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 281)
    - `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 330).

Stufen der Objektorientierung

"Konservative" objektrelationale Erweiterungen (seit Oracle 8)

(siehe Folie 206)

- 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 229)

- Beziehungen nicht mehr über Schlüssel/Fremdschlüssel sondern über Referenzen
  $\Rightarrow$ 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>(attr_1, ..., attr_n))
  ```

  anstatt

  ```
  INSERT INTO <table>
  VALUES(attr_1, ..., attr_n)
  ```

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

9.1.1 Definition von Objekttypen

Typdeklaration

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

Typ-Implementierung

- **Type Body:** Implementierung der Methoden in PL/SQL
- seit Oracle 8i auch in PL/SQL+Java (siehe Folien 281 und 285)

OBJEKTTYPDEKLARATION

```
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>]
  );
/
```

- <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: 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

- 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)
  );
/
```

[Filename: ObjRel/geocoord-type.sql]

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.

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(0, 51));
  END;
END;
/
```

[Filename: ObjRel/geocoord-body.sql]

ERZEUGUNG VON OBJEKTEN

- Konstruktormethode:

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

  Also kein `NEW`, sondern nur einfach

```
GeoCoord(8,48)
CityType('Berlin', 'Berlin', 'D',
         3472009, GeoCoord(13.3,52,45))
```

METHODENAUFRUF

- 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(-30,52.45).Distance(geoCoord(-31,52.45))
FROM DUAL;
SELECT geoCoord(-30,0).Distance(geoCoord(-31,0))
FROM DUAL;
```


9.1.2 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,
   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)));
```

[Filename: ObjRel/mountain-table.sql]

- 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.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

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: ***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,
   MEMBER FUNCTION NoOfOrganizations
      RETURN NUMBER,
   PRAGMA RESTRICT_REFERENCES
      (Distance, WNPS, WNDS, RNPS, RNDS),
   PRAGMA RESTRICT_REFERENCES
      (NoOfOrganizations, WNPS, WNDS, RNPS));
/
```

[Filename: ObjRel/citytype.sql]

ZEILENOBJEKTE

Beispiel: ***City_Type***

```
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;
  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/citytypebody.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 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 ORCity
SELECT City_Type
  (Name, Province, Country, Population,
   GeoCoord(Longitude, Latitude))
FROM City
WHERE Country = 'D'
  AND NOT Longitude IS NULL;
```

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

VERWENDUNG VON OBJEKTTABELLEN

Auslesen und Ändern von Attributen wie bekannt

- Auslesen:

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

- Ändern:

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

Methodenaufrufe wie vermutet

```
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';
```

Die VALUE()-Funktion

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

selektiert ein Objekt als Objekt:

```
SELECT VALUE(cty)
FROM ORCity 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 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

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

9.1.3 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: (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 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,
       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` < Parameter,
  - 0 (Gleichheit), oder
  - $x > 0$ falls `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).

9.1.4 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 239 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.is_member($carcode$)
  country.is_member_in(org-$abbrev$)
  organization.make_member($carcode$)
  country.make_member_in(org-$abbrev$)

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

  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.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

9.2 Objektorientierte Modellierung

... soweit dienten 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. ($\Rightarrow$ unsymmetrisch)

OBJEKTREFERENZEN

- 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 Objekttabelle – können referenziert werden.
- problemlose Integration referentieller Integritätsbedingungen von Objekttabellen zu bestehenden relationalen Tabellen.
- `PRIMARY KEY`s dürfen keine `REF`-Attribute umfassen.
- 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('SLO') = '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));
/
```

[Filename: Obj/org-type.sql]

Beispiel: Objekttyp *Organization*

Tabellendefinition:

```
CREATE TABLE Organization_ObjTab OF Organization_Type
  (Abbrev PRIMARY KEY,
   SCOPE FOR (has_hq_in) 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 Objektkonstruktors:

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

Setzen des Referenzattributes *has_hq_in*:

```
UPDATE Organization_ObjTab
SET has_hq_in =
  (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, 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))

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

Die Funktionen VALUE, REF, DEREF

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

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(32),
   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(32),
   Code VARCHAR2(4),
   Capital City_Type);
/ -- Capital waere ein Spaltenobjekt
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

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);
/
```

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.is_member($carcode$)
  country.is_member_in($org\text{-}abbrev$)
  organization.make_member($carcode$)
  country.make_member_in($org\text{-}abbrev$)

- Interne Implementierung von z.B. Mitgliedschaften wie oben als Collection von Referenzen, oder über die bekannte Tabelle `is_member`.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

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 281).

Member-Methods: Beispiel

```
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;                                    (bitte umblättern)
```


Member-Methods: Beispiel (Forts.)

```
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;
/
```

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

- `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';
```

(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 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 org
   FROM Organization_ObjTab o
   WHERE Abbrev = the_org;
  IF o.is_member(the_country)='no' THEN
   o.add_member(the_country, the_type);
  END IF;
END;
/
```

[Filename: Obj/makemember.sql]

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

Beispiel: Füllen der Objekttabelle

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

[Filename: Obj/is-member.sql]

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

is_member_in('EU', 'CZ')
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';
```

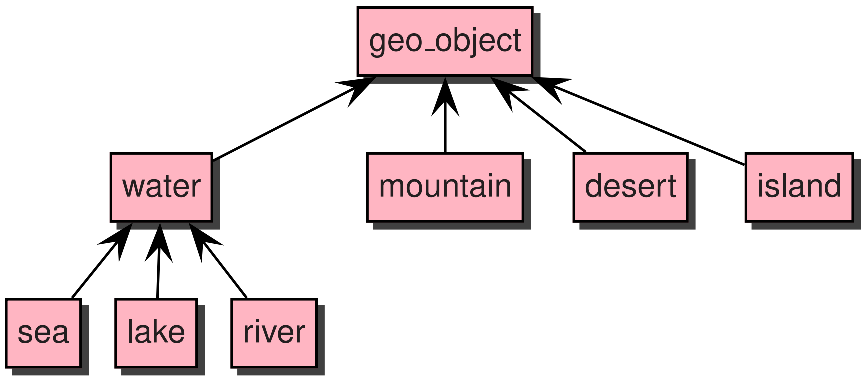
MODELLIERUNG VS. IMPLEMENTIERUNG

- 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 `is_member` 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(25),
   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, 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;
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

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

Praktikum: Datenbankprogrammierung in SQL/ORACLE

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.height`

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


9.5 Diverses zu Objekttypen

Änderungen an Objekttypen

Benutzerdefinierte Typen können mit `ALTER TYPE` verändert werden
(seit ORACLE 9):

- 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>
      -- requires new CREATE TYPE BODY!
  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 nächten Benutzung neu compiliert.
- Tabellen: neue Attribute werden mit `NULL`werten 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.Longitude,
                  coordinates.Latitude);
```

- Funktions-basierte Indexe:

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

  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 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);
/
```

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,
   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.