


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.
- ORACLE 8: für jeden Objekttyp eine MAP FUNCTION oder ORDER FUNCTION.

MAP-Funktion:

- 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:

- 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 erfordern PRAGMA RESTRICT_REFERENCES (<name>,WNDS,WNPS,RNPS,RNDS), d.h. sie 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(0,51));
  END;
END;
/
```

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


Praktikum: Datenbankprogrammierung in SQL/ORACLE

`MAP`-Methoden

Viele Operationen sind in `MAP`-Methoden nicht erlaubt:

- keine Anfragen an die Datenbank:

  In *Organization_Type* kann *People* nicht als `MAP`-Methode verwendet werden.

- keine built-in Methoden von geschachtelten Tabellen:

  In *Organization_Type* kann *number_of_members* ebenfalls nicht als `MAP`-Methode verwendet werden.

ORDER-Methoden

- Vergleich von SELF mit einem anderen Objekt desselben Typs, das formal als Parameter angegeben wird.
- Ergebnis -1 (SELF < Parameter), 0 (Gleichheit) oder 1 (SELF > Parameter)
- Wird ORDER BY angegeben, werden die Ausgabeobjekte paarweise verglichen und entsprechend der ORDER-Methode geordnet.
- Ein Beispiel hierfür ist etwa 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).

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.Longitude,
                  coordinates.Latitude);
```

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.

Änderungen von Objekttypen

- Verwendung von Objekttypen und Referenzattributen erzeugt ein ähnliches Netz von Referenzen wie bei der Definition von referentiellen Integritätsbedingungen.
- Änderungen an Objekttypen in ORACLE 8.0 sehr restriktiv: `CREATE OR REPLACE TYPE` und `ALTER TYPE` sind (zumindest in ORACLE 8.0) nicht erlaubt, wenn der Objekttyp irgendwo verwendet wird.

! nicht möglich, einem irgendwo verwendeten Objekttyp ein Attribut (oder auch nur eine Methode!) hinzuzufügen.

In conclusion, carefully plan the object types for your database so that you get things right the first time. Then keep your fingers crossed and hope that things do not change once you have everything up and running (ORACLE 8: Architecture).

Ein erstes Fazit

- Daten*haltung* in einem objektorientierten Schema bereits bei kleinen Änderungen sehr problematisch.
- anwendungsorientierte (nicht-relationale) Repräsentation durch Methoden sowie freie Prozeduren und Funktionen.
- Integration von anwendungsspezifischer Funktionalität durch Objektmethoden vereinfacht.

⇒ Datenhaltung: relational
Nutzung: objektorientiert.

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:
relationale Repräsentation 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 Definition 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.

- 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.

Objektrelationale Views

- Tupel-Views ohne Methoden:

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

- `SELECT`-clause: zusätzlich Konstruktormethoden für Objekte und geschachtelte Tabellen.
- Für die Erzeugung geschachtelter Tabellen für Object Views werden die Befehle `CAST` und `MULTISET` verwendet.

Beispiel

```
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;
```


Objektviews

- enthalten Zeilenobjekte, d.h. hier werden neue Objekte *definiert*.
- durch `WITH OBJECT OID <attr-list>` wird angegeben, wie die Objekt-ID berechnet wird werden soll.
- Verwendung von `CAST` und `MULTISET`.

```
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(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;
```


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,
   has_hq_in REF City_Type,
   number_of_people NUMBER);
/
```

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

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

Beide angegebenen Attribute sind auch einzeln nicht erlaubt.

Fazit

+ Kompatibilität mit den grundlegenden Konzepten von ORACLE 7.
  U.a. Fremdschlüsselbedingungen von Objekttabellen zu relationalen Tabellen.
+ *Object Views* erlauben ein objektorientiertes externes Schema. Benutzer-Interaktionen werden durch Methoden und `INSTEAD OF`-Trigger auf das interne Schema umgesetzt.
– Flexibilität/Ausgereiftheit:
  Typen können nicht verändert/erweitert werden.
  (inkrementelle!) Anpassungen des Schemas nicht möglich.