TEIL III: Erweiterungen

Teil I: Grundlagen

Teil II: Diverses

Teil III: Prozedurale Konzepte, OO, Einbettung

- PL/SQL: Prozeduren, Funktionen, Trigger
- Objektorientierung
- SQL und Java
- SQL und XML

SITUATION

- keine prozeduralen Konzepte in SQL (Schleifen, Verzweigungen, Variablendeklarationen)
- viele Aufgaben nur umständlich über Zwischentabellen oder überhaupt nicht in SQL zu realisieren.
  - Transitive Hülle.
- Programme repräsentieren anwendungsspezifisches Wissen, das nicht in der Datenbank enthalten ist.

ERWEITERUNGEN

- Einbettung von SQL in prozedurale Wirtssprachen (*embedded SQL*); meistens Pascal, C, C++, oder auch Java (JDBC/SQLJ),
- Erweiterung von SQL um prozedurale Elemente *innerhalb* der SQL-Umgebung, *PL/SQL* (*Procedural language extensions to SQL*).
- Vorteile von PL/SQL: Bessere Integration der prozeduralen Elemente in die Datenbank; Nutzung in Prozeduren, Funktionen und Triggern.
- benötigt für Objektmethoden.

Kapitel 8
Prozedurale Erweiterungen: PL/SQL

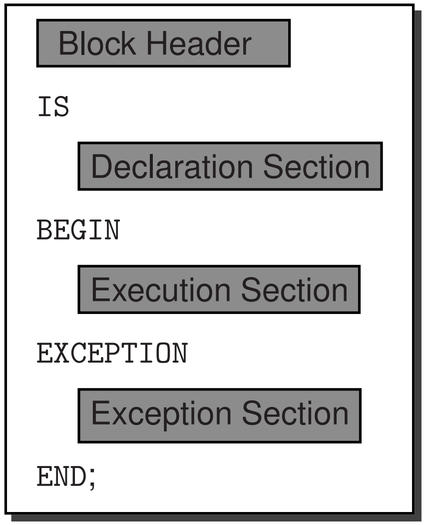
- Erweiterung von SQL um prozedurale Elemente *innerhalb* der SQL-Umgebung, *PL/SQL* (*Procedural language extensions to SQL*).
- “Stored Procedures/Functions” innerhalb der DB
- direkter Zugriff auf Datenbankinhalt
- Vorteile von PL/SQL: Bessere Integration der prozeduralen Elemente in die Datenbank; Nutzung in Prozeduren, Funktionen und Triggern

Weitere Nutzung

- Programmierung von Objektmethoden (seit Oracle 8/1997)

8.1 Prozeduren, Funktionen und Kontrollstrukturen in PL/SQL

Blockstruktur von PL/SQL



- Block Header: Art des Objekts (Funktion, Prozedur oder *anonym* (innerhalb eines anderen Blocks)), und Parameterdeklarationen.
- Declaration Section: Deklarationen der in dem Block verwendeten Variablen,
- Execution Section: Befehlssequenz des Blocks,
- Exception Section: Reaktionen auf eventuell auftretende Fehlermeldungen.

Einfache, Anonyme Blöcke

- nur Declaration und Execution Section
- werden direkt ausgeführt
- `DECLARE ... BEGIN ... END; /`

Ausgabe-Generierung

- verwendet das DBMS_Output Package
- einmalig SET SERVEROUTPUT ON (z.B., beim Starten von sqlplus)
- innerhalb von PL/SQL-Blocks:

  `dbms_output.put_line('bla');`
- Bei Prozeduren etc: Ausgabe erscheint erst *nach* kompletter Ausführung der Prozedur etc.

```
set serveroutput on;
DECLARE
bla NUMBER;
BEGIN
 bla := 42;
 dbms_output.put_line(bla);
END;
/
```

[Filename: PLSQL/output.sql]

Prozeduren

```
CREATE [OR REPLACE] PROCEDURE <proc_name>
  [(<parameter-list>)]
  IS <pl/sql-body>;
/
```

- `OR REPLACE`: existierende Prozedurdefinition wird überschrieben.
- `(<parameter-list>)`: Deklaration der formalen Parameter:

  ```
  (<variable> [IN|OUT|IN OUT] <datatype>,
    ⋮
   <variable> [IN|OUT|IN OUT] <datatype>)
  ```
- `IN`, `OUT`, `IN OUT`: geben an, wie die Prozedur/Funktion auf den Parameter zugreifen kann (Lesen, Schreiben, beides).
- Default: `IN`.
- Bei `OUT` und `IN OUT` muss beim Aufruf eine Variable angegeben sein, bei `IN` ist auch eine Konstante erlaubt.
- `<datatype>`: alle von PL/SQL unterstützten Datentypen; *ohne* Längenangabe (`VARCHAR2` anstelle `VARCHAR2(20)`).
- `<pl/sql-body>` enthält die Definition der Prozedur in PL/SQL.

FUNKTIONEN

Analog, zusätzlich wird der Datentyp des Ergebnisses angegeben:

```
CREATE [OR REPLACE] FUNCTION <funct_name>
  [(<parameter-list>)]
  RETURN <datatype>
  IS <pl/sql body>;
/
```

- PL/SQL-Funktionen werden mit

  ```
  RETURN <ausdruck>;
  ```

  verlassen. Jede Funktion muss mindestens ein `RETURN`-Statement im `<body>` enthalten.
- Eine Funktion darf keine Seiteneffekte auf die Datenbasis haben (siehe Oracle-Dokumentation *PL/SQL User's Guide and Reference*).

Wichtig: nach dem Semikolon noch ein Vorwärtsslash ("/"), um die Deklaration auszuführen!!!

Im Falle von "... created with compilation errors":

```
SHOW ERRORS;
```

ausgeben lassen.

Prozeduren und Funktionen können mit `DROP PROCEDURE/FUNCTION <name>` gelöscht werden.

PROZEDUREN UND FUNKTIONEN

- Aufruf von Prozeduren im PL/SQL-Skript:
  `<procedure> (arg1,...,argn);`
  (wenn ein formaler Parameter als OUT oder IN OUT angegeben ist, muss das Argument eine Variable sein)
- Aufruf von Prozeduren in SQLPlus:
  `execute <procedure> (arg1,...,argn);`
- Verwendung von Funktionen in PL/SQL:
  `... <function> (arg1,...,argn) ...`
  wie in anderen Programmiersprachen.

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

```
SELECT <function> (arg1,...,argn)
FROM DUAL;
```

Beispiel: Prozedur

- Einfache Prozedur: PL/SQL-Body enthält nur SQL-Befehle

Informationen über Länder sind über mehrere Relationen verteilt.

```
CREATE OR REPLACE PROCEDURE InsertCountry
 (name VARCHAR2, code VARCHAR2,
  area NUMBER, pop NUMBER,
  gdp NUMBER, inflation NUMBER, pop_growth NUMBER)
IS
BEGIN
  INSERT INTO Country (Name,Code,Area,Population)
         VALUES (name,code,area,pop);
  INSERT INTO Economy (Country,GDP,Inflation)
         VALUES (code,gdp,inflation);
  INSERT INTO Population (Country,Population_Growth)
         VALUES (code,pop_growth);
END;
/
```

[Filename: PLSQL/insertcountry.sql]

```
EXECUTE InsertCountry
   ('Lummerland', 'LU', 1, 4, 50, 0.5, 0.25);
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

Beispiel: Funktion

- Einfache Funktion: Einwohnerdichte eines Landes

```
CREATE OR REPLACE FUNCTION Density (arg VARCHAR2)
RETURN number
IS
  temp number;
BEGIN
  SELECT Population/Area
    INTO temp
    FROM Country
    WHERE code = arg;
  RETURN temp;
END;
/
```

[Filename: PLSQL/density.sql]

```
SELECT Density('D')
FROM dual;
```

PL/SQL-VARIABLEN UND DATENTYPEN.

Deklaration der PL/SQL-Variablen in der Declaration Section:

```
DECLARE
<variable> <datatype> [NOT NULL] [DEFAULT <value>];
⋮
<variable> <datatype> [NOT NULL] [DEFAULT <value>];
```

Einfache Datentypen:

`BOOLEAN`: `TRUE`, `FALSE`, `NULL`,

`BINARY_INTEGER`, `PLS_INTEGER`: Ganzzahlen mit Vorzeichen.

`NATURAL`, `INT`, `SMALLINT`, `REAL`, ...: Numerische Datentypen.

```
DECLARE
anzahl NUMBER DEFAULT 0;
name VARCHAR2(30);
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

anchored TYPDEKLARATION

Angabe einer PL/SQL-Variablen, oder Tabellenspalte (!) deren Typ man übernehmen will:

```
<variable> <variable'>%TYPE
  [NOT NULL] [DEFAULT <value>];
oder
<variable> <table>.<col>%TYPE
  [NOT NULL] [DEFAULT <value>];
```

- `cityname City.Name%TYPE`
- `%TYPE` wird zur Compile-Time bestimmt.

ZUWEISUNG AN VARIABLEN

- “klassisch” innerhalb des Programms:

```
a := b;
```

- Zuweisung des (einspaltigen und einzeiligen!) Ergebnisses einer Datenbankanfrage an eine PL/SQL-Variable:

```
SELECT ...
INTO <PL/SQL-Variable>
FROM ...
```

```
DECLARE
cname country.name%TYPE;
BEGIN
 SELECT name
 INTO cname
 FROM country
 WHERE code='D';
 dbms_output.put_line(cname);
END;
/
```

[Filename: PLSQL/simple.sql]

PL/SQL-DATENTYPEN: RECORDS

Ein RECORD enthält mehrere Felder, entspricht einem Tupel in der Datenbasis:

```
TYPE city_type IS RECORD
  (Name City.Name%TYPE,
   Country VARCHAR2(4),
   Province VARCHAR2(32),
   Population NUMBER,
   Longitude NUMBER,
   Latitude NUMBER);

the_city city_type;
```

anchored Typdeklaration für Records

Records mit Tabellenzeilen-Typ deklarieren: %ROWTYPE:

```
<variable> <table-name>%ROWTYPE;
```

Äquivalent zu oben:

```
the_city city%ROWTYPE;
```

Zuweisung an Records

- Aggregierte Zuweisung: zwei Variablen desselben Record-Typs:

  ```
  <variable> := <variable'>;
  ```

- Feldzuweisung: ein Feld wird einzeln zugewiesen:

  ```
  <record.feld> := <variable>|<value>;
  ```

- `SELECT INTO`: Ergebnis einer Anfrage, die *nur ein einziges Tupel* liefert:

  ```
  SELECT ...
  INTO <record-variable>
  FROM ... ;
  ```

```
DECLARE
c continent%ROWTYPE;
BEGIN
 SELECT *
 INTO c
 FROM continent
 WHERE name='Europe';
 dbms_output.put_line(c.name || ' : ' || c.area);
END;
/
```

[Filename: PLSQL/simple2.sql]

Vergleich von Records

Beim Vergleich von Records muss jedes Feld einzeln verglichen werden.

PL/SQL-Datentypen: PL/SQL Tables

Array-artige Struktur, *eine* Spalte mit beliebigem Datentyp (also auch `RECORD`), normalerweise mit `BINARY_INTEGER` indiziert.

```
TYPE <type> IS TABLE OF <datatype>
  [INDEX BY BINARY_INTEGER];

<var> <type>;
```

```
plz_table_type IS TABLE OF City.Name%TYPE
  INDEX BY BINARY_INTEGER;

plz_table plz_table_type;
```

- Adressierung: `<var>(1)`

  ```
  plz_table(79110):= Freiburg;
  plz_table(33334):= Kassel;
  ```

- *sparse*: nur die Zeilen gespeichert, die Werte enthalten.

Tabellen können auch als Ganzes zugewiesen werden

```
andere_table := plz_table;
```

PL/SQL-Datentypen: PL/SQL Tables (Forts.)

Zusätzlich *built-in*-Funktionen und -Prozeduren:

```
<variable> := <pl/sql-table-name>.<built-in-function>;
```

oder

```
<pl/sql-table-name>.<built-in-procedure>;
```

- `COUNT` (fkt): Anzahl der belegten Zeilen.
  plz_table.count = 2
- `EXISTS(i)` (fkt): `TRUE` falls Zeile `i` der Tabelle nicht leer.
- `DELETE` (proc): Löscht alle Zeilen einer Tabelle.
- `DELETE(i)`: Löscht Zeile `i` einer Tabelle.
- `FIRST/LAST` (fkt): niedrigster/höchster belegter Indexwert.
  plz_table.first = 33334
- `NEXT/PRIOR(n)` (fkt): Gibt ausgehend von n den nächsthöheren/nächstniedrigen belegten Indexwert.
  plz_table.next(33334) = 79110

SQL-Statements in PL/SQL

- DML-Kommandos `INSERT`, `UPDATE`, `DELETE` sowie `SELECT INTO`-Statements.
- Diese SQL-Anweisungen dürfen auch PL/SQL-Variablen enthalten.
- Befehle, die *nur ein einziges Tupel betreffen*, können mit `RETURNING` Werte an PL/SQL-Variablen zurückgeben:

```
UPDATE ... SET ... WHERE ...
RETURNING <expr-list>
INTO <variable-list>;
```

Z.B. Row-ID des betroffenen Tupels zurückgeben:

```
DECLARE tmprowid ROWID;
BEGIN
    :
    INSERT INTO Politics (Country,Independence)
      VALUES (Code,SYSDATE)
      RETURNING ROWID
      INTO tmprowid;
    :
END;
```

- DDL-Statements in PL/SQL nicht direkt unterstützt: DBMS_SQL-Package, `EXECUTE IMMEDIATE`

KONTROLLSTRUKTUREN

- `IF THEN - [ELSIF THEN] - [ELSE] - END IF`,
- verschiedene Schleifen:
- Simple `LOOP`: `LOOP ... END LOOP;`
- `WHILE LOOP`:

  `WHILE <bedingung> LOOP ... END LOOP;`
- Numeric `FOR LOOP`:

  ```
  FOR <loop_index> IN
    [REVERSE] <Anfang> .. <Ende>
   LOOP ... END LOOP;
  ```

  Die Variable `<loop_index>` wird dabei *automatisch* als `INTEGER` deklariert.
- `EXIT [WHEN <bedingung>]`: `LOOP` verlassen.
- den allseits beliebten `GOTO`-Befehl mit Labels:

  `<<label_i>> ... GOTO label_j;`
- `NULL`-Werte verzweigen immer in den `ELSE`-Zweig.
- `GOTO`: nicht von außen in ein `IF`-Konstrukt, einen `LOOP`, oder einen lokalen Block hineinspringen, nicht von einem `IF`-Zweig in einen anderen springen.
- hinter einem Label muss immer mindestens ein ausführbares Statement stehen;
- `NULL` Statement.

GESCHACHTELTE BLÖCKE

Innerhalb der *Execution Section* werden *anonyme Blöcke* zur Strukturierung verwendet. Hier wird die *Declaration Section* mit `DECLARE` eingeleitet (es gibt keinen Block Header):

```
BEGIN
  -- Befehle des äußeren Blocks --
  DECLARE
   -- Deklarationen des inneren Blocks
  BEGIN
   -- Befehle des inneren Blocks
  END;
  -- Befehle des äußeren Blocks --
END;
```

8.2 Cursore/Iteratoren zur Verarbeitung von Ergebnismengen

- Datenbankanfragen: mengenorientiert
- Programmiersprache: variablenbasiert

Design Patterns: Kollektionen und Iteratoren

(vgl. Informatik I)

- Kollektion: Sammlung von Items (Liste, Baum, Heap, Menge)
- Iterator: Hilfsklasse zum Durchlaufen/Aufzählen aller Items
- Methoden:
  - Erzeugen/Initialisieren des Iterators,
  - Weiterschalten, Test, ob noch weitere Elemente vorhanden sind,
  - Zugriff auf ein Element,
  - (Schliessen des Iterators)

... Iteratoren werden im Weiteren immer wieder verwendet.

Cursorbasierter Datenbankzugriff

Zeilenweiser Zugriff auf eine Relation aus einem PL/SQL-Programm.

Cursordeklaration in der *Declaration Section*:

```
CURSOR <cursor-name> [(<parameter-list>)]
IS
  <select-statement>;
```

- `(<parameter-list>)`: Parameter-Liste,
- nur `IN` als Übergaberichtung erlaubt.
- Zwischen `SELECT` und `FROM` auch PL/SQL-Variablen und PL/SQL-Funktionen. PL/SQL-Variablen können ebenfalls in den `WHERE`-, `GROUP`- und `HAVING`-Klauseln verwendet werden.

Beispiel:

Alle Städte in dem in der Variablen `the_country` angegebenen Land:

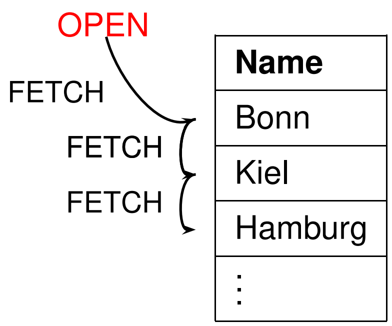
```
DECLARE
CURSOR cities_in (the_country Country.Code%TYPE)
IS SELECT Name
   FROM City
   WHERE Country=the_country;
```

Cursore: Grundprinzip

- OPEN <cursor-name>[(<argument-list>)];

  Erzeugt mit dem gegebenen SELECT-Statement eine *virtuelle Tabelle* mit einem “Fenster”, das über einem Tupel stehen kann und schrittweise vorwärts bewegt wird. Mit OPEN wird die Anfrage ausgeführt und der Cursor initialisiert:

  ```
  OPEN cities_in ('D');
  ```



Cursore: Verwendung

- FETCH <cursor-name> INTO <record-variable>; oder
  FETCH <cursor-name> INTO <variable-list>;
  bewegt den Cursor auf die nächste Zeile des Ergebnisses der Anfrage und kopiert diese in die angegebene Record-Variable oder Variablenliste.

  Diese kann z.B. mit <cursor-name>%ROWTYPE mit dem Record-Typ des Cursors definiert werden:

  <variable> <cursor-name>%ROWTYPE;

- CLOSE <cursor-name>; schließt einen Cursor.

```
DECLARE CURSOR cities_in
   (crs_country Country.Code%TYPE)
  IS SELECT Name
     FROM City
     WHERE Country = crs_country;
city_in cities_in%ROWTYPE;
BEGIN
 OPEN cities_in ('D');
 FETCH cities_in INTO city_in;
 dbms_output.put_line(city_in.Name);
 FETCH cities_in INTO city_in;
 dbms_output.put_line(city_in.Name);
 CLOSE cities_in;
END;
/
```

[Filename: PLSQL/cursor1.sql]

Cursore: Attribute

Kontrolle über die Verarbeitung eines Cursors:

- `<cursor-name>%ISOPEN`: Cursor offen?
- `<cursor-name>%FOUND`: Solange ein Cursor bei der letzten `FETCH`-Operation ein neues Tupel gefunden hat, ist `<cursor-name>%FOUND = TRUE`.
- `<cursor-name>%NOTFOUND`: `TRUE` wenn man alle Zeilen eines Cursors ge`FETCH`t hat.
- `<cursor-name>%ROWCOUNT`: Anzahl der von einem Cursor bereits gelesenen Tupel.
- nicht innerhalb eines SQL-Ausdrucks.

Cursore: Attribute

```
CREATE OR REPLACE PROCEDURE first_city
     (the_country country.code%TYPE)
IS BEGIN
 DECLARE CURSOR cities_in
 (crs_country Country.Code%TYPE)
  IS SELECT Name
     FROM City
     WHERE Country = crs_country;
 city_in cities_in%ROWTYPE;
 BEGIN
   OPEN cities_in (the_country);
   FETCH cities_in INTO city_in;
   IF cities_in%FOUND
   THEN DBMS_OUTPUT.PUT_LINE(city_in.name);
   ELSE DBMS_OUTPUT.PUT_LINE('Nothing found!');
   END IF;
   CLOSE cities_in;
 END;
END;
/
```

[Filename: PLSQL/cursor-attrs.sql]

```
execute first_city('D');
execute first_city('X');
```

- Aufgabe: Programmieren Sie eine explizite WHILE-Schleife, die alle Städte eines Landes ausgibt.

Cursore: Hinweis

nicht möglich:

```
OPEN cities_in ('D');
OPEN cities_in ('CH');
FETCH cities_in INTO <variable>;
```

- *ein* parametrisierter Cursor,
- *nicht* eine Familie von Cursoren!

CURSOR FOR LOOP

Spezielle Schleife zur Iteration über den Inhalt eines Cursors:

```
FOR <record_index> IN <cursor-name>
LOOP ... END LOOP;
```

- `<record_index>` wird dabei *automatisch* als Variable vom Typ `<cursor-name>%ROWTYPE` deklariert,
- `<record_index>` *immer* von einem Record-Type – ggf. einspaltig.
- Es wird automatisch ein `OPEN` ausgeführt,
- bei jeder Ausführung des Schleifenkörpers wird *automatisch* ein `FETCH` ausgeführt,
- → Schleifenkörper enthält i.a. *keinen* `FETCH`-Befehl,
- am Ende wird automatisch ein `CLOSE` ausgeführt,
- Spalten müssen explizit adressiert werden.

Cursor FOR LOOP: Beispiel

Beispiel: Für jede Stadt in dem gegebenen Land soll der Name ausgegeben werden:

```
CREATE OR REPLACE PROCEDURE list_cities
     (the_country country.code%TYPE)
IS
BEGIN
  DECLARE CURSOR cities_in
     (crs_country country.Code%TYPE)
  IS SELECT Name
     FROM City
     WHERE Country = crs_country;
  BEGIN
     FOR the_city IN cities_in(the_country)
     LOOP
          dbms_output.put_line(the_city.name);
     END LOOP;
  END;
END;
/
```

[Filename: PLSQL/cursor-loop1.sql]

```
execute list_cities('D');
```


Eingebetteter Cursor FOR LOOP

- SELECT-Anfrage kann auch direkt in die FOR-Klausel geschrieben werden.

```
CREATE OR REPLACE PROCEDURE list_big_cities
(the_country country.code%TYPE)
IS
BEGIN
  FOR the_city IN
   (  SELECT Name
      FROM City
      WHERE Country = the_country
      AND Population > 1000000 )
  LOOP
   dbms_output.put_line(the_city.Name);
  END LOOP;
END;
/
```

[Filename: PLSQL/cursor-loop2.sql]

```
execute list_big_cities('D');
```

SCHREIBZUGRIFF VIA CURSOR

Mit `WHERE CURRENT OF` <cursor-name> kann man auf das zuletzt von dem genannten Cursor ge`FETCH`te Tupel zugreifen:

```
UPDATE <table-name>
SET <set_clause>
WHERE CURRENT OF <cursor_name>;
```

```
DELETE FROM <table-name>
WHERE CURRENT OF <cursor_name>;
```

- Dabei bestimmt die Positionierung des Cursors bezüglich der Basistabellen den Ort der Änderung (im Gegensatz zu View Updates).

Praktikum: Datenbankprogrammierung in SQL/ORACLE

8.3 Zugriffsrechte auf PL/SQL-Datenbankobjekte

Benutzung von Funktionen/Prozeduren:

- Benutzungsrechte vergeben:
  `GRANT EXECUTE ON <procedure/function> TO <user>;`
- Prozeduren und Funktionen werden jeweils mit den Zugriffsrechten des *Besitzers* ausgeführt.
- nach
  `GRANT EXECUTE ON <procedure/function> TO <user>;`
  kann dieser User die Prozedur/Funktion auch dann aufrufen, wenn er kein Zugriffsrecht auf die dabei benutzten Tabellen hat.
- Möglichkeit, Zugriffsberechtigungen strenger zu formulieren als mit `GRANT ... ON <table> TO ...`: Zugriff nur in einem ganz speziellen, durch die Prozedur oder Funktion gegebenen Kontext.

8.4 Geschachtelte Tabellen unter PL/SQL

Nested_Languages		
Country	**Languages**	
D	German	100
CH	German	65
	French	18
	Italian	12
	Romansch	1
FL	NULL	
F	French	100
⋮	⋮	

Nutzung geschachtelter Tabellen in ORACLE nicht ganz unproblematisch:
"Bestimme alle Länder, in denen Deutsch gesprochen wird, sowie den Anteil der deutschen Sprache in dem Land"

Eine solche Anfrage muss für *jedes* Tupel in *Nested_Languages* die innere Tabelle untersuchen.

- `SELECT THE` kann jeweils nur ein Objekt zurückgeben,
- keine Korrelation mit umgebenden Tupeln möglich.
- Verwendung einer (Cursor-)Schleife.

Geschachtelte Tabellen unter PL/SQL: Beispiel

```
CREATE TABLE tempCountries
 (Land    VARCHAR2(4),
  Sprache VARCHAR2(20),
  Anteil  NUMBER);

CREATE OR REPLACE PROCEDURE Search_Countries
      (the_Language IN VARCHAR2)
IS CURSOR countries IS
   SELECT Code
   FROM Country;
BEGIN
   DELETE FROM tempCountries;
   FOR the_country IN countries
   LOOP
      INSERT INTO tempCountries
      SELECT the_country.code,Name,Percentage
      FROM THE(SELECT Languages
               FROM Nested_Language
               WHERE Country = the_country.Code)
      WHERE Name = the_Language;
   END LOOP;
END;
/

EXECUTE Search_Countries('German');
SELECT * FROM tempCountries;
```

(RE)AKTIVES VERHALTEN

- Bis jetzt: Funktionen und Prozeduren werden durch den Benutzer explizit aufgerufen.
- Trigger: Ausführung wird durch das Eintreten eines Ereignisses in der Datenbank angestossen.

8.5 Trigger

EINSCHUB: INTEGRITÄTSBEDINGUNGEN

- Spalten- und Tabellenbedingungen
- Wertebereichsbedingungen (*domain constraints*),
- Verbot von Nullwerten,
- Uniqueness und Primärschlüssel-Bedingungen,
- `CHECK`-Bedingungen.

! Alles nur als Bedingungen an *eine* Zeile innerhalb *einer* Tabelle formulierbar.

ASSERTIONS

- Bedingungen, die den gesamten DB-Zustand betreffen.

  `CREATE ASSERTION <name> CHECK (<bedingung>)`
- Diese werden allerdings von ORACLE bisher nicht unterstützt.

⇒ Also muss man sich etwas anderes überlegen.

TRIGGER

- spezielle Form von PL/SQL-Prozeduren,
- werden beim Eintreten eines bestimmten Ereignisses ausgeführt.
- Spezialfall aktiver Regeln nach dem **E**vent-**C**ondition-**A**ction-Paradigma.
- einer Tabelle (oft auch noch einer bestimmten Spalte) zugeordnet.
- Bearbeitung wird durch das Eintreten eines Ereignisses (Einfügen, Ändern oder Löschen von Zeilen der Tabelle) ausgelöst (Event).
- Ausführung von Bedingungen an den Datenbankzustand abhängig (Condition).
- Action:
  - *vor* oder *nach* der Ausführung der entsprechenden aktivierenden Anweisung ausgeführt.
  - einmal pro auslösender Anweisung (Statement-Trigger) oder einmal für jede betroffene Zeile (Row-Trigger) ausgeführt.
  - Trigger-Aktion kann auf den alten und neuen Wert des gerade behandelten Tupels zugreifen.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

TRIGGER

```
CREATE [OR REPLACE] TRIGGER <trigger-name>
  BEFORE | AFTER
  {INSERT | DELETE | UPDATE} [OF <column-list>]
  [ OR {INSERT | DELETE | UPDATE} [OF <column-list>]]
   :
  [ OR {INSERT | DELETE | UPDATE} [OF <column-list>]]
  ON <table>
  [REFERENCING OLD AS <name> NEW AS <name>]
  [FOR EACH ROW]
  [WHEN (<condition>)]
  <pl/sql-block>;
```

- `BEFORE`, `AFTER`: Trigger wird vor/nach der auslösenden Operation ausgeführt.
- `OF <column>` (nur für `UPDATE`) schränkt Aktivierung auf angegebene Spalte ein.
- Zugriff auf Zeileninhalte vor und nach der Ausführung der aktivierenden Aktion mittels `OLD` bzw. `NEW`. Schreiben in `NEW`-Werte nur mit `BEFORE`-Trigger.
- `FOR EACH ROW`: Row-Trigger, sonst Statement-Trigger.
- `WHEN (<condition>)`: zusätzliche Bedingung; hier werden `OLD` und `NEW` verwendet; Subqueries an die Datenbank sind nicht erlaubt.
- Referenzieren der Variablen im PL/SQL-Teil als `:OLD` und `:NEW`.

TRIGGER: BEISPIEL

Wenn ein Landes-Code geändert wird, pflanzt sich diese Änderung auf die Relation *Province* fort:

```
CREATE OR REPLACE TRIGGER change_Code
BEFORE UPDATE OF Code ON Country
FOR EACH ROW
BEGIN
  UPDATE Province
  SET Country = :NEW.Code
  WHERE Country = :OLD.Code;
END;
/
```

[Filename: PLSQL/changecode.sql]

```
UPDATE Country
SET Code = 'UK'
WHERE Code = 'GB';

SELECT * FROM Province WHERE Country='UK';
```


TRIGGER: BEISPIEL

Wenn ein Land neu angelegt wird, wird ein Eintrag in *Politics* mit dem aktuellen Jahr erzeugt:

```
CREATE TRIGGER new_Country
AFTER INSERT ON Country
FOR EACH ROW
WHEN (:NEW.population > 2)
BEGIN
   INSERT INTO Politics (Country,Independence)
   VALUES (:NEW.Code,SYSDATE);
END;
/
```

[Filename: PLSQL/newcountry.sql]

```
INSERT INTO Country (Name,Code,Population)
  VALUES ('Lummerland', 'LU', 4);

SELECT * FROM Politics WHERE country='LU';
```

TRIGGER: MUTATING TABLES

- Zeilenorientierte Trigger: immer direkt vor/nach der Veränderung einer Zeile aufgerufen
- jede Ausführung des Triggers sieht einen anderen Datenbestand der Tabelle, auf der er definiert ist, sowie der Tabellen, die er evtl. ändert
- $\leadsto$ Ergebnis *abhängig von der Reihenfolge* der veränderten Tupel

ORACLE: Betroffene Tabellen werden während der gesamten Aktion als *mutating* gekennzeichnet, können nicht von Triggern gelesen oder geschrieben werden.

Nachteil: Oft ein zu strenges Kriterium.

- Trigger soll auf Tabelle zugreifen auf der er selber definiert ist.
  - Nur das auslösende Tupel soll von dem Trigger gelesen/geschrieben werden: Verwendung eines `BEFORE`-Triggers und der `:NEW`- und `:OLD`-Variablen
  - Es sollen neben dem auslösenden Tupel auch weitere Tupel verwendet werden: Verwendung eines Statement-orientierten Triggers
- Trigger soll auf andere Tabellen zugreifen: Verwendung von Statement-Triggern und ggf. Hilfstabellen.

INSTEAD OF-TRIGGER

- *View Updates*: Updates müssen auf Basistabellen umgesetzt werden.
- View-Update-Mechanismen eingeschränkt.
- `INSTEAD OF`-Trigger: Änderung an einem View wird durch andere SQL-Anweisungen ersetzt.

```
CREATE [OR REPLACE] TRIGGER <trigger-name>
  INSTEAD OF
  {INSERT | DELETE | UPDATE} ON <view>
  [REFERENCING OLD AS <name> NEW AS <name>]
  [FOR EACH STATEMENT]
  <pl/sql-block>;
```

- Keine Einschränkung auf bestimmte Spalten möglich
- Keine `WHEN`-Klausel
- Default: `FOR EACH ROW`

View Updates und INSTEAD OF-Trigger

```
CREATE OR REPLACE VIEW AllCountry AS
SELECT Name, Code, Population, Area,
       GDP, Population/Area AS Density,
       Inflation, population_growth,
       infant_mortality
FROM Country, Economy, Population
WHERE Country.Code = Economy.Country
  AND Country.Code = Population.Country;
```

[Filename: PLSQL/allcountry-view.sql]

```
INSERT INTO AllCountry
 (Name, Code, Population, Area, GDP,
  Inflation, population_growth, infant_mortality)
 VALUES ('Lummerland','LU',4,1,0.5,0,25,0);
```

[Filename: PLSQL/insert-allcountry.sql]

Fehlermeldung: Über ein Join-View kann nur *eine* Basistabelle modifiziert werden.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

View Updates und INSTEAD OF-Trigger

```
CREATE OR REPLACE TRIGGER InsAllCountry
INSTEAD OF INSERT ON AllCountry
FOR EACH ROW
BEGIN
  INSERT INTO
    Country (Name,Code,Population,Area)
  VALUES (:NEW.Name, :NEW.Code,
          :NEW.Population, :NEW.Area);
  INSERT INTO Economy (Country,Inflation)
  VALUES (:NEW.Code, :NEW.Inflation);
  INSERT INTO Population
      (Country, Population_Growth,infant_mortality)
  VALUES (:NEW.Code, :NEW.Population_Growth,
          :NEW.infant_mortality);
END;
/
```

[Filename: PLSQL/instead-of.sql]

- aktualisiert *Country*, *Economy* und *Population*.
- Trigger *New_Country* (`AFTER INSERT ON COUNTRY`) aktualisiert zusätzlich *Politics*.

Fehlerbehandlung

- Declaration Section: Deklaration (der Namen) benutzerdefinierter Exceptions.

```
DECLARE <exception> EXCEPTION;
```

- Exception Section: Definition der beim Auftreten einer Exception auszuführenden Aktionen.

```
WHEN <exception>
   THEN <PL/SQL-Statement>;
WHEN OTHERS THEN <PL/SQL-Statement>;
```

- Exceptions können dann an beliebigen Stellen des PL/SQL-Blocks durch `RAISE` ausgelöst werden.

```
IF <condition>
  THEN RAISE <exception>;
```

Ablauf

- auslösen einer Exception
- entsprechende Aktion der `WHEN`-Klausel ausführen
- innersten Block verlassen (oft Anwendung von anonymen Blöcken sinnvoll)

Trigger/Fehlerbehandlung: Beispiel

Nachmittags dürfen keine Städte gelöscht werden:

```
CREATE OR REPLACE TRIGGER nachm_nicht_loeschen
BEFORE DELETE ON City
BEGIN
  IF TO_CHAR(SYSDATE,'HH24:MI')
       BETWEEN '12:00' AND '18:00'
  THEN RAISE_APPLICATION_ERROR
           (-20101,'Unerlaubte Aktion');
  END IF;
END;
/
```

[Filename: PLSQL/trigger-nachmittag.sql]

BEISPIEL

```
CREATE OR REPLACE TRIGGER bla
INSTEAD OF INSERT ON AllCountry
FOR EACH ROW
BEGIN
 IF user='may'
  THEN NULL;
 END IF;
...
END;
/

INSERT INTO AllCountry
 (Name, Code, Population, Area, GDP, Inflation,
  population_growth, infant_mortality)
 VALUES ('Lummerland','LU',4,1,0.5,0,25,0);

1 Zeile wurde erstellt.

SQL> select * from allcountry where Code='LU';

Es wurden keine Zeilen ausgewaehlt.
```

(aus A. Christiansen, M. Höding, C. Rautenstrauch und G. Saake, ORACLE 8 effizient einsetzen, Addison-Wesley, 1998)

8.6 Weitere PL/SQL-Features

- *Packages*: Möglichkeit, Daten und Programme zu kapseln;
- `FOR UPDATE`-Option bei Cursordeklarationen;
- *Cursorvariablen*;
- *Exception Handlers*;
- *benannte* Parameterübergabe;
- PL-SQL Built-in Funktionen: Parsing, String-Operationen, Datums-Operationen, Numerische Funktionen;
- Built-in Packages.

- Definition komplexer Transaktionen,
- Verwendung von `SAVEPOINT`s für Transaktionen.

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 Abschn. 277)
    * `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 325).

Praktikum: Datenbankprogrammierung in SQL/ORACLE

STUFEN DER OBJEKTORIENTIERUNG

"Konservative" objektrelationale Erweiterungen (seit Oracle 8)

(siehe Folie 203)

- 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 225)

- 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 Foliee 277 und 281)

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]

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.
- 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;
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

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.

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]

Praktikum: Datenbankprogrammierung in SQL/ORACLE

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 in Folie 235 analog ausprogrammiert)

- City, Country, Organization als Objekttypen und -tabellen
- Komfortablere Methoden: Mitgliedschaften werden über Methoden eingetragen und abgefragt:

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

  Interne Implementierung über die bekannte Tabelle is_member.
- ... man kann aber diese Implementierung dann auch beliebig ändern.

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. (⇒ 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 KEYs 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;
/
CREATE OR REPLACE TYPE Country_Type_2 AS OBJECT
  (Name VARCHAR2(32),
   Code VARCHAR2(4),
   Capital City_Type);
/
```

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-abbrev*)
  organization.make_member(*carcode*)
  country.make_member_in(*org-abbrev*)

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

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 277).

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 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;
/
```

[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
 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;
/
```

[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';
```

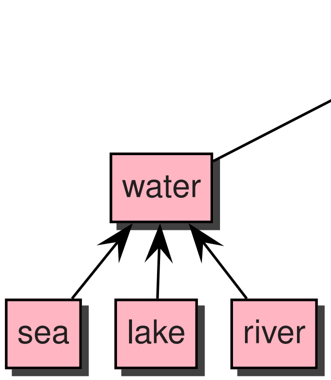

MODELLIERUNG VS. IMPLEMENTIERUNG

- 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.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

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;
```

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 Subtyps 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.

9.7 Fazit

- *Objektrelationale Tabellen* (Folie 203):
  Kompatibilität mit den grundlegenden Konzepten von SQL.
  U.a. Fremdschlüsselbedingungen von objektrelationalen Tabellen zu relationalen Tabellen.
- *Objektorientiertes Modell* (Folie 225):
  ... etwas kompliziert zu handhaben.
- *Object/Objekt-Relationale Views* (Folie 258):
  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 325), Multimedia etc.).